Binary Code Security

Fall 2024

R. Sekar

Inline Reference Monitoring

- Foundations
  - Software Fault Isolation (SFI)
  - Control-flow Integrity (CFI)
- Examples
  - Google Native Client (NaCl)
  - Web Assembly (Wasm)
  - eBPF

Inline Reference Monitors (IRMs)

- Provide finer granularity
  - "Variable x is always greater than y"
  - "Never dereference an invalid pointer"
  - Provides much more expressive power

Inline Reference Monitors (IRMs)

- Provide finer granularity
  - “Variable x is always greater than y”
  - “Never dereference an invalid pointer”
  - Provides much more expressive power
- Efficient
  - Does not require a switching execution contexts
    - unlike the crossing of user/kernel or process boundaries

Inline Reference Monitors (IRMs)

- Provide finer granularity
  - "Variable x is always greater than y"
  - "Never dereference an invalid pointer"
  - Provides much more expressive power
- Efficient
  - Does not require a switching execution contexts
    - unlike the crossing of user/kernel or process boundaries
- Key challenge:
  - Protecting IRM from hostile code

Securing RMs in the same address space

- Protect RM data that is used in enforcing policy
  - Software-based fault isolation (SFI)

Securing RMs in the same address space

- Protect RM data that is used in enforcing policy
  - Software-based fault isolation (SFI)
- Protect RM checks from being bypassed
  - Control-flow integrity (CFI)

Securing RMs in the same address space

- Protect RM data that is used in enforcing policy
  - Software-based fault isolation (SFI)
- Protect RM checks from being bypassed
  - Control-flow integrity (CFI)
- Note
  - For vulnerability defenses (e.g., Stackguard), we implement the checks using an IRM
  - But we don't worry so much about these properties since we are dealing with benign (and not malicious) code

Fault Isolation

- Goal: Limit negative consequences when “something bad” happens.

Fault Isolation

- Goal: Limit negative consequences when "something bad" happens.
- Why is it needed?
  - Untrusted plug-ins make applications unreliable.
  - Third-party modules make the OS unreliable.

Fault Isolation

- Goal: Limit negative consequences when “something bad” happens.
- Why is it needed?
  - Untrusted plug-ins make applications unreliable.
  - Third-party modules make the OS unreliable.
- Hardware based Fault Isolation
  - Isolated address space
  - Traps instead of cross-domain control transfers

Software Fault Isolation (SFI) [Wahbe et al 1994]

- Motivation: Hardware fault isolation can be expensive
  - Requires context switches that are very expensive on modern hardware
    - flushing of TLB and caches

Software Fault Isolation (SFI) [Wahbe et al 1994]

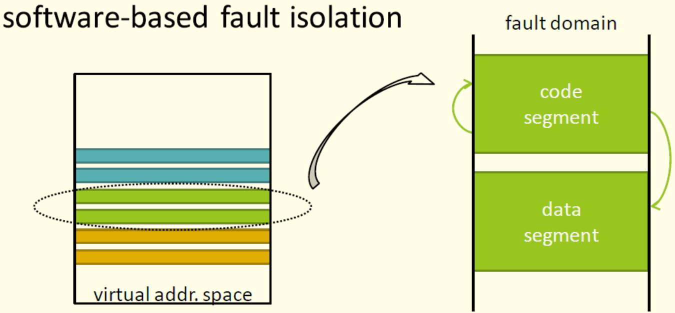
- Motivation: Hardware fault isolation can be expensive
  - Requires context switches that are very expensive on modern hardware
    - flushing of TLB and caches
- SFI idea: Avoid context switches using inline address range checks
  - for data accesses
  - for indirect control-flow transfers (CFT)
    - Note: direct CFTs can be statically checked

Software Fault Isolation (SFI) [Wahbe et al 1994]

- Motivation: Hardware fault isolation can be expensive
  - Requires context switches that are very expensive on modern hardware
    - flushing of TLB and caches
- SFI idea: Avoid context switches using inline address range checks
  - for data accesses
  - for indirect control-flow transfers (CFT)
    - Note: direct CFTs can be statically checked
- SFI challenges
  - Efficiency: each memory access has the overhead of checking
  - Security: Preventing circumvention or subversion of checks

Software-based fault isolation

- Even when running in the same virtual address space, limit some code components to access only a part of the address space
  - This subspace is called a "fault domain"



Software Fault Isolation

- Virtual address segments
  - Fault domain (guest) has two segments, one for code, the other for data.
  - Each segment share a unique upper bits (segment identifier)

Software Fault Isolation

- Virtual address segments
  - Fault domain (guest) has two segments, one for code, the other for data.
  - Each segment share a unique upper bits (segment identifier)
- Components of the technique
  - Isolation
    - Segment matching to limit cross-domain data and code access
    - Optimization: simply override the segment bits
  - Data sharing
  - Domain crossing

Segment Matching

- Insert checking code before every unsafe instruction
  - To prevent subversion of checks, use dedicated registers
    - Need only worry about indirect accesses
    - Don't forget that returns are indirect jumps too
- Checking code verifies that the unsafe instruction has the correct segment identifier

Segment Matching

`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.

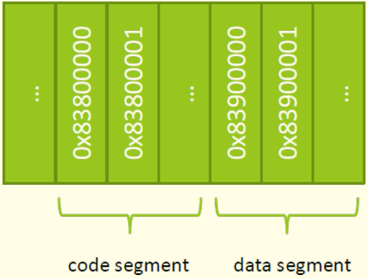
`store instruction uses dedicated-reg`

Optimization 1: Address Sandboxing

`dedicated-reg` ⇐ `target-reg&and-mask-reg`
Use dedicated register `and-mask-reg` *to clear segment identifier bits.*

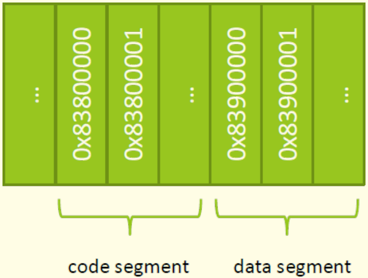
`dedicated-reg` ⇐ `dedicated-reg|segment-reg`
Use dedicated register `segment-reg` *to set segment identifier bits.*

`store instruction uses dedicated-reg`



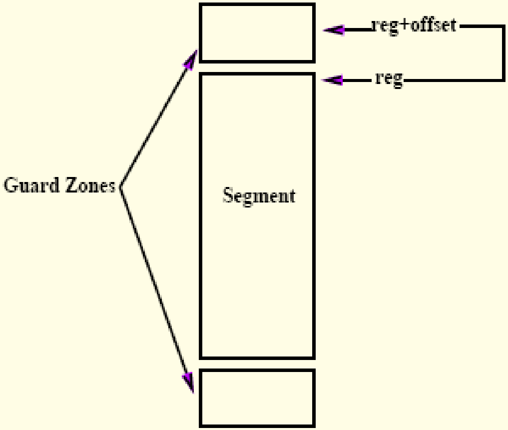
Optimization 1: Address Sandboxing

`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`

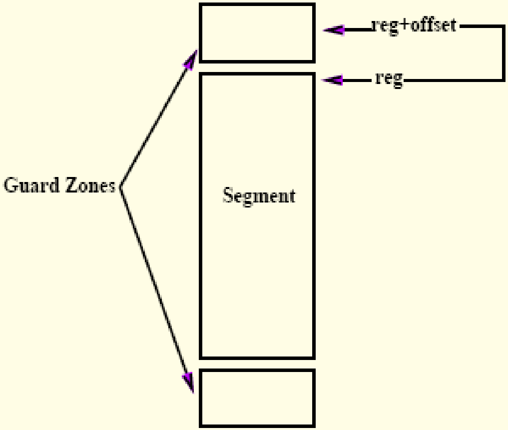


- 3 instructions instead of 5.
- Correctness: Relies on the invariant that dedicated registers always contain valid values before any control transfer instruction.

Optimization 2: Guarding pages



Optimization 2: Guarding pages



- A single instruction accesses multiple bytes of memory (4, 8, or may be more)
- Need to check whether all bytes are within the segment
  - Require at least two checks!
- Optimization
  - Guard zones ensure that reg+offset will also be in bounds

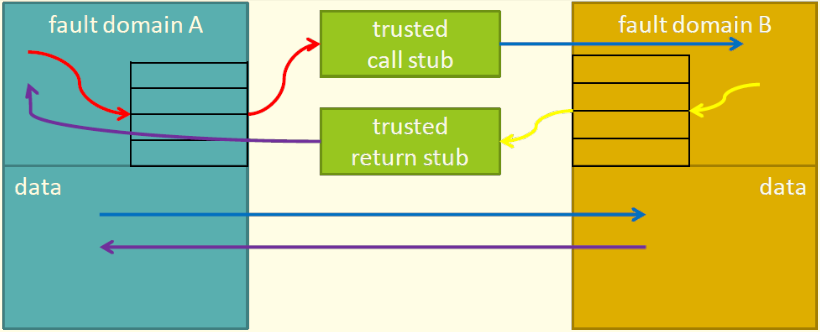
Data sharing

- Read-only sharing can be achieved in several ways:
  - Option 1: Don't restrict read accesses
  - Option 2: Allow reads to access some segments other than that of untrusted code
  - Option 3: Remap shared memory into the address space of both the untrusted and trusted domains

Data sharing

- Read-only sharing can be achieved in several ways:
  - Option 1: Don't restrict read accesses
  - Option 2: Allow reads to access some segments other than that of untrusted code
  - Option 3: Remap shared memory into the address space of both the untrusted and trusted domains
- Shared writable memory can use Option 2 or 3.

Domain Crossing



- trusted stubs to handle RPC
  - for each pair of fault domains
  - stub: copy arguments, re/store registers, switch the exe. stack, validate dedicated regs but! no traps or address space switching (thus, cheaper than HW RPC)
- jump tables to transfer control
  - consists of jump instructions of which target address is legal, outside the domain

SFI details

- Need compiler assistance
  - to dedicate registers, and to insert SFI checks
  - But we cannot trust the compiler used to produce an untrusted binary!

SFI details

- Need compiler assistance
  - to dedicate registers, and to insert SFI checks
  - But we cannot trust the compiler used to produce an untrusted binary!
- Need a verifier that operates on binary code.
  - Verification is quite simple
    - Target registers to be loaded strictly after address-sandboxing
    - All direct memory accesses and direct jumps should stay within untrusted domain.

SFI details

- Need compiler assistance
  - to dedicate registers, and to insert SFI checks
  - But we cannot trust the compiler used to produce an untrusted binary!
- Need a verifier that operates on binary code.
  - Verification is quite simple
    - Target registers to be loaded strictly after address-sandboxing
    - All direct memory accesses and direct jumps should stay within untrusted domain.
- Note that SFI checks all indirect accesses and control-transfers at runtime
  - Verifier is verifying the presence of bounds checks
  - It's not guaranteeing that the program will never generate an invalid target address
    - Generally, an undecidable problem.

SFI details (continued)

- Precursor to proof-carrying code [Necula et al]
  - Code producer provides the proof, consumer needs to check it.
    - Proof-checking is much easier than proof generation
    - Especially in an automated verification setting:
    - producer needs to navigate a humongous search space to construct a proof tree
    - consumer needs to just verify that the particular tree provided is valid

Difficulties of bringing SFI to CISC

- Problem 1: Variable-length instructions
  - What happens if code jumps to the middle of an instruction?

Difficulties of bringing SFI to CISC

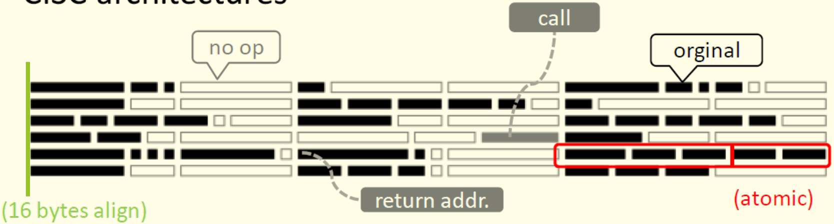
- Problem 1: Variable-length instructions
  - What happens if code jumps to the middle of an instruction?
- Problem 2: Insufficient registers
  - SFI requires several dedicated registers, but x86 has very few.
    - eax, ebx, ecx, edx, esi, edi

Difficulties of bringing SFI to CISC

- Problem 1: Variable-length instructions
  - What happens if code jumps to the middle of an instruction?
- Problem 2: Insufficient registers
  - SFI requires several dedicated registers, but x86 has very few.
    - eax, ebx, ecx, edx, esi, edi
  - PittsSFIeld: uses ebx as a dedicated register AND treats esp and ebp as sandboxed registers (adds needed checks)

Solution to Problem 1

CISC architectures



- padding with no-ops to enforce alignment constraints (power of two)
  - because CISC architectures allow various instruction streams, which makes SFI harder
- `call` placed at the end of chunks
  - because the next addresses are targets of returns
  - they also have <u>low 4 bits zero</u> due to 16 bytes align
- put unsafe operation and its corresponding check together in a chunk
  - atomic, i.e. unsafe op. must be followed by check; no dedicated registers required

Solution to Problem 2

- Hardcode valid segments
  - Avoids need for segment registers etc.

Solution to Problem 2

- Hardcode valid segments
  - Avoids need for segment registers etc.
- Make code and data segments adjacent, with addresses differing in just a single bit
  - Data access restriction needs just a single instruction!
    - and 0x20ffffff, %ebx
    - Read or store using ebx

Solution to Problem 2

- Hardcode valid segments
  - Avoids need for segment registers etc.
- Make code and data segments adjacent, with addresses differing in just a single bit
  - Data access restriction needs just a single instruction!
    - and 0x20ffffff, %ebx
    - Read or store using ebx
  - For indirect jumps, use:
    - and 0x10fffff0, %ebx
    - Jump using ebx

Solution to Problem 2

- Hardcode valid segments
  - Avoids need for segment registers etc.
- Make code and data segments adjacent, with addresses differing in just a single bit
  - Data access restriction needs just a single instruction!
    - and 0x20ffffff, %ebx
    - Read or store using ebx
  - For indirect jumps, use:
    - and 0x10fffff0, %ebx
    - Jump using ebx
- Alternative approach
  - Use x86 segment (CS, DS, ES) registers
    - Very efficient but not available on x86_64

Control-flow Integrity (CFI) [Abadi et al]

- Motivation: Unrestricted control-flow transfers (CFTs) can subvert the IRM
  - Simply jump past checks, or
  - Jump into IRM code that updates critical IRM data

Control-flow Integrity (CFI) [Abadi et al]

- Motivation: Unrestricted control-flow transfers (CFTs) can subvert the IRM
  - Simply jump past checks, or
  - Jump into IRM code that updates critical IRM data
- Step 1: Compile time: Compute possible indirect branch targets using static analysis
  - Option A: Coarse-grained analysis, e.g., list all valid function starts and return targets
  - Option B: Fine-grained static analysis (for a tighter policy)
    - for each function pointer, compute a safe superset of all possible values
    - restrict returns to valid direct and indirect call sites of a function.

Control-flow Integrity (CFI) [Abadi et al]

- Motivation: Unrestricted control-flow transfers (CFTs) can subvert the IRM
  - Simply jump past checks, or
  - Jump into IRM code that updates critical IRM data
- Step 1: Compile time: Compute possible indirect branch targets using static analysis
  - Option A: Coarse-grained analysis, e.g., list all valid function starts and return targets
  - Option B: Fine-grained static analysis (for a tighter policy)
    - for each function pointer, compute a safe superset of all possible values
    - restrict returns to valid direct and indirect call sites of a function.
- Step 2: Runtime: check actual targets against the permissible ones
  - Note: No need to check direct calls, just indirect calls and (all) returns

CFI: Forward Edge Vs Backward Edge

- Forward edge: Enforce policies on targets of indirect calls
- Backward edge: Enforce policies on returns
  - Coarse-grained is insufficient to mitigate control-flow hijack
  - ROP restricted to gadgets beginning at valid return targets ("call-site gadgets") is still too powerful
  - Shadow stack can enforce the ultimate fine-grained backward edge policy
    - But there may be some corner cases in terms of compatibility
    - Recent intel processors include HW support for shadow stacks
- For protecting the IRM, coarse-grained CFI is enough
- For control-flow mitigation, even fine-grained CFI is relatively weak

Coarse-Grained CFI

- Takes into account the type of control transfer, but not much additional info available at the control transfer source ("context insensitive")
- Here is a typical policy
  - All calls should go to beginning of functions
  - All returns should go to instructions following calls
  - No control flow transfers can target instructions belonging to IRM

Coarse-Grained CFI

- Main benefits
  - Simple
    - no need for any nontrivial static analysis
    - efficient implementation using compact read-only tables
  - Does not pose compatibility problems
  - Sufficient for protecting IRMs

Fine-Grained CFI

- Context sensitive: Uses one or more of the following types of info from the control transfer site
    - Location of the source instruction
    - Types of arguments to indirect function calls
    - Possible data flows into the variable holding the code pointer or the arguments
- Benefits
    - Increased security by reducing the # of possible targets

Fine-Grained CFI

- Drawbacks
  - Increased complexity (static analysis and enforcement)
  - Poses compatibility challenges with separate compilation and dynamic loading
- Status
  - Type-based fine-grained CFI available in LLVM/GCC (but not default)
  - Particularly important for C++ because of widespread use of function pointers (virtual functions)

CFI for Securing the IRM

- Coarse-grained version is sufficient to protect IRM
  - Like SFI, CFI is self-protecting
    - CFI checks the targets of jump, so it can prevent unsafe CFTs that attempt to jump just beyond CFI checks
    - In PittSFIeld, this was achieved by ensuring that the check and access operations were within the same bundle
    - Jumps can only go to the beginning of a bundle, so you can't jump between check and use
  - Because of this, SFI and CFI provide a foundation for securing untrusted code using inline checks.

CFI for Securing the IRM

- CFI can also be applied to protect against control-flow hijack attacks
  - Jump to injected code (easy)
  - Return to libc (most obvious cases are easy)
  - Return-oriented programming (requires considerable effort to devise ROP attacks that defeat CFI)
  - But not a foolproof defense
- In addition:
  - IRM code shouldn't assume that untrusted code will follow ABI conventions on register use
  - IRM code should use a separate stack
    - To prevent return-to-libc style attacks within IRM code

CFI Implementation Strategies

- Approach 1 (proposed in the original CFI paper)
  - Associate a constant index with each CFT target
  - Verify this index before each CFT
    - Ideal for fine-grained approach, where static analysis has computed all potential targets of each indirect CFT instruction

CFI Implementation Strategies

- Issues
  - If locations L1 and L2 can be targets of an indirect CFT, then both locations should be given the same index
  - If another CFT can go to either L2 or L3, then all three must have same index
  - A particular problem when you consider returns
  - Accuracy can be improved by using a stack, but then you run into the same compatibility issues as stack smashing defenses that store a second copy of return address

CFI Instrumentation

- Method (a): unsafe, since ID is embedded in callsite (could be used by attacker)
- Method (b): safe, but pollute the data cache

Source: Opcode bytes	Source: Instructions			Destination: Opcode bytes	Destination: Instructions	
FF E1	jmp	ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst
				...		

can be instrumented as (a):

```
81 39 78 56 34 12 cmp  [ecx], 12345678h  ; comp ID & dst     78 56 34 12   ; data 12345678h   ; ID
75 13             jne  error_label        ; if != fail        8B 44 24 04   mov   eax, [esp+4] ; dst
8D 49 04          lea  ecx, [ecx+4]       ; skip ID at dst    ...
FF E1             jmp  ecx                ; jump to dst
```

or, alternatively, instrumented as (b):

```
B8 77 56 34 12    mov  eax, 12345677h     ; load ID-1         3E 0F 18 05   prefetchnta        ; label
40                inc  eax                ; add 1 for ID      78 56 34 12     [12345678h]      ;   ID
39 41 04          cmp  [ecx+4], eax       ; compare w/dst     8B 44 24 04   mov   eax, [esp+4] ; dst
75 13             jne  error_label        ; if != fail        ...
FF E1             jmp  ecx                ; jump to label
```

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

Approach 1: Assumptions

- UNQ: Unique IDs.
  - choose longer ID to prevent ensure the uniqueness
  - Otherwise: jump in the middle of a instruction or arbitrary place (in data or code)
- NWC: Non-Writable Code.
  - Code could not be modified. Otherwise, verifier is meaningless, thus all the work is meaningless …
- NXD: Non-Executable Data
  - Otherwise, attacker can execute data that begins with a correct ID.
- All the assumptions should hold. Otherwise, this CFI implementation can be defeated.

Approach 1: Implementation

- Although the enforcement technique can support some fine-grained policies, the implementation only attempts coarse-grained enforcement
  - Indirect calls can only target functions whose addresses are taken in the program
  - Returns can only target instructions following calls

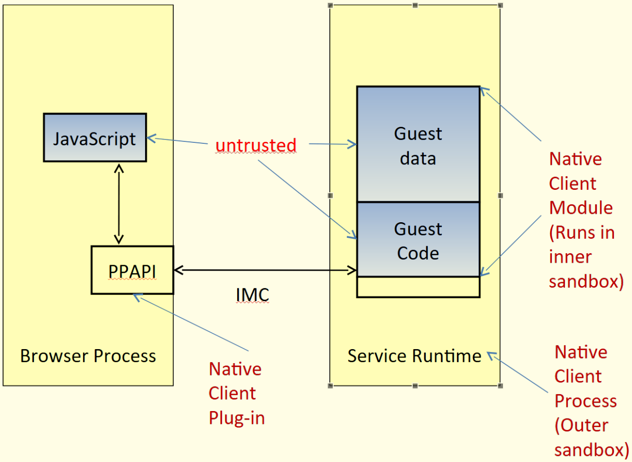
CFI Implementation: Simplified Approach Using Tables

- Use an array V indexed by address, and holding the following values
  - Function_begin, Valid_return, Invalid
- A call to target X is permitted if V[X] == Function_begin
- A return to target X is permitted if V[X] == Valid_return
- Otherwise, CFT is not permitted
  - Note that CFI implementations need only check indirect CFTs
- Store V in read-only memory to protect it

Case Study I: Google Native Client (NaCl) (Motivation)

- Browsers already allow Javascript code from arbitrary sites, but its performance is inadequate for some applications
  - Games
  - Fluid dynamics (physics simulation)
- Permitting native code from arbitrary sites is too dangerous
- NaCl is an environment + toolchain that uses SFI/CFI to enables safe native code execution

System Architecture



Native Client Approach

- Dual sandbox for safe native code execution
  - SFI for inner sandbox that runs downloaded native code
    - Basically, PittSFIeld with two important differences
    - 32-byte bundles rather than 16 byte bundles
    - x86_32 segmentation feature for limiting data access (instead of inserting checking instructions)
    - Native code must be generated by NaCl compiler toolchain
    - safety properties verified at client site at load-time
  - Code in the inner sandbox can call permitted functions in the (trusted) service runtime, e.g., display/rendering functions
    - Service runtime is not subject to SFI

Native Client Approach

- For added security
  - Both these run within a separate process disjoint from the browser
  - This process is sandboxed using seccomp (“outer sandbox”)

Safety Properties Checked At Load-time

- All instructions are reachable by fall-through disassembly from starting address
- All direct transfers to valid instruction boundaries
- All indirect control transfer use nacljmp (pseudo) instruction
- No instructions overlap 32-byte boundary
- No self modifying code
- code is not writable (and cannot be made writable at runtime)
- Statically linked with a fix start address of text segment
- to simplify and speedup sandboxing checks
- The binary is padded up to the nearest page with hlt

Case Study II: WebAssembly (Wasm)(Motivation and Status)

- Same use case as NaCl
  - Support safe downloaded native code in browsers
  - Work seamlessly with the same origin policy
- “Virtualizes” untrusted code within a single process, enabling safe execution alongside trusted code
  - In all major browsers
  - Cloud deployments, e.g., within Cloudflare CDN
- Allows (more or less) arbitrary C/C++ code to be downloaded and run safely
  - Relies on LLVM compiler that translates to Wasm
  - If you are curious, you should check this out
    - Install and try out the emscripten package

Wasm Approach

- Unlike NaCl's use of intel instructions, WebAssembly uses an abstract instruction set (wasm)
- Wasm designed with safety in mind
  - CFI
    - Structured control flow (i.e., no need to check direct transfers)
    - Indirect calls use type-based forward edge checks
    - Use only four basic types for arguments
    - Returns use safe stack protected from memory errors
  - SFI is based on a simple version of memory safety
    - Variables whose addresses are not taken are referenced by indices and stored in safe index spaces
    - Variables whose addresses are taken are stored in a linear memory section. Accesses are bounds-checked to prevent overflow to other regions

Wasm Approach

- Wasm translated into native code before run
  - Wasm compiler (but not C/C++ compiler) is responsible for secure translation

Limitations of NaCl/Wasm Approach

- Need for compiler support
  - Does not work on arbitrary binaries — binaries should have been compiled using a cooperative compiler
  - Otherwise, the binary will trivially fail the verification step
- Question: Can we instrument arbitrary COTS binaries to insert inline security checks?

Motivation for COTS Binary Instrumentation

- No source code needed
  - Language-neutral (C, C++ or other)
- Can be largely independent of OS
- Ideally, would provide instruction-set independent abstractions
  - This ideal is far from today's reality
- Benefits
  - Application extension
    - Functionality
    - Security
    - Monitoring and debugging

Challenges: Disconnect between source and binary code

```c
#include <stdio.h>
void f(int c) {
  printf("%d\n", c);
}
void h(int i) {
  f(i+1);
}
int i(int j) {
  return j+1;
}
int main(int argc, char*argv[]) {
  h(i(argc));
  f(argc+2);
}
```

Compiled Code Example

```
void f(int c) {
  printf("%d\n",
 c);
}
```

```
pushl    %ebp
movl     %esp, %ebp
subl     $16, %esp

pushl    8(%ebp)
pushl    $.LC0 ("%d")
call     printf

leave
ret
```

Function prologue

Function epilogue

Optimized Code Example

```
void h(int i) {   h:
  f(i+1);             pushl    %ebp
}                     movl     %esp, %ebp
                      subl     $8, %esp
```

No push of arguments

```
                      incl     8(%ebp)
```

No epilogue,
not even a call!

```
                      leave
                      jmp      f
```

Optimized Code Example

```
main(int argc,
  char*argv[]) {
   h(i(argc));
   f(argc+2);
}
```

```
main:
   pushl    %ebp
   movl     %esp, %ebp
   pushl    %ebx
   subl     $16, %esp
   movl     8(%ebp), %ebx
   pushl    %ebx
   call     i
   movl     %eax, (%esp)
   call     h
   addl     $2, %ebx
   movl     %ebx, 8(%ebp)
   addl     $16, %esp
   movl     -4(%ebp), %ebx
   leave
   jmp      f
```

Return value in eax reg,
No argument push!

No push of arguments
to f, tail call

Binary Instrumentation Approaches

- Static analysis/transformation
  - Binaries files are analyzed/transformed
  - Binaries files are analyzed/transformed
    - No runtime performance impact
    - No need for runtime infrastructure
- Weakness
  - Error-prone, problem with signed code (can work around)
- Dynamic analysis/transformation
  - Code analyzed/transformed at runtime
  - Benefit: more robust/accurate
  - Weakness
    - High runtime overhead
    - Runtime complexity (infrastructure)

Phases in Static Analysis of Binaries

- Disassembly
- Instruction decoding/understanding
- Insertion of new code

Disassembly

- Required first step for any binary analysis or instrumentation
- Principal Approaches
  - Linear Sweep
  - Recursive Traversal

Linear Sweep Algorithm

- Used by GNU objdump
- Problem
  - There can be data embedded within code
    - There may also be padding, alignment bytes or junk
    - Linear sweep will incorrectly disassemble such data

```
Addr = startAddr;
while (Addr < endAddr ) {
    ins=decode(addr);

    addr+=LengthOf(ins);
}
```

Linear Sweep Algorithm

```
804964c:  55           push  %ebp
804964d:  89 e5        mov   %esp,%ebp
804964f:  53           push  %ebx
8049650:  83 ec 04     sub   $0x4,%esp
8049653:  eb 04        jmp   0x8049658
8049655:  e6 02 04     <junk>
8049658:  be 05000000  mov   $0x5,%esi
```

```
804964c:  55           push  %ebp
804964d:  89 e5        mov   %esp,%ebp
804964f:  53           push  %ebx
8049650:  83 ec 04     sub   $0x4,%esp
8049653:  eb 04        jmp   0x8049658
8049655:  e6 02        out 0x2, al
8049657:  04 be        add al, 0xbe
8049659:  05 00000012  add eax, 0x12000000
```

- Incorrectly disassembles junk (or padding) bytes
- Confusion typically cascades past the padding, causing subsequent instructions to be missed or misinterpreted.

Self Repairing Disassembly

- Property of a disassembler where it re-synchronizes with the actual instruction stream
- Makes detecting disassembly errors difficult
  - 216 of 256 opcodes are valid
- Observation: re-synchronization happens quickly, within 2-3 instructions beyond point of error.

Self Repairing Disassembly (example)

Consider the byte stream
55 89 e5 eb 03 90 90 83 0c 03 b8 01 00 00 00 c9

- **Linear Sweep output**

  100: push ebp

  101: mov ebp, esp

  103: jmp 109

  105: nop

  106: nop

  107: or dword ptr ds:[ebx+eax*1], 0xb8

  111:add dword ptr ds:[eax+eax*1], eax

  113: add byte ptr ds:[eax+eax*1], al

  116: leave

- **Correct Output**

  100: push ebp

  101: mov ebp, esp

  103: jmp 109

  106: <GAP>

  107: <GAP>

  108: <GAP>

  109: or al, 0x3

  111: mov eax, 0x1

  116: leave

Recursive Traversal

- Approach: Takes into account the control flow behavior of the program
- Weakness: For indirect jumps, jump target cannot be determined statically, so no recursive traversal of the target can be initiated
- Some error cases not handled, e.g., jump to the middle of an instruction

```
RecursiveTraversal (addr) {
  while (!visited[addr]) {
     visited[addr] = true;
     ins = decode (addr);
     if (isControlTransfer(ins))
       RecursiveTraversal (target(ins))

     if (uncondJumpOrRet(ins))
       return
     else addr+=LengthOf(ins);
  }
}
```

Obfuscation against Static Disassembly

- Conditional jumps where the condition is always true (or false)
  - Use an opaque predicate to hide this
- Use an opaque predicate to hide this
  - Execution continues in exception handler
- Embedding data in the midst of code
  - With indirect jumps that make it impossible to distinguish between code and data

Code Transformation Challenges

- Code transformations change its size
  - Consider, for example, the addition of CFI or SFI checks
- This means code locations are changed
  - Control-flow targets will all be wrong, and need to be “fixed up”
- As usual, direct transfers are easier to handle
  - Their locations can be determined at transformation time
    - It is nontrivial effort in a binary instrumentation tool to fix them up, but doable

Code Transformation: Main Challenge

- Key Problem: Indirect control transfers
  - Code pointers look like (integer) data values
    - For instance, “fptr = &f” will look like “mov eax, 0x080010b8”
- Finding the new location corresponding to 0x080010b8 isn’t hard
  - No different from the handling of direct control transfer targets
- But what do we with the constant itself?
  - If 0x080010b8 is a reference to code address, then it should be replaced with the new location of the code residing at this address
  - If it is data, it should be left alone
  - How do we decide?
    - If we make a mistake, the program won’t work correctly

Code Transformation: Main Challenge

- And what if it is both?
  - Used as code address in some contexts, data in other contexts
    - Examples:

      code that examines itself

      hash table of code pointers
- Note: Some of these code addresses may be stored in read-only data
  - const void (fptr)(int) = &f;

Dynamic Binary Translation

Just-in-time Disassembly

- Key question: If disassembly is hard because we don't know what is code, why not wait until runtime?
- Key point: Code knows itself
  - Valid code will only jump to valid locations
  - So, delay disassembly of a code snippets until program jumps to them
    - Code is transformed one basic block at a time

      Note: It is trivial to reliably disassemble a single basic block, which is straight-line code with no control-transfers in the middle
  - Even obfuscated code can be handled
    - No way to "hide" code: it will be found before execution

Just-in-time Pointer Fixups

- Just as we rely on code to reveal itself, can we wait for code pointers to reveal themselves??
- Yes
  - If a register is used as a jump target, then the content of the register is a code pointer
    - Fixup code pointers just before they are used
    - Called (runtime or dynamic) address translation
  - Otherwise it is a data pointer or integer
    - Left alone — this means that we don't change any of the constants in the original code or data

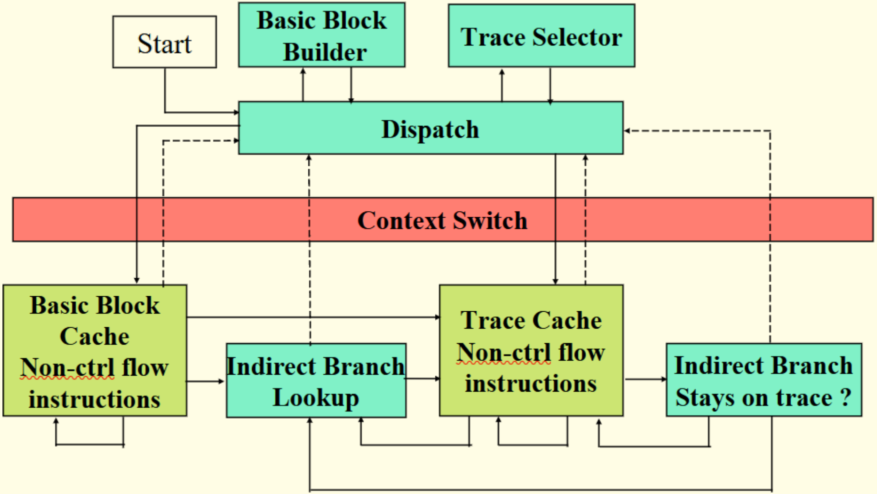
Dynamic Translation: Can it Work?

- May seem like a road to nowhere
  - Likely to be incredibly slow?
- Not necessarily:
  - Just as SFI can be competitive with hardware memory protection, DynamoRIO showed that runtime disassembly and instrumentation can be practical

DynamoRIO

- All code is discovered at runtime, analyzed at runtime, and then rewritten ("translated") at runtime
  - Code is transformed one basic block at a time
- Only the first execution of a basic block requires analysis and rewriting.
  - Subsequent executions can use the same rewritten block.
- Control transfers occur in the last instruction of a basic block.
  - These instructions need to instrumented
    - perform address translation
    - if the target code is not already instrumented, disassemble and instrument it
- Non-control-transfer instructions are executed natively

RIO System Infrastructure



DynamoRIO Operation

- Instrumented programs run in two contexts
  - DynamoRio context (above the redline, representing DynamoRIO runtime). Responsible for detecting the execution of new basic blocks (BBs)
    - These BBs are disassembled, analyzed and then transformed: just-in-time disassembly/rewriting, just before first execution
    - DynamoRIO provides an API for instrumentation: one can use this API to implement custom instrumentation, e.g., count number of BBs executed, number of memory accesses, etc.
  - Application context (below the red line, application code executes natively)
    - Non-control-transfer instructions need no special treatment
    - Control-transfers need to be checked

      If they are direct transfers, then we check if the target has already been instrumented (and hence is in the code cache). If so, directly jump there. If not, switch into DynamoRIO context to perform instrumentation.

      Indirect transfers need to go through a translation table

Address Translation

- Implemented using a translation table
  - A hash table jmptab maps the original address of a BB to its new address in the code cache (corresponding to the location of the instrumented version of code)
  - A hash table jmptab maps the original address of a BB to its new address in the code cache (corresponding to the location of the instrumented version of code)
- At runtime, every indirect CFT to a location l is translated into jmptab[l]
  - Each indirect jump requires a hash table lookup, and has a performance cost
  - Fortunately, common cases (e.g., returns and repeated calls to same target) can be optimized
- If the target is not in jmptab, then control transferred to DynamoRIO runtime.

DynamoRIO Context Switch

- Preserve the following conditions
  - All GPRs (8 in x86-32)
  - Eflags
  - Some system state. Eg: error code
    - DynamoRIO uses one slot in TLS (thread local storage) to store error code (errno) of the application.
      DynamoRIO will use some library routines that may modifiy the state as error code, so it is necessary to preserve application's errno.

DynamoRIO Context Switch

```
add   %eax, %ecx
cmp   $4, %eax
jle   $0x40106f
```

bb with conditional jump

- Assumes that the BBs at 0x40106f and the immediately following BBs are not in the code cache. In this case, control has to be transferred to DynamoRIO runtime when execution reaches the end of this BB. Before context switch, all of the application state (in particular, registers) need to be saved.

```
frag7: add  %eax, %ecx
       cmp  $4, %eax
       jle  stub0
       jmp  stub1
stub0: mov  %eax, eax-slot
       mov  &dstub0, %eax
       jmp  context_switch
stub1: mov  %eax, eax-slot
       mov  &dstub1, %eax
       jmp  context_switch
```

bb in code cache

Transparency and OS Issues

- Transparency: application cannot tell that it is running inside DynamoRIO
- Why does DynamoRIO need transparency?
  - Ensures that application behaves exactly the same way as before: it can't even tell the difference.
  - So, it can't evade DynamoRIO, nor can it behave differently.
- Transparency Issues
  - Library transparency
  - Thread transparency
  - Stack transparency
  - Address space transparency
  - Context Translation
  - Performance transparency (not preserved)

Library Transparency

- Issues when both DynamoRIO and application enters the same non-re-entrant library routine
  - System state might be broken (errno)
  - Library routine may fail to work (malloc)
- Solution:
  - Use system call on both windows and Linux
  - Use stateless library routines
  - Implement own memory (de)allocation routines.

Thread Transparency

- DynamoRIO does not create its own thread
- Why?
  - violate transparency when application that monitors all reads in a process
  - Performance issue when threads double
- What about one DynamoRIO thread?
  - Still violate transparency
  - Performance degrades when multiple threads switch into DynamoRIO mode
- Therefore, use app thread with new context

Stack Transparency

- DynamoRIO does not ‘touch” application stack.
  - Some applications may access data beyond the top of stack. Eg: Microsoft office
  - Usual stack conventions may not be followed by hand-crafted assembly
    - use of esp as a GPR
  - Ability to read return address off stack and use in computing code location (or modify it)
    - Used in PIC (position-independent code)
- Solution:
  - Use a private stack for each thread in DynamoRIO mode
  - Do not modify content of original stack

Address Space Transparency

- DynamoRIO should not “leak” information about itself.
  - On Windows, intercept
    - NtQueryVirtualMemory() that traverse memory regions
    - GetModuleFileName() (library call) to check if library is present
  - On Linux, intercept
    - mmap(). etc.
- More measures (security)
  - Mark DynamoRIO code as NX, when in code cache

Context Translation

- When exception occurs, the faulting place should be the original code address.
  - Intercept user signal handler
  - Check the address map, find the original address
  - Modify the signal stack and go to user signal handler

Transparency and OS Issues

- Operating System Issues
  - Kernel Mediated Control Flow
  - System Call Handling
  - Thread synchronization

Kernel Mediated Control Flow

- Signal Handling
  - DynamoRIO routine will get control first
  - Signals will be queued and delayed, except urgent signals
    - Eg: SIGSEGV
- When signal arrives, if the thread is at
  - Code cache:
    - Unlink the current basic block, go back to DynamoRIO
    - If bb contains syscall, jump to exit stub before syscall.

      Why? Bound timing of signal handler, since syscall is expensive.
  - DynamoRIO code:
    - Delay signal until reaching a safe place
    - Emulate kernel behavior

System Call Handling

- OIf syscall number is not statically known or on DynamoRIOs list
  - Insert pre-syscall & post-syscall routines around the instruction
  - Uninterested syscall: left unchanged. However:
  - For signal handling, app must LEAVE code cache QUICKLY (for timing issue)

- Insert a jump prior to the syscall :
  - **Jmp <syscall or bail>**
  - **Bail: jmp <exit stub>**
  - **Syscall:**
  - **<system call insruction>**

Program Shepherding: An IRM based on DynamoRIO

- Introduces in-line checks to defend against common exploits
  - Buffer overflow attacks
  - Format string attacks
  - Injection of malicious code
  - Re-use of existing code (existing code attacks)
- Sandboxing

Program Shepherding Performance under Linux

Program Shepherding Performance under Linux

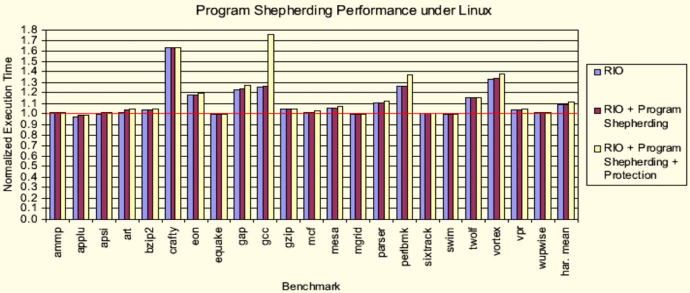


Figure 3: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN 90 benchmarks) on Linux. They were compiled using `gcc -O3`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

- gcc is slow since it consists of many short runs with little code re-use

Program Shepherding Performance under Windows

Program Shepherding Performance under Windows

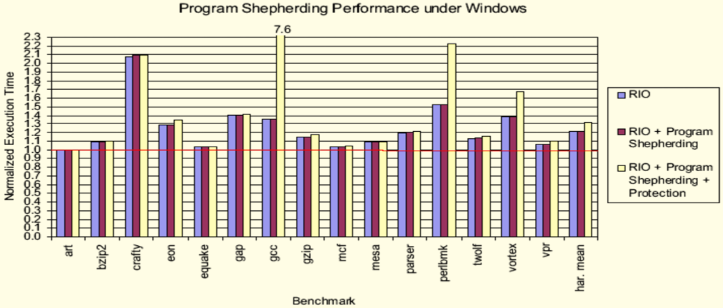


Figure 4: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN benchmarks) on Windows 2000. They were compiled using `cl /Ox`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

- Windows is much less efficient at changing privileges on memory pages than Linux

Caveat about performance

- DBT performance measurements usually based very long-running CPU-intensive benchmarks
- These applications represent the “best case scenario” for DBT systems
  - Rewrite once, execute for a long time
- Real-world performance can be bad
  - 10x to 40x slowdown in the worst case
- Example DBT systems
  - DynamoRIO, Pin, Valgrind, …

Caveat about performance (Continued)

- But its exceptional level of compatibility with arbitrary binary code can still be compelling for
  - CPU-intensive applications with tight loops
  - Coarse-granularity instrumentation (i.e., very small fraction of instructions instrumented)
  - Debugging applications

Other Dynamic Transformation Tools

- Pin
  - better supported now than DynamoRIO
  - better engineered for Linux
- Strata
- Valgrind
  - Most popular open-source tool for finding memory errors and many other applications
- Qemu
  - Can support whole system emulation

DynamoRIO vs Pin

- Architecture dependency
  - Pin tools: written in c/c++
  - DynamoRIO: written in x86 assembly
- DynamoRIO's tools allow users to operate at a lower level
  - Have more control over efficiency, but programming can be hard, and architecture dependent.

BBCount Pin Tool

- For more information, including tutorials and examples, see https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

```
static int bbcount;

VOID PIN_FAST_ANALYSIS_CALL docount() { bbcount++; }

VOID Trace(TRACE trace, VOID *v) {
  for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl);
                                       bbl = BBL_Next(bbl)) {
    BBL_InsertCall(bbl, IPOINT_ANYWHERE, AFUNPTR(docount),
                        IARG_FAST_ANALYSIS_CALL, IARG_END);
  }
}

int main(int argc, char *argv[]) {
    PIN_InitSymbols();
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_StartProgram();
    return 0;
}
```

BBCount DynamoRIO Tool

```
static int global_count;

static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating) {
    instr_t *instr, *first = instrlist_first(bb);
    uint flags;
    /* Our inc can go anywhere, so find a spot where flags are dead. */
    for (instr = first; instr != NULL; instr = instr_get_next(instr)) {
        flags = instr_get_arith_flags(instr);
        /* OP_inc doesn't write CF but not worth distinguishing */
        if (TESTALL(EFLAGS_WRITE_6, flags) && !TESTANY(EFLAGS_READ_6, flag
s))
            break;
    }
    if (instr == NULL)
        dr_save_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    instrlist_meta_preinsert(bb, (instr == NULL) ? first : instr,
        INSTR_CREATE_inc(drcontext, OPND_CREATE_ABSMEM((byte *)&global_coun
t, OPSZ_4)));
    if (instr == NULL)
        dr_restore_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    return DR_EMIT_DEFAULT;
}

DR_EXPORT void dr_init(client_id_t id) {
    dr_register_bb_event(event_basic_block);
}
```

Applicability of Static Vs Dynamic Techniques

- Some techniques require static instrumentation
  - Any technique that uses static analysis to compute a property and then enforces it at runtime
    - CFI, some aspects of bounds-checking, some types of randomizations, ...
- Others can use dynamic instrumentation
  - Stackguard, SFI (but may be limited if CFI can't be assured)
- And yet others that cannot use static instrumentation
  - Obfuscated code, mainly malware