Software Fault Isolation and Control Flow Integrity

Fritz Henglein

Lecture 6, Software Security, DIKU

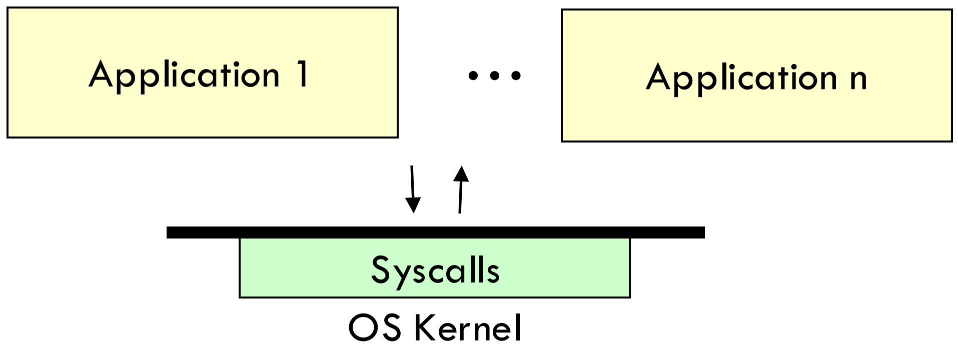
2025-09-23

All slides by: Gang Tan, *Principles and Implementation Techniques of Software-Based Fault Isolation*, Penn State University, Spring 2019

Isolation via Protection Domains

- A fundamental idea in computer security
  - [Lampson 74] “Protection”
- Structure a computer system to have multiple **protection domains**
  - Each domain is given a set of privileges, according to its trustworthiness

Example: the Separation between OS and User Applications



- One OS domain (the kernel mode)
  - Privileged: execute privileged instrs; set up virtual memory; perform access control on resources; …
- Multiple application domains
  - Go through OS syscalls to request access to privileged operations
  - Application domains are isolated by OS processes

Isolating Untrusted Components

- Using separate protection domains is a natural choice for isolating untrusted components
- E.g., isolating plug-ins in a web browser
  - Malfunctioning/malicious plug-ins must not crash or violate the security of the browser
- E.g., isolating device drivers in an OS

Many Forms of Protection Domains

- **Hardware-based virtualization**: Each domain in a virtual machine
  - Pros: easy to use; high degree of isolation
  - Cons: extremely high overhead when context switching between domains
- **OS processes**: each domain in a separate OS process
  - Pros: easy to use; cons: high context-switch overhead
- **Language-based isolation**: rely on safe languages or language features such as types
  - Pros: fine grained, portable, flexible, low overhead
  - Cons: high software engineering effort to use safe languages/features
    - Guaranteed safety and security by construction requires languages with effective support for reasoning about program semantics

Comparison of Forms of Protection Domains

	Context-switch overhead	Per-instruction overhead	Require compiler support	Software engineering effort
Virtual machines	Very high	None	No	None
OS processes	High	None	No	None
Language-based isolation	Low	Medium (dynamic checking) or none (static checking)	Yes	High
SFI	Low	Low	Maybe	None or medium

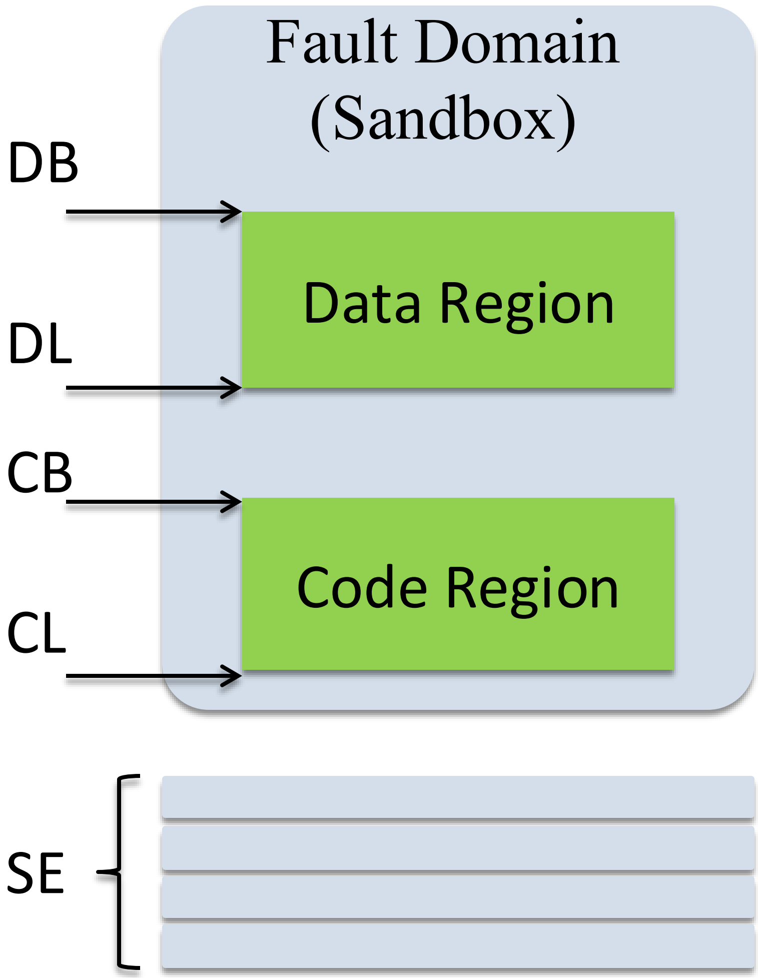
Per-instruction overhead: whether for each instruction additional checking is needed

Software-Based Fault Isolation (SFI)

- Introduced by [Wahbe et al. 93] for MIPS
  - PittSFIeld [McCamant & Morrisett 06] extended it to x86
- SFI isolation is within the same process address space
  - Each protection domain has a designated memory region
  - Same process: avoiding costly context switches
- Implementation by inserting software checks before critical instructions
  - E.g., memory reads/writes, indirect branches.
- Pros: fine grained, flexible, low context-switch overhead
- Cons: may require some compiler support and software engineering effort

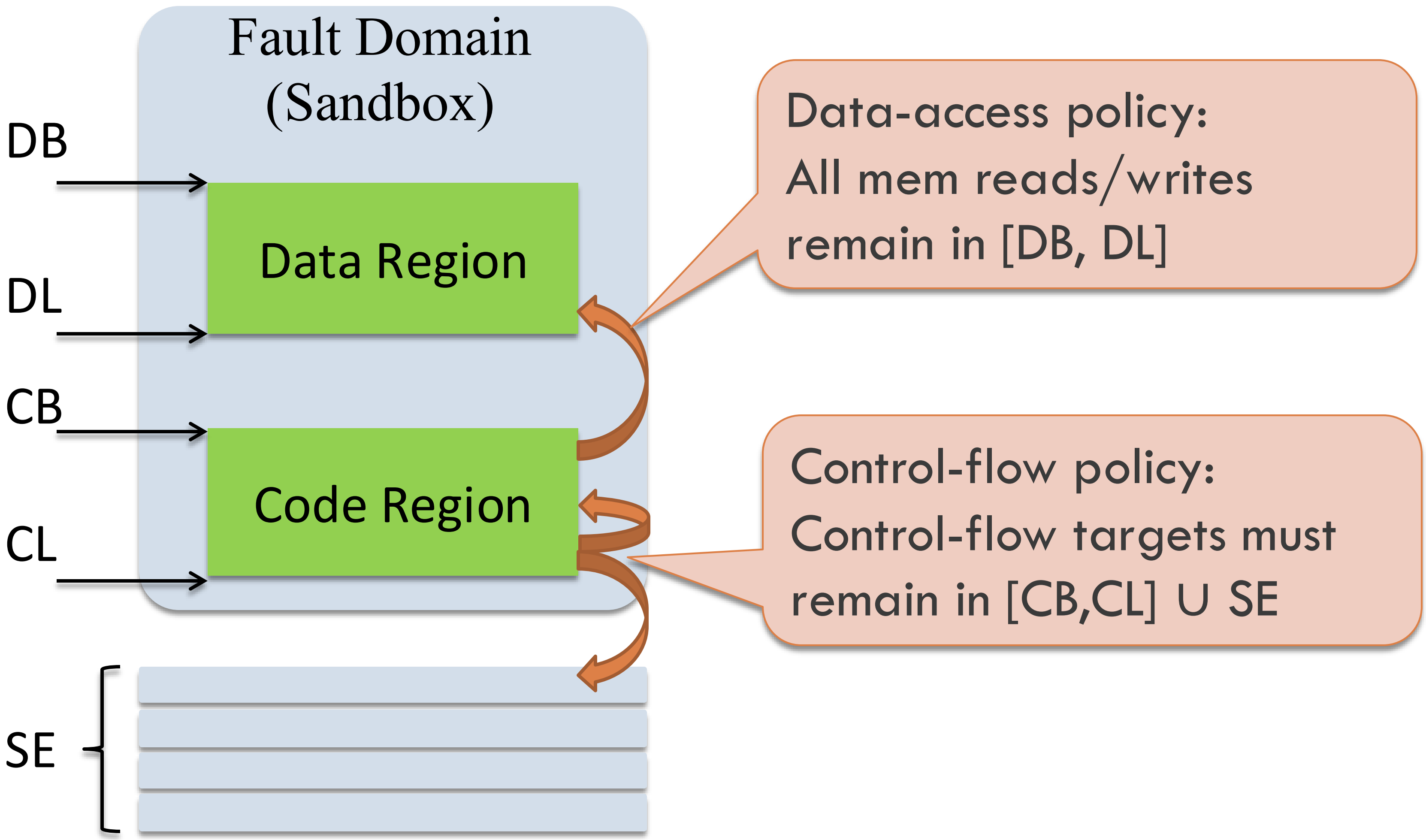
THE SFI POLICY

The SFI Sandbox Setup



- Data region (DR): [DB,DL]
  - Holds data: stack, heap
  - Data Begin, Data Limit
    - Data Limit may be an offset
- Code region (CR): [CB,CL]
  - Holds code
- Safe External (SE) addresses
  - Host trusted services that require higher privileges
  - Code can jump to them for accessing resources
- DR, CR, and SE are disjoint

The SFI Policy



Implications of the SFI Policy

- **Non-writable code**
  - All memory writes must write to DR
  - Code region must not be modified
    - No self-modifying code
- **Non-executable data**
  - Control flow cannot transfer to the data region
  - Cannot inject data to DR and execute it as code
    - Code injection prevented

Stronger Policies

- An SFI implementation might implement a stronger policy
  - For implementation convenience
  - For efficiency
- E.g., PittSFIeld [McCamant & Morrisett 06]
  - Disallow jumping into the middle of instructions on x86, which has variable-sized instructions
- E.g., NaCl [Yee et al. 09]
  - Disallow system call instructions in the code region

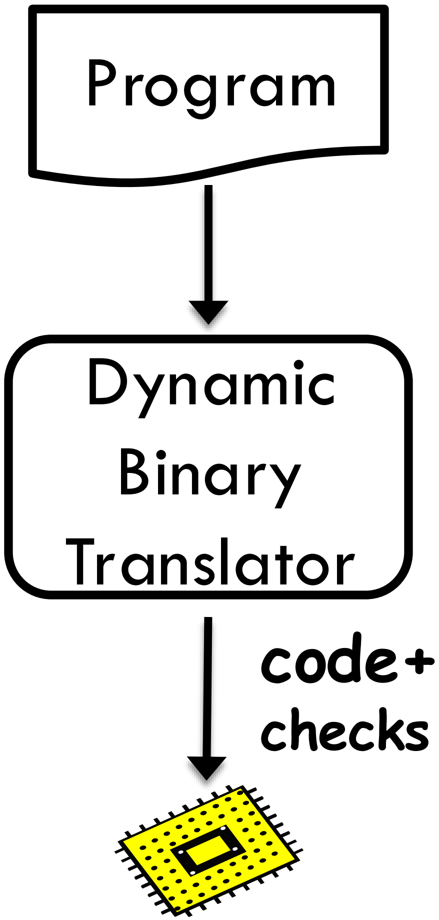
SFI ENFORCEMENT OVERVIEW

SFI Enforcement Overview

- ***Dangerous instructions***: memory reads, memory writes, control-transfer instructions
  - They have the potential of violating the SFI policy
- SFI enforcement
  - Checks every dangerous instruction to ensure it obeys the policy
- Two general enforcement strategies
  - Dynamic binary translation
  - Inlined reference monitors

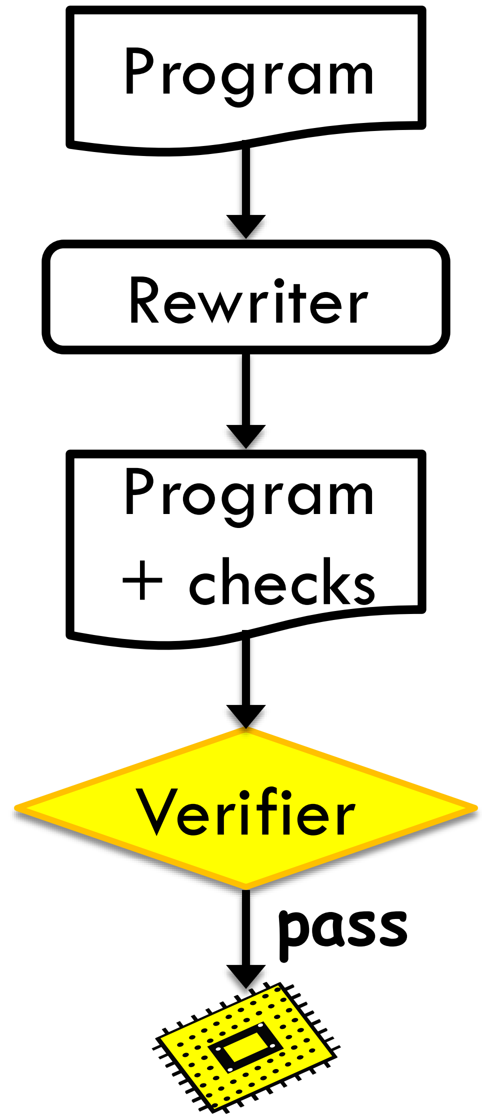
Dynamic Binary Translation

- Efficient interpretation of instructions
- For a dangerous instruction, the interpreter checks it is safe according to the policy
- Examples
  - Program shepherding [Kiriansky et al. 02]
  - libdetox [Payer & Gross 11]
  - VX32 [Ford & Cox, 08]



Inlined Reference Monitors (IRM)

- A static program rewriter
  - Inlines checks into the input program
- More efficient
  - No dynamic translation costs
  - Can optimize checks via static analysis
- More trustworthy
  - A separate verifier can check that checks are inlined correctly
- The main SFI implementation strategy and the focus of the rest slides



Strategies for Implementing IRM Rewriters

- Binary Rewriting
  - Input: binary code
  - Steps: perform disassembly; insert checks; assemble the instrumented code
  - Pros: not requiring source code
  - Cons: hard to disassemble stripped binaries
- Inside a compiler
  - Input: source code
  - Steps: the compiler inlines checks when generating binary code
  - Pros: can perform more optimizations on checks with richer information on code (e.g., types)

ENFORCING SFI'S DATA-ACCESS POLICY AND OPTIMIZATIONS

An Idealized Assembly Language

- We introduce an idealized assembly language
  - For writing assembly-code examples to show SFI enforcement and optimizations

$$
\begin{array}{llll}
(Instr) & i & ::= & r_d := r_s \; aop \; op \\
 & & | & r_d := \mathrm{mem}(r_s + w) \;\; | \;\; \mathrm{mem}(r_d + w) := r_s \\
 & & | & \mathrm{if} \; (r_s \; cop \; op) \; \mathrm{goto} \; w \;\; | \;\; \mathrm{jmp} \; op \\
(Register) & r & ::= & \mathrm{r0} \; | \; \mathrm{r1} \; | \; \mathrm{r2} \; | \; \ldots \\
(Operand) & op & ::= & r \; | \; w \\
(ALOp) & aop & ::= & + \; | \; - \; | \; \gg \; | \; \ll \; | \; \& \; | \; `|\text{'} \; | \; \ldots \\
(CompOp) & cop & ::= & > \; | \; < \; | \; \leq \; | \; \geq \; | \; = \; | \; \neq \; | \; \ldots
\end{array}
$$

* w for a static constant word

Abbreviations and Terminology

- r := r’+0 abbreviated as r := r’
- In memory instructions, mem(r+0) abbreviated as mem(r)
- Direct branches: jmp w
  - The jump target is a static constant word w
- Indirect branches: jmp r
  - The jump target is in a register and cannot always be statically determined

Example


```
  r3 := r1
  r4 := r2 * 4
  r4 := r1 + r4
  r5 := 0
loop:
  if r3 ≥ r4 goto end
  r6 := mem(r3)
  r5 := r5 + r6
  r3 := r3 + 4
  jmp loop
end:
```

- r1 is a pointer to the beginning of an array
- r2 holds the array length
- The program computes in r5 the sum of array elements

Naïve Enforcement

- Insert checks before memory reads/writes

```
mem(r1+12) := r2 // unsafe mem write
```

```
r10 := r1 + 12
if r10 < DB goto error
if r10 > DL goto error
mem(r10) := r2
```

*Assume r10 is a scratch register

Naïve Enforcement

- Sufficient for security
- Has a high runtime overhead
  - Two checks per memory access
- A practical SFI implementation
  - Need to implement a range of optimizations to drive down the cost
    - Discussed next
  - Side note: a good illustration of what's needed to make a simple security scheme practical

Optimization: Integrity-Only Isolation

- A program performs many more reads than writes
  - In SPEC2006, 50% of instructions perform some memory reads or writes; only 10% perform memory writes [Jaleel 2010]
- For integrity, check only memory writes
- Sufficient when confidentiality is not needed
- Much more efficient
  - [Wahbe et al. 1993] on MIPS using typical C benchmarks
    - 22% execution overhead when checking both reads and writes; 4% when checking only writes
  - PittSFIeld on x32 using SPECint2K
    - 21% execution overhead when checking both reads and writes; 13% when checking only writes
- As a result, most SFI systems do not check reads

Optimization: Data Region Specialization

- Special bit patterns for addresses in DR
  - To make address checks more efficient
- One idea in the original SFI [Wahbe et al. 1993]
  - Data region addresses have the same upper bits, which are called the **data region ID**
  - Only one check is needed: check whether an address has the right region ID

Optimization: Data Region Specialization

- Example: DB = 0x12340000 ; DL = 0x1234FFFF
  - The data region ID is 0x1234
  - “mem(r1+12) := r2” becomes

```
r10 := r1 + 12
r11 := r10 ≫ 16 // right shift 16 bits to get the region ID
if r11 ≠ 0x1234 goto error
mem(r10) := r2
```

Q: What does “r3 := mem(r4-20)” become?

Optimization: Address Masking

- **Address checking** stops the program when the check fails
  - Strictly speaking, unnecessary for isolating faults
- A more efficient way: force the address of a memory operation to be a DR address and continue execution
  - Called **address masking**
  - “Ensure, don’t check”
  - When using data region specialization, just modify the upper bits in the address to be the region ID
  - PittSFIeld reported 12% performance gain when using address masking instead of checking for SPECint2000

Optimization: Address Masking

- Example: DB = 0x12340000 ; DL = 0x1234FFFF
  - "mem(r1+12) := r2" becomes

```
r10 := r1 + 12
r10 := r10 & 0x0000FFFF
r10 := r10 | 0x12340000
mem(r10) := r2
```

Force the address to be in DR

Q: What does "r3 := mem(r4-20)" become?

Wait! What about Program Semantics?

- “Good” programs won’t get affected
  - “Good” programs won’t access memory outside DR
  - For bad programs, we don’t care about whether its semantics is destroyed
- Cons: does not pinpoint the policy-violating instruction
  - A downside for debugging and assigning blame

Optimization: One-Instruction Address Masking

- Idea
  - The data region ID has only a single bit on
    - E.g. 0x0001, 0x0002, 0x0004, 0x0008, 0x0010, …, 0x1000, 0x2000, 0x4000, 0x8000 for 16-bit data region IDs
  - Make the zero-ID region, 0x0000 for 16-bit data regions, unmapped in the virtual address space
- A memory access is safe
  - If the address is either in the data region or in the zero-ID region
  - Reason: an access to the zero-ID region generates a hardware trap because it accesses unmapped memory
- Benefit: cut down one instruction for masking
  - PittSFIeld reported 10% performance gain on SPECint2000

Optimization: One-Instruction Address Masking

- Example: DB = 0x20000000 ; DL = 0x2000FFFF
  - Region ID is 0x2000
  - “mem(r1+12):= r2” becomes

```
r10 := r1 + 12
r10 := r10 & 0x2000FFFF
mem(r10) := r2
```

  - Result is an address in DR or in the (unmapped) zero-ID region
- Cons: limit the number of DRs
  - In a 32-bit system, if a DR’s size is 2^n, then we can have at most (32-n) DRs

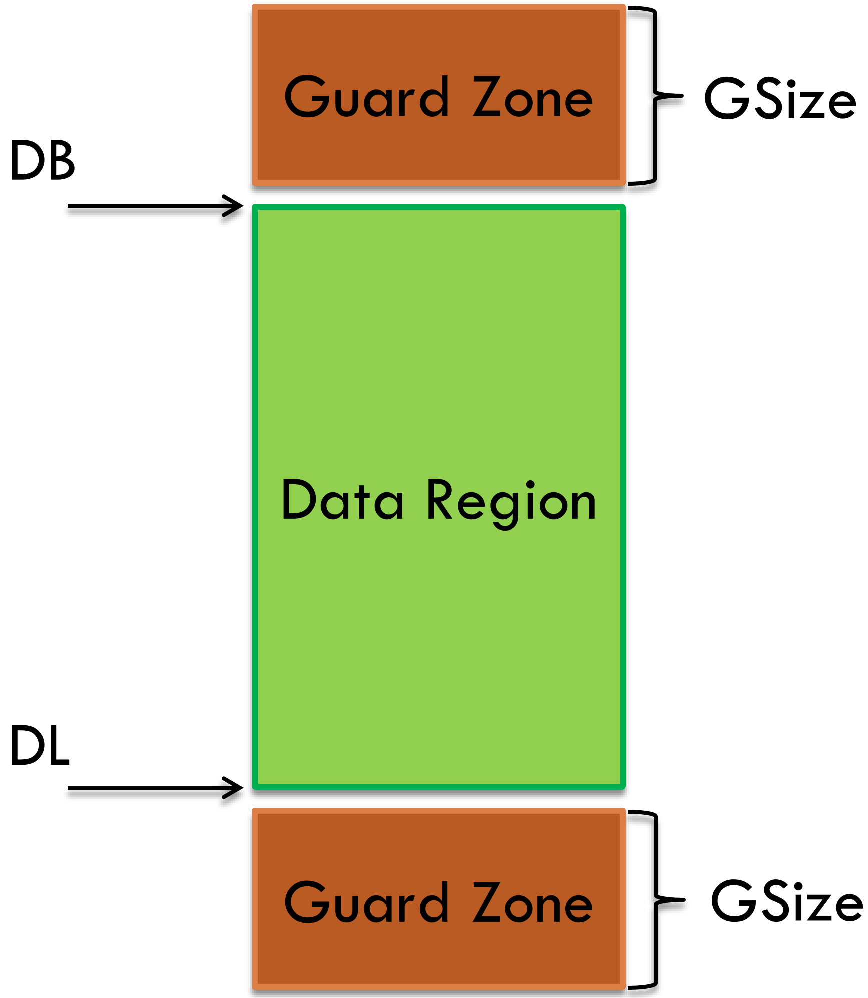
Data Guards

- A **data guard** refers to either address checking or address masking
  - When which one is used is irrelevant
- Introduce a pseudo-instruction "r'=dGuard(r)"
  - To hide implementation details
- An implementation should satisfy the following properties of "r'=dGuard(r)"
  - If r is in DR, then r' should equal r
  - If r is outside DR, then
    - For address checking, an error state is reached
    - For address masking, r' gets an address within the safe range
    - The safe range is implementation specific; it's typically DR; for PittSFIeld, it's DR plus the zero-ID region

Optimization: Guard Zones

- Place a guard zone directly before and after the DR
- First described by Wahbe et al. (1993); further extended by Zeng et al. (2001) and Sehr et al. (2010)

Guard Zones: Safe Accesses



- E.g., GSize=4k
- **Assumption**: Guard zones are unmapped
  - Thus, access to guard zones are trapped by hardware
- A memory read/write is **safe** if the address is in [DB-GSize, DL+GSize]

Guard Zones Enable More Optimizations

- In-place sandboxing
- Redundant check elimination
- Loop check hoisting

Similar to those optimizations performed in an optimizing compiler, enabled by classic static analysis

Optimization: In-Place Sandboxing

- A commonly used addressing mode in memory operations
  - A base register plus/minus a small constant offset
  - E.g., the register points to the start address of a struct, and the constant is the offset to a field
- In this case, just guard the base register in place is sufficient, when the constant is no greater than GSize

Optimization: In-Place Sandboxing

- Example: "mem(r1+12):= r2" becomes

```
r1 := dGuard(r1)
mem(r1+12) := r2
```

  - No need for a scratch register
- Why is the above safe?
  - "r1 := dGuard(r1)" constrains r1 to be in DR and then r1+12 must be in [DB-GSize, DL+GSize], assuming GSize $\geq$ 12
  - Note: for PittSFIeld, we need to have guard zones around the zero-ID region too, since dGuard constrains r1 to be either in DR or the zero-ID region in PittSFIeld
    - Will ignore this for the rest of the slides

Optimization: In-Place Sandboxing

- NaCl-x86-64 (Sehr et al., 2010) implemented a similar optimization
- Put guard zones of 40GB above and below a 4GB sandbox
  - 64-bit machines have a large virtual address space
  - As a result, most addresses in memory operations can be guaranteed to stay in [DB-GSize, DL+GSize]
    - By carefully controlling the registers in "base register + a scaled index register + displacement" addressing mode

Optimization: Redundant Check Elimination

- Idea: perform range analysis to know the range of values of registers and use that to remove redundant data guards

```
r1 := dGuard(r1)
                      <= = = = = = = = = = = = r1 ∈ [DB,DL]
r2 := mem(r1 + 4)
...  // r1 is not changed in between
                      <= = = = = = = = = = = = r1 ∈ [DB,DL]
r1 := dGuard(r1)
r3 := mem(r1 + 8)
```

Removing the redundant guard

Optimization: Loop Check Hoisting

- Idea: a guard in a loop is hoisted outside
  - The guard is performed only once per loop instead of once per loop iteration
- Key observation
  - If addr ∈ [DB-GSize, DL+GSize], then a successful (untrapped) memory operation via addr means
    addr ∈ [DB, DL]
    - If it were in any of the guard zones, then a trap would be generated

Loop Check Hoisting Example

Before optimization

```
  r3 := r1
  r4 := r2 * 4
  r4 := r1 + r4
  r5 := 0
loop:
  if r3 ≥ r4 goto end
  r3 := dGuard(r3)
  r6 := mem(r3)
  r5 := r5 + r6
  r3 := r3 + 4
  jmp loop
end:
```

After optimization

```
  r3 := r1
  r4 := r2 * 4
  r4 := r1 + r4
  r5 := 0
  r3 := dGuard(r3)
loop:
  if r3 ≥ r4 goto end
  r6 := mem(r3)
  r5 := r5 + r6
  r3 := r3 + 4
  jmp loop
end:
```

* r1 is a pointer to the beginning of an array; r2 holds the array length; the program computes in r5 the sum of array elements

Why is the Previous Optimized Code Safe?


```
  r3 := r1
  r4 := r2 * 4
  r4 := r1 + r4
  r5 := 0
  r3 := dGuard(r3)
loop:
  if r3 ≥ r4 goto end
  r6 := mem(r3)
  r5 := r5 + r6
  r3 := r3 + 4
  jmp loop
end:
```

r3 ∈ [DB,DL]

r3 ∈ [DB,DL+4]

r3 ∈ [DB,DL+4]

r3 ∈ [DB,DL

r3 ∈ [DB+4 +4]

[DB, DL+4]
⊆ [DB-GSize, DL+GSize]

Optimization:
Guard Changes Instead of Uses

- Some registers are used often
  - E.g., in 32-bit code, ebp is usually set in the function prologue and used often in the function body
- Idea
  - Sandbox the changes to those special registers, instead of uses
  - E.g., ebp := esp becomes

    ebp := esp

    ebp := dGuard(ebp)

    later uses of %ebp plus a small constant do not need to be guarded, if used together with guard zones

Scratch Registers

- The SFI rewriting may require finding scratch registers to store intermediate results
  - E.g., r10 in many of our previous examples
- If the old values of scratch registers need to be preserved
  - Need to save and restore the old values on the stack
- How to avoid that?

Optimization: Finding Scratch Registers

- Binary rewriting
  - Perform binary-level liveness analysis to find dead registers as scratch registers [Zeng et al. 11]
- Compile-level rewriting
  - Approach 1: reserve dedicated registers as scratch registers
    - E.g., PittSFIeld reserves ebx as the scratch register by passing GCC a special option
    - Downside: increase register pressure
  - Approach 2: rewrite at the level of an IR that has unlimited number of variables
    - E.g., LLVM IR
    - A later register allocation phase maps those variables to registers or stack slots

Architecture-Specific Optimization

- An SFI implementation can use specific hardware features for efficient sandboxing
- NaCl and VX32 on Intel x32
  - Use x32's segmentation support
  - Data segment: base gets DB and limit and DL
  - Hardware automatically performs checks
    - However, not supported in x64
- ISBoxing On x64 [Deng et al. 15]
  - Put the data region in the first 4GB
  - Add address-override prefix to a memory instruction
  - Cons: only support one data region with a fixed size
- ARMlock on ARM [Zhou et al. 14]
  - Use ARM's memory domain feature

ENFORCING SFI’S CONTROL-FLOW POLICY

Control-Flow Policy

- Recall the policy: control-flow targets must stay in [CB,CL] ∪ SE
- However, when using the IRM approach for SFI enforcement
  - Must also restrict the control flow to disallow bypassing of guards

Risk of Indirect Branches


```
l1:     r10 := r1 + 12
l2:     r10 := dGuard(r10)
l3:     mem(r10) := r2
```

- Worry: what if there is a return instruction somewhere else and the attacker corrupts the return address so that the return jumps to l3 directly?
  - Then the attacker bypasses the guard at l2!
  - If attacker can further control the value in r10, then he can write to arbitrary memory location

Risk of Indirect Branches

- In general, any **indirect branch** might cause such a worry
  - If not carefully checked, it may bypass the guard
- Indirect branches include
  - Indirect calls (calls via register or memory operands)
  - Indirect jumps (jumps via register or memory operands)
  - Return instructions
- In contrast, direct branches are easy to deal with
  - Targets of a direct branch encoded in the instruction; can statically inspect the target

The Original SFI Solution [Wahbe et al. 93]

- Make r10 (in MIPS) a dedicated register
  - r10 only used in the monitor code, not used by application code
  - Also maintain the invariant that r10 always contains an address in DR before any branch
  - So even if an indirect branch bypasses the guard before a memory operation, the memory access stays within DR
- Cons?
  - Reduce the number of registers available to application code
  - Allow an indirect branch to target the middle of an instruction; problem for variable-sized instruction sets

A More Direct Approach: Control-Flow Integrity

- Define a **pseudo-instruction**
  - Either a non-dangerous instruction
  - Or a guard followed by a dangerous instruction
- **Strengthened control-flow policy**
  - All control-flow transfers must target the beginning of a pseudo-instruction in CR or an address in SE
- Note the strengthened policy rules out
  - Bypassing a guard
  - And jumping into the middle of an instruction

Aligned-Chunk Enforcement (PittSFIeld)

- Divide the code into chunks of some size
  - E.g., 16 or 32 bytes
- Each chunk starts at an aligned address
  - addr is aligned if addr mod chunkSz = 0
- Make dangerous instrs and their guards stay within one chunk
  - E.g., "r10 := dGuard(r10); mem(r10) := r2" stay within one chunk
- Insert guards before indirect branches so that they target only aligned addresses (chunk beginnings)

Example

- Assume
  - CR is [0x10000000, 0x1000FFFF], the one-bit code region 0x1000
  - Chunk size is 16 bytes
  - Zero-ID region [0x00000000, 0x0000FFFF] unmapped
- Then "jmp r" becomes

```
r := r & 0x1000FFF0
jmp r
```

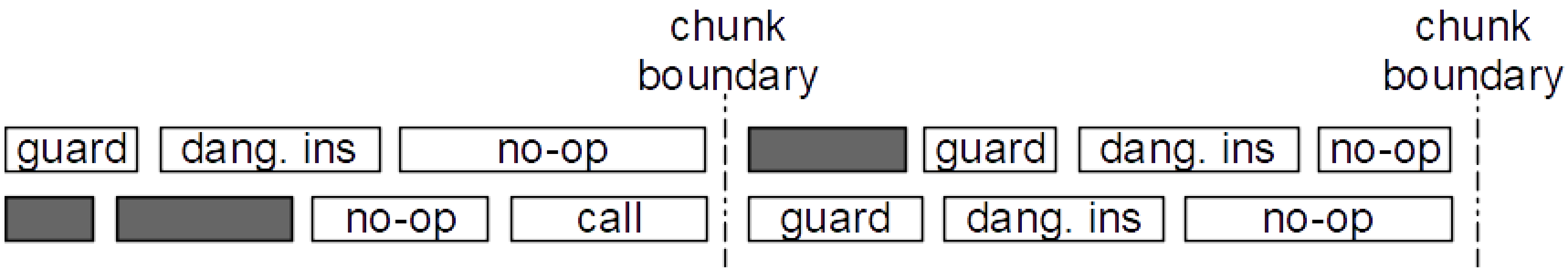
Q: why does the above ensures that the target address is (1) in CR or zero-ID region, and (2) a chunk beginning

- after &, r's upper 16 bits must be either 0x0000 or 0x1000

- after &, r's lower four bits must all be 0, meaning it's 16-byte aligned

Downside of Aligned-Chunk Enforcement

- All legitimate jump targets have to be aligned
  - No-ops have to be inserted for that



- Extra no-ops slow down execution and increase code size
  - In PittSFIeld, inserted no-ops account for half of the runtime overhead; NaCl-JIT incurs 37% slowdown because of no-ops
  - In NaCl-x64, the code size becomes 60% larger

Bitmap Based Enforcement (MIP [Niu & Tan, 13])

- Allow variable-sized chunks
  - A guard and the following dangerous instr still stay within one chunk
  - Chunk beginnings are remembered in an immutable bitmap
    - b[addr]=1 iff addr is the beginning of a chunk
  - Before an indirect branch, insert a guard to check if b[addr] is 1, assuming addr is the target
    - If not, jump to error
- Benefit: no need to insert no-ops
  - MIP-x32: 4% runtime overhead; 13% code increase
  - MIP-x64: 7% runtime overhead; 16% code increase

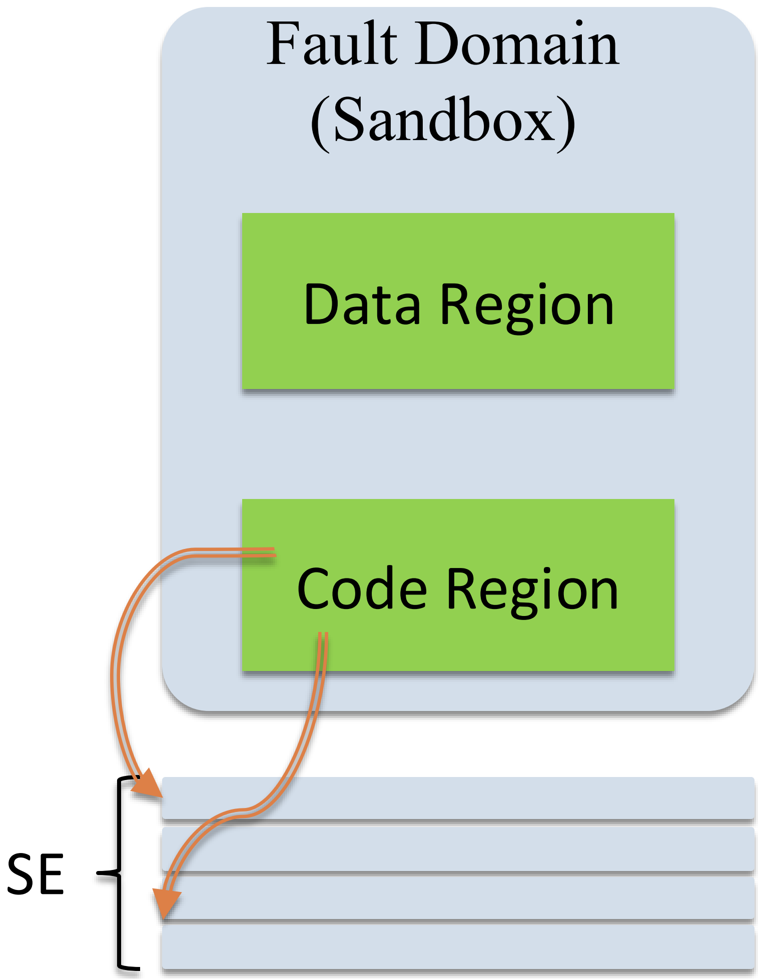
Fine-Grained CFI

- Enforce that a program follows a fine-grained control-flow graph
  - [Abadi et al, 05] and many other follow-up work
  - E.g., for each return, the fine-grained CFG defines a set of possible return targets
- Stronger than the pseudo-instruction based CFI policy
- Pros: we can use the fine-grained CFI to optimize away more guards [Zeng et al., 11]
- Cons: enforcing it incurs additional overhead
  - Unnecessary for the control-flow policy in SFI

Jumping Outside of Fault Domains

- Total isolation is rarely what's desired in practice
- Sandboxed code must interact with other parts of the system for its functionality
  - E.g., a browser plug-in must communicate with the browser's core for exchanging data with the core and other plug-ins

Allow Only Controlled Interaction



- The sandboxed code can jump to a pre-defined set of SE (Safe External) addresses
- Each SE address holds a trusted service
  - E.g., service for invoking OS syscalls (fopen, fread, …)
  - E.g., service for allowing communication with other fault domains

Trusted Services

- Implemented outside of the fault domain
- They can implement additional security policies
  - E.g., can restrict fopen to open files only in a particular directory
  - Or can disallow fopen completely
    - Just do not set up a service entry for fopen

APPLICATIONS OF SFI

SFI Applications Overview

- Isolating OS kernel modules such as device drivers
  - MiSFIT [Small 97]; XFI [Erlingsson et al. 06]; BGI [Castro et al. 09]; LXFI [Mao et al. 11]
- **Isolating plug-ins in Chrome**
  - NaCl [Yee et al. 09]; NaCl-x64 [Sehr et al. 10]
- **Isolating native libraries in the Java Virtual Machine**
  - Robusta [Siefers et al. 10]; Arabica[Sun & Tan 12]

Google's Native Client (NaCl)

- SFI service in Chrome
  - [Yee et al. Oakland 09]
- Goal: download native code and run it safely in the Chrome browser
  - Much safer than ActiveX controls
  - Much better performance than JavaScript, Java, etc.
- Google's main motivation: run native-code games in Chrome

DOOM in NaCl

NaCl: Code Verification

- Code is verified before running
  - Allow restricted subset of x86 instructions
    - No unsafe instructions: memory-dependent jmp and call, privileged instructions, modifications of segment state, …
  - Ensure SFI checks are correctly implemented for the SFI policy

NaCl Sandboxing

- x86-32 sandboxing based on hardware segments
  - Sandboxing reads and writes for free
  - 5% overhead for SPEC2000 benchmarks
- However, hardware segments not available in x86-64 or ARM
  - Use instructions for address masking [Sehr et al. 10]
  - x86-64/ARM: 20% for sandboxing mem writes and computed jumps

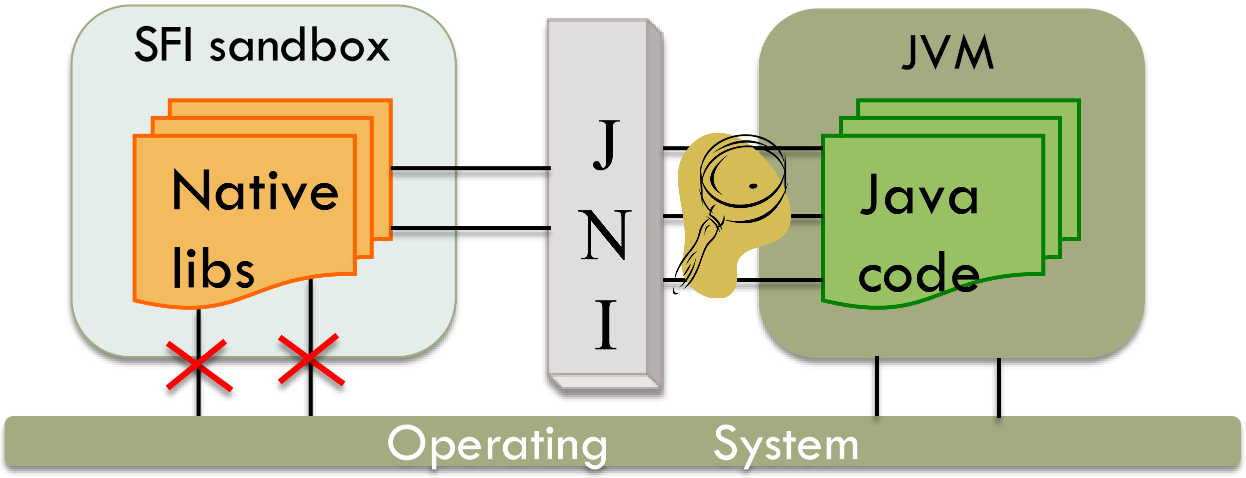
NaCl SDK

- Modified GCC tool-chain
  - Inserts appropriates masks, alignment requirements
- Trampolines allow restricted system-call interface and also interaction with the browser
  - Pepper API: access to the browser, DOM, 3D acceleration, etc.

Robusta [Siefers, Tan, Morrisett 10]

- SFI service in a Java Virtual Machine (JVM)
  - Allow Java code to invoke native code safely through the Java Native Interface (JNI)
- The basic idea
  - Put native code in an SFI sandbox and allows only controlled access to JVM services

Robusta [Siefers, Tan, Morrisett 10]



Native Code Threat

- Direct JVM mem access
- Abusive JNI calls
- OS syscalls

Robusta Remedy

- SFI: Prevent direct JVM access
- Perform JNI safety checking
- Reroute syscall requests to Java's security manager

FUTURE DIRECTIONS

Future Directions

- Tool and programming support for program partitioning
  - How to turn a monolithic application into components in separate protection domains?
    - Privilege separation
  - It took Google significant effort to privilege separate Chrome into a system of cooperating processes [Barth et al. 08]

Future Directions

- Security enforcement on interface code
  - Trusted services in SE addresses are security critical
  - Experience shows that bugs are plenty in such interface code
  - Should apply program analysis/verification for bug finding
  - Or take a specification about interface security and enforce the security a la LXFI [Mao et al. 11]

Future Directions

- Side channel control
  - SFI provides memory isolation but side channels are possible
  - E.g., we might structure a server to have a trusted core and have a sandbox to handle each client connections
    - However, if the core maintains some state that is shared by all connections, there might be a side channel
    - Similar channels were discovered in TCP (“Off-Path TCP Exploits: Global Rate Limit Considered Dangerous”)

Future Directions

- Recovery mechanism
  - Address checking terminates the sandbox when there is an illegal access
    - May still need to release resources
  - Address masking turns an illegal access to a legal one
    - May cause a benign but buggy sandboxed component to misbehave
    - It does not pinpoint the violating instruction
  - [Seltzer et al. 96] Wrap sandbox calls in transactions
    - Transactions are aborted when sandbox misbehavior is detected; resources are released as a result

More in the Survey Article

- G. Tan “Principles and Implementation Techniques of Software-Based Fault Isolation”, Foundations and Trends in Privacy and Security: Vol. 1, No. 3, pp 137–198.
  - http://www.cse.psu.edu/~gxt29/papers/sfi-final.pdf
- SFI verifier
  - Verifies that the result after SFI rewriting is correct
  - Basic idea and formalization
- References