CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science

George Washington University

Slides materials are partially credited to Gang Tan of PSU.

Outline

- Review: Testing and Fuzzing
- Software-based Memory Isolation
- Hardware-based Memory Isolation

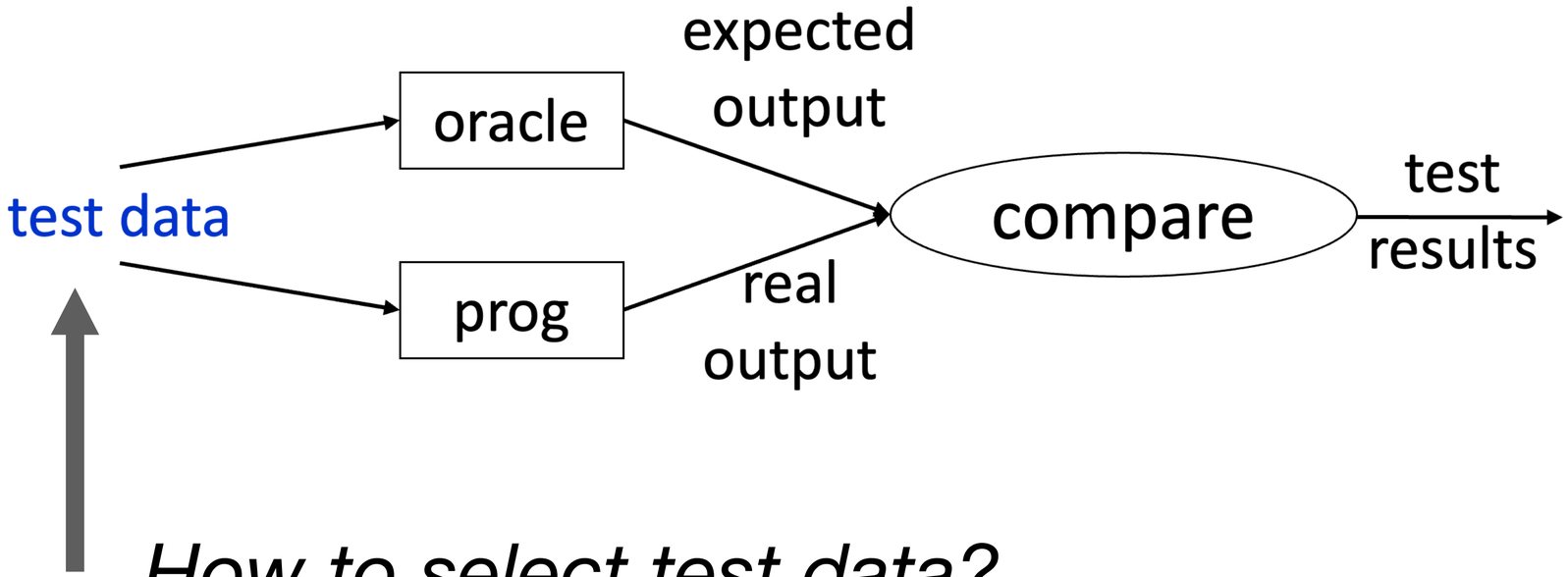
Program Testing

The process of running a program on a set of test cases and comparing the actual results with expected results.

- E.g., for the implementation of a factorial function, test cases could be {0, 1, 5, 10}.
- Testing cannot guarantee program correctness.
  - What's the simplest program that can fool the test cases above?
- However, testing can catch many bugs.

Testing Process

Test oracle: A mechanism/tool that determines the correctness of the tested program under a test case (input).



How to select test data?

Selecting Test Data

- Testing is w.r.t. a finite test set.
  - Exhaustive testing is usually not possible
  - E.g, a function takes 3 integer inputs, each ranging over 1 to 1000
    - Assume each test takes 1 second
    - Exhaustive testing would take 10^9 = 1 billion seconds (~31.7 years!)
- How should we design the test set?
  - Black-box testing
  - White-box (or, glass-box) testing

Black-box Testing

Generating test cases based on specification alone, without considering the implementations (internals).

- Only focusing on the inputs and outputs
- Advantages
  - No need for code knowledge
  - Test cases are not biased toward an implementation.

White-box Testing

Looking into the internals of the program to figure out a set of test cases

Boundary Conditions

- Common programming mistakes: not handling boundary cases
  - Input is zero
  - Input is negative
  - Input is null
  - …
- Test data should cover these boundary cases.

Test Coverage

- Idea: code that has not been covered by tests are likely to contain bugs.
  - Divide a program into a set of elements
    - The definition of elements leads to different kinds of test coverage.
  - Define the coverage of a test suite to be:

$$\frac{\text{\# of elements executed by the test suite}}{\text{\# of elements in total}}$$

Why is Test Coverage Important?

- Test quality is determined by the coverage of the program by the test set so far.
- Benefits:
  - Can be used as a **stopping rule**: e.g., stop testing if 95% of elements have been covered.
  - Can be used as a **metric**: a test set that has a test coverage of 80% is better than one that covers 70%
  - Can be used in a **test case generator**: look for a test which exercises new elements not covered by the tests so far

Possibly Infinite Number of Paths

- Loop may cause infinite # of paths
  - In general, impossible to cover all of them.
- One heuristic
  - Include test data that cover zero, one, and two iterations of a loop
  - Why two iterations?
    - A common programming mistake is failing to reinitialize data in the second iteration.
  - This offers no guarantee, but can catch many errors.

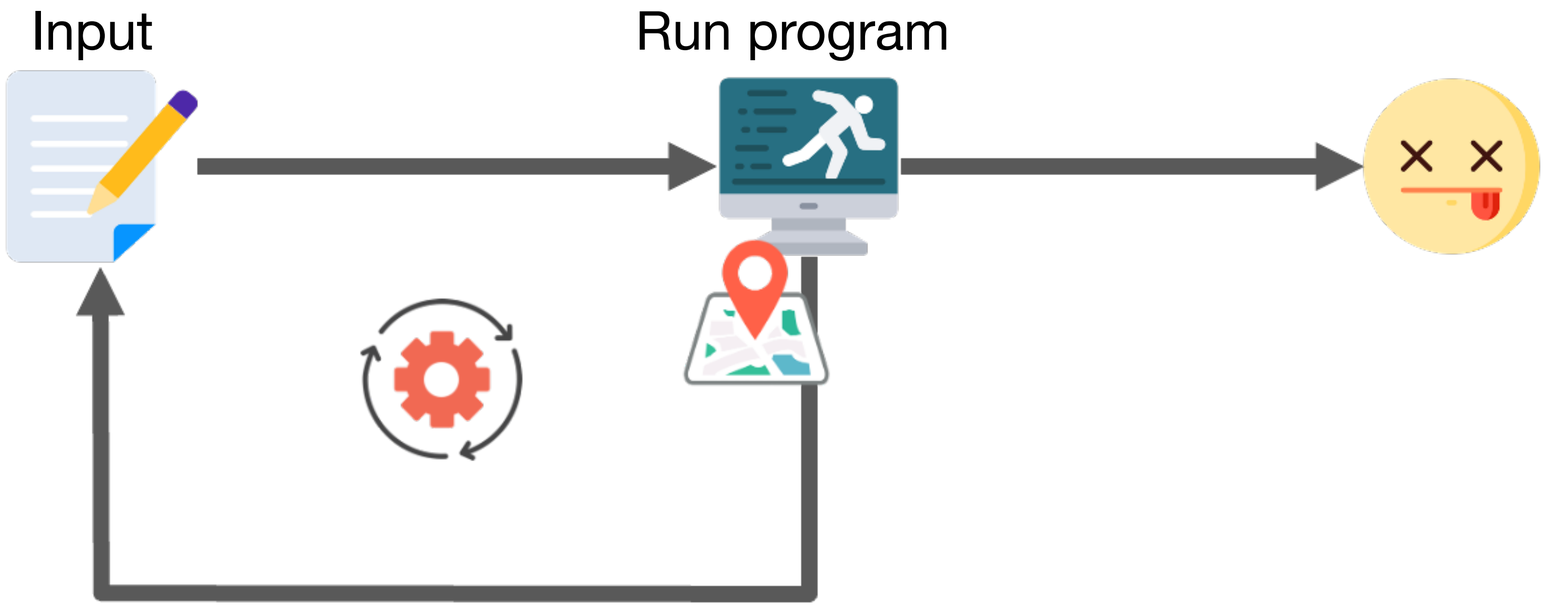
Combine Them All

- A good set of test data combines various testing strategies.
  - Black-box testing
    - Generating test cases by specifications
    - Boundary conditions
  - White-box testing
    - Test coverage (e.g., being edge complete)

Fuzz Testing

Run programs on many random, abnormal inputs and look for bad behaviors in the responses.

- Bad behaviors such as crashes or hangs



Fuzz Testing Overview

- Black-box fuzzing
  - Treating the system as a black box during fuzzing, i.e., not knowing details of the implementation
- White-box fuzzing
  - Designing input generation with full knowledge of the target software
- Grey-box fuzzing
  - Having partial knowledge of the internals of the target

Mutation-based Fuzzing

- User supplies a well-formed input.
- Fuzzing: Generate random changes to that input, i.e., *mutating* the input
- Seed inputs: A set of initial inputs
- Mutations: bit flipping, truncation, duplications, byte changes, etc.
- No assumption about input
  - Only assumes that variants of well-formed input may be problematic for the program
- Example: zzuf
  - https://github.com/samhocevar/zzuf

Generation-based Fuzzing

- Generate inputs from scratch according to predefined rules/specifications
- Generated inputs are well-formed, adhering to the specs
- Can write a generator to generate well-formatted inputs
- Suitable for inputs with a specific format requirement
  - e.g., JSON/XML files, network traffic of certain protocols

Coverage-based Fuzzing

- Rather than treating the program as a black box, instrument the program to track coverage
  - E.g., the coverage of statements/edges/paths
- Uses feedback from the program's execution to guide new input generation
- Also called grey-box fuzzing
- Maintain a pool of high-quality tests
  1. Start with some initial ones (seeds) specified by users
  2. Run tests and record the code coverage
  3. Mutate tests in the pool to generate new tests
  4. Run new tests
  5. If a new test leads to new coverage (e.g., edges), save the new test to the pool; otherwise, discard the new test

AFL

- Mutation-based, coverage-guided, grey-box fuzzer
- The original version is no longer maintained; `afl++` is the newer version.

AFL Mutation Strategies

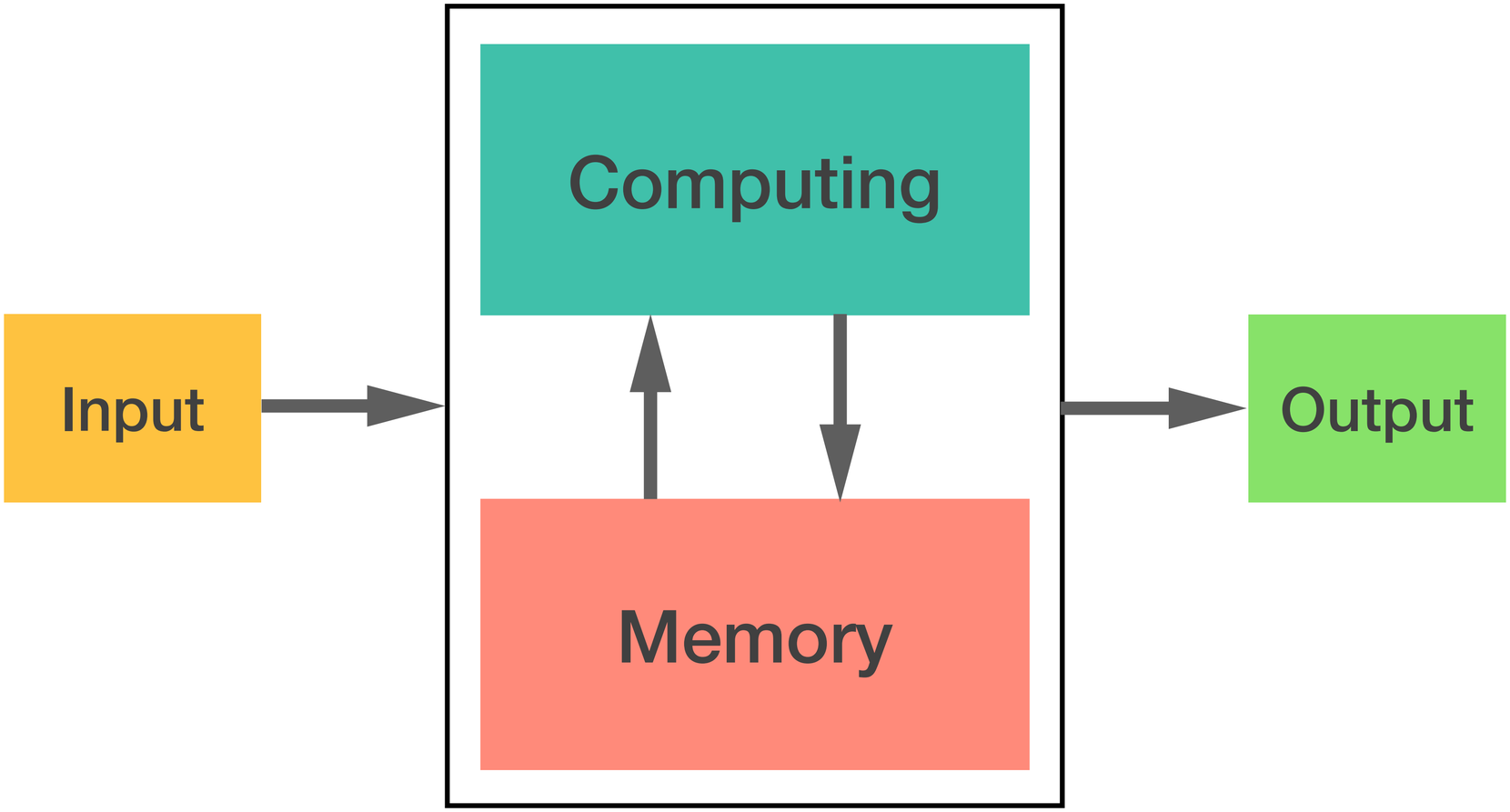
- Highly deterministic at first
  - bit flips
  - adding/subtracting integer values
  - Overwriting parts of the input with “interesting values” (e.g., INT_MAX)
  - Replacing parts of the input with predefined or auto-detected values
- Then, non-deterministic choices
  - insertion/deletion bytes
  - Overwriting with random values
  - Others

Grey-box Fuzzing

- Finds flaws, but still does not understand the program
- Pros: Much more effective than black-box fuzzing
  - Essentially no configurations
  - Lots of crashes have been identified
- Cons: Still a bit of a stab in the dark
  - Searches for inputs independently from the program
  - May not be able to execute some paths
- Need to improve the effectiveness further

Memory Isolation

Architecture of Modern Computers



How to ensure safety when sharing memory with untrusted programs?

How to ensure safety when sharing memory with untrusted components?

In the Beginning Days

- Batch Processing
  - Single User, Single Process, Single Machine
- Submitted your code to the person who ran your program on the machine
- Problem: Lots of idle time
  - I/O waits
  - Human/operational delays
  - Debug/reseting cycles

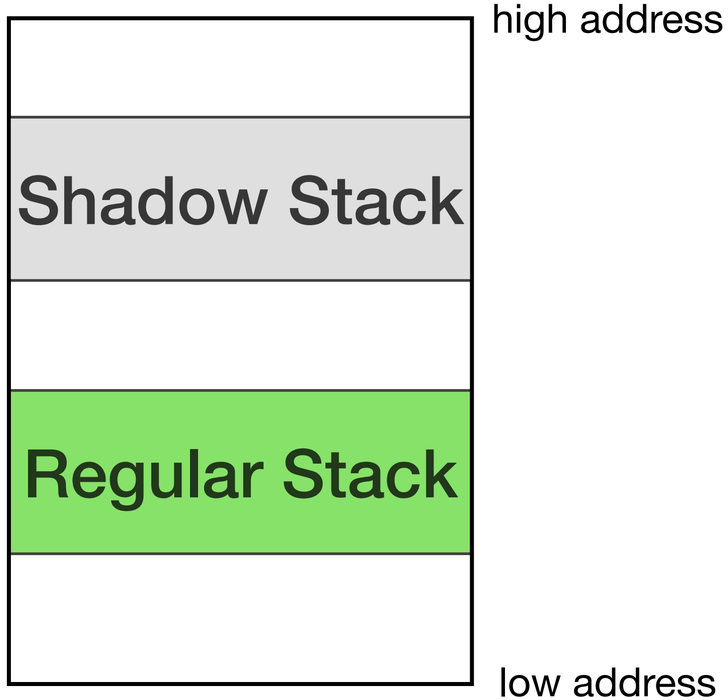
Time Sharing System

- Each program has a “share” of the CPU time
- The program has the impression of the *only* program.
  - Each process is sharing the computer resources.
  - Swapping in and out when share is up or waiting on other resources
  - Appearance of concurrently running process
- **Problem:** How to maintain isolation between programs?

Shadow Stack for Return Address Integrity

A separate stack dedicated to storing a copy of each return address

- A program can use the return address on the shadow stack
  - Checking the validity of the original return address
  - Directly using the copy on the shadow stack to return



Weaknesses of Shadow Stack for Return Addresses

- Increased complexity due to additional abstraction
  - Increasing the complexity of the protected software
  - Performance and memory overhead
  - New security risks
- Limited scope of protection
- Integrity of shadow stack itself
  - Shadow stack protects return addresses, who protects the shadow stack?

Principles for Building Secure Software Systems

- Isolation
- Least Privilege
- Fault Compartmentalization
- Trust and Correctness

Principle: Isolation

Isolate two components from each other

- One component cannot access data/code of the other component except through a well-defined API

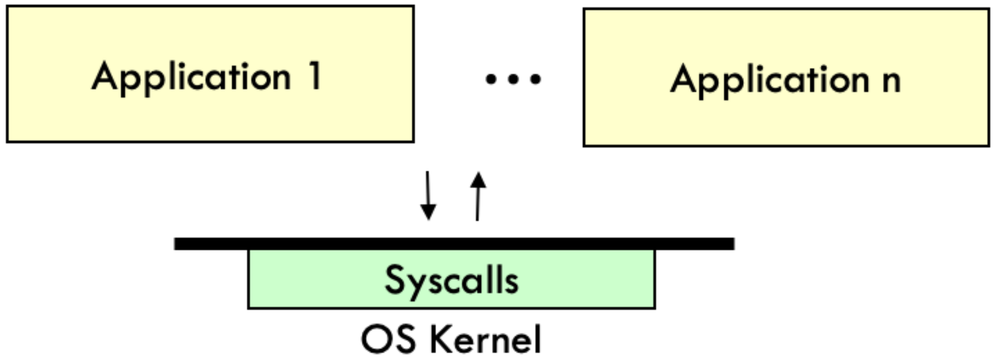
User-space application may only access the disk through the filesystem API (i.e., the OS prohibits direct block access to raw data). The OS isolates the user-space process from the disk.

Isolation incurs overhead due to switching cost between components.

Isolation via Protection Domains

- A fundamental idea in computer security
  - [Lampson 74] “Protection”: https://dl.acm.org/doi/pdf/10.1145/775265.775268
- Structure a computer system to have multiple protection domains
  - Each domain is given a set of privileges, according to its trustworthiness

Example: Separation between OS and Applications



- One OS domain (the kernel mode)
  - Privileged: executed privileged instructions; 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 the OS.

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 would not crash or violate the security of the browser.
  - E.g., isolating device drivers in an OS

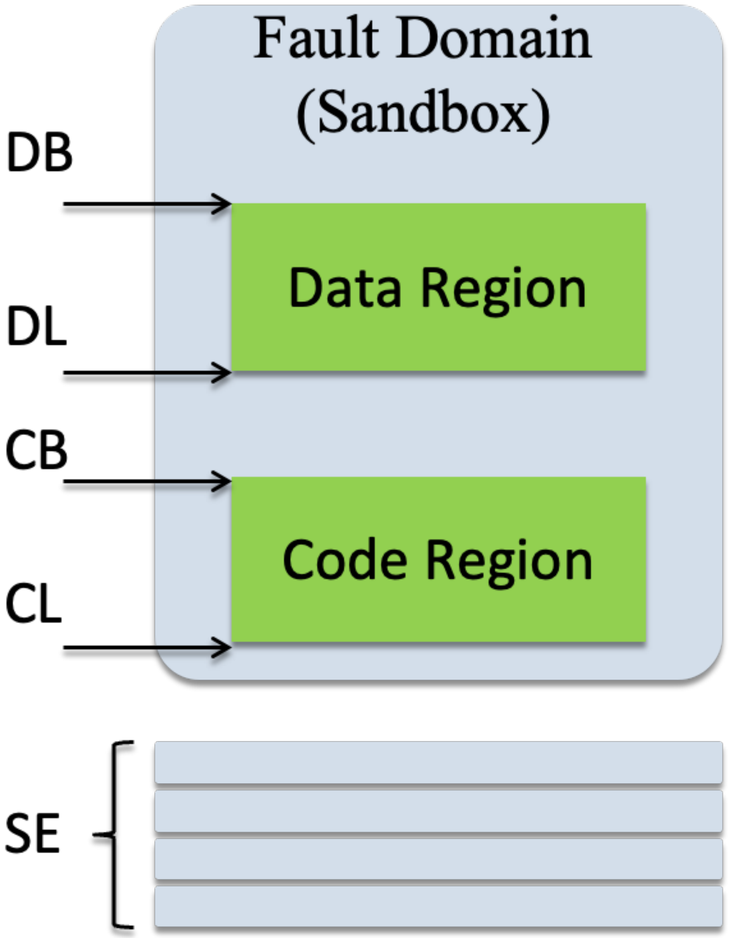
Many Forms of Protection Domains

- Hardware-based virtualization: Each domain in a virtual machine
  - Pros: 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; strong isolation
  - Cons: high context-switch overhead
- Language-based isolation: rely on languages features such as types
  - Pros: fine-grained, portable, flexible, low overhead
  - Cons: high engineering effort to use languages/features

Software-based Fault Isolation (SFI)

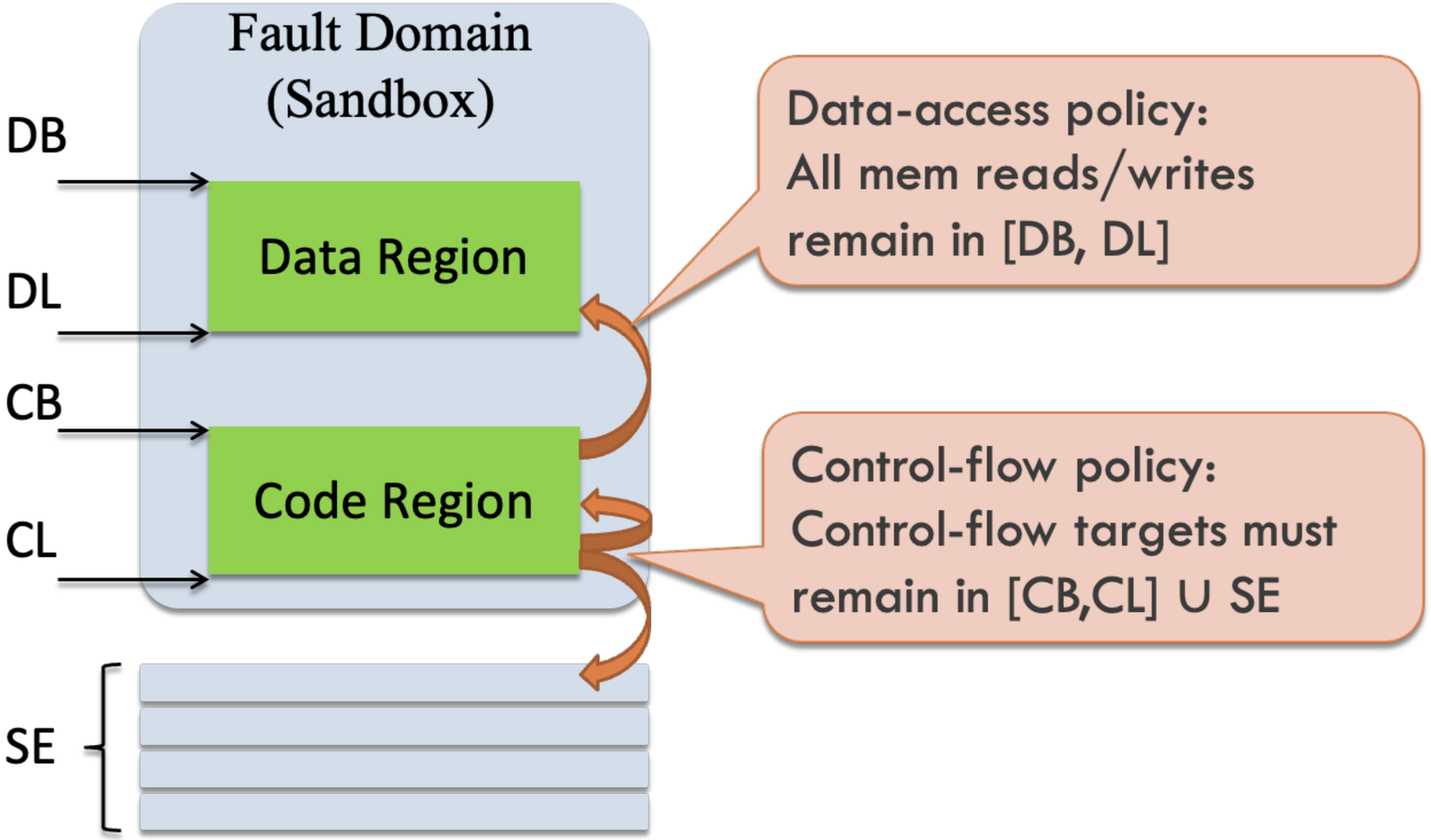
- Introduced by [Wahbe et al. 93] for MIPS
  - [McCamant & Morrisett 06] extended it to x86
  - [PNaCl] Google implemented SFI for ARM, ADM64, & MIPS for Chrome
- SFI is within the the same process address space
  - One type of *intra-address space* isolation
  - Each protection domain has a designated 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 compiler support and software engineering effort

SFI Sandbox Setup



- Data region (DR): [DB, DL]
  - Hold data: stack, heap, global
- Code region (DR): [CB, CL]
  - Hold code
- Safe external (SE) addresses
  - Host trusted services that require higher privileges
  - Code can jump to them for accessing resources.
  - Code can safely transition out of the current domain.
- DR, CR, and SE are disjoint.

SFI Policy


Fault Domain
(Sandbox)
DB
DL
CB
CL
SE
Data Region
Code Region
Data-access policy:
All mem reads/writes
remain in [DB, DL]
Control-flow policy:
Control-flow targets must
remain in [CB,CL] U SE

Implications of the SFI Policy

- Non-writable code
  - All memory writes must write to DR.
  - Code region cannot 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 disallowed

Stronger SFI Policies

- An SFI might implement a stronger/more restrictive policy
  - For implementation convenience
  - For fine-grained safety
  - 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

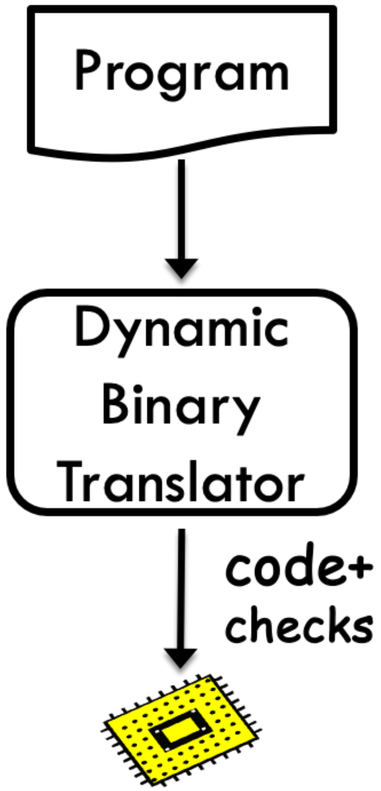
SFI Enforcement Overview

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

Dynamic Binary Translation (DBT)

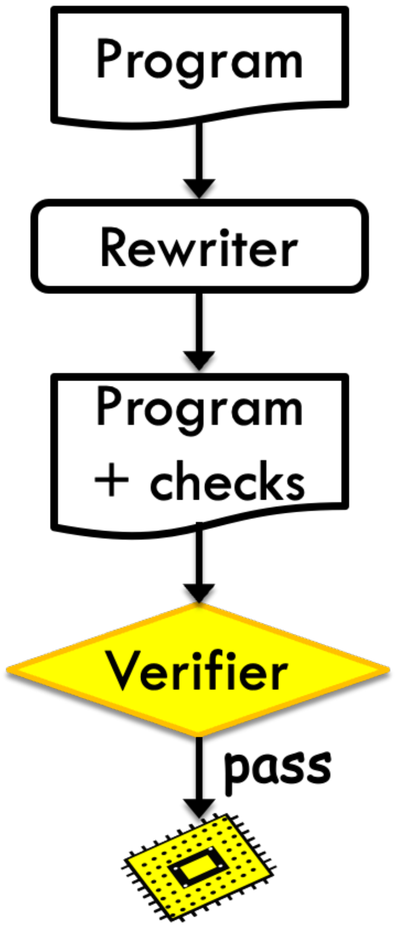
Translates binary code at execution time

- Commonly used in virtualization and instruction emulation.



- No need to modify program's code ahead of time
- Flexible and adaptable to the running environment
- For a dangerous instruction, the interpreter checks if it is safe according to the policy
- High runtime overhead
- Complex to implement
- Writable code region is generally more dangerous.

Inlined Reference Monitors (IRM)



- A static program rewriter
  - Inlines checks into the target 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.

Strategies for Implementing IRM Rewriters

- Binary rewriting
  - Input: binary code
  - Steps: perform disassembly; insert checks; assembly the instrumented code
  - Pros: not requiring source code
  - Cons: hard to disassemble and analyze stripped binaries
- Inside a compiler
  - Input: source or IR code
  - Steps: compiler inlining checks before generating binary code
  - Pros: can perform more optimizations on checks with richer information about code, e.g. types
  - Cons: need source or IR code

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's length
- The program computes in r5 the sum of the array items.

```
int *end = arr + len * 4;
int sum = 0;
while (arr < end) {
    sum += *arr;
    arr++;
}
```

Naive Enforcement

```
  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:
```

- Insert checks before memory reads/writes

Assume we want to ensure memory access is contained in data region, we need to insert checks before this memory access:

```
if r3 < DB goto error
if r3 > DL goto error
```

Naive Enforcement

- Sufficient for security requirement for isolation
- Has a high runtime overhead
  - Two checks per memory access
- A practical SFI needs to implement optimizations to drive down the cost.
  - E.g., remove redundant checks

Optimization: Integrity-only Isolation

- A program performs many more reads than writes.
  - In SPEC2006, 50% 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 or less of a concern.
- 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

Jaleel 2010: https://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf

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`
- `r6 = mem(r3)` becomes

```
r10 = r3 >> 16 // right shift 16 bits to get the region id
if r10 != 0x1234 goto error
r6 = mem(r3)
```

Optimization: Address Masking

- Address checking stops the program when the check fails
  - Strictly speaking, unnecessary for isolating faults

Software-based Fault Isolation (SFI)

- Introduced by [Wahbe et al. 93] for MIPS
  - [McCamant & Morrisett 06] extended it to x86
  - [PNaCl] Google implemented SFI for ARM, ADM64, & MIPS for Chrome
- SFI is within the the same process address space
  - One type of *intra-address space* isolation
  - 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 compiler support and software engineering effort

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, but 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`
  - The data region ID is `0x1234`
- Instead of

```
r10 = r3 >> 16 // right shift 16 bits to get the region id
if r10 != 0x1234 goto error
r6 = mem(r3)
```

- `r6 = mem(r3)` becomes

```
r3 = r3 & 0x0000FFFF  // bit-mask to clear the first 16 bits
r3 = r3 | 0x12340000 // bit-mask to set the first 16 bits to 0x1234
r6 = mem(r3)
```

Anything suspicious/questionable?

Wait! What about Program Semantics?

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

Optimization: One-instruction Address Masking

- Idea
  - The data region ID has only a single bit on.
  - Mark the zero-ID region unmapped in the address space
- A memory access is safe
  - if an address is either in the data region or in the zero-ID region
  - an access to the zero-ID region generates a hardware trap because it accesses unmapped memory
- Benefit: Cutting down one instruction for masking
  - PittSFIeld reported 10% performance gain on SPECint2000

Optimization: One-instruction Address Masking

- Example: `DB = 0x20000000` ; `DL = 0x2000FFFF`
  - The data region ID is `0x2000`
- `r6 = mem(r3)` becomes

```
r3 = r3 & 0x2000FFFF  // bit-mask to set the region ID
r6 = mem(r3)
```

- Result is an address in DR or in the unmapped zero-ID region.
- Cons: Limit the number of domains
  - In a 32-bit system, if a DR's size is 2^n, then we can have at most $32-n$ fault domains.

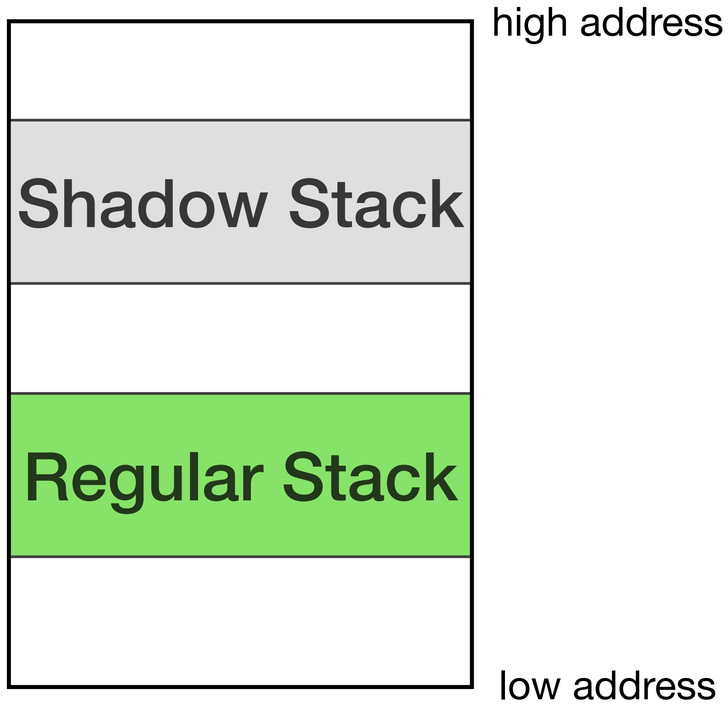
Data Guards

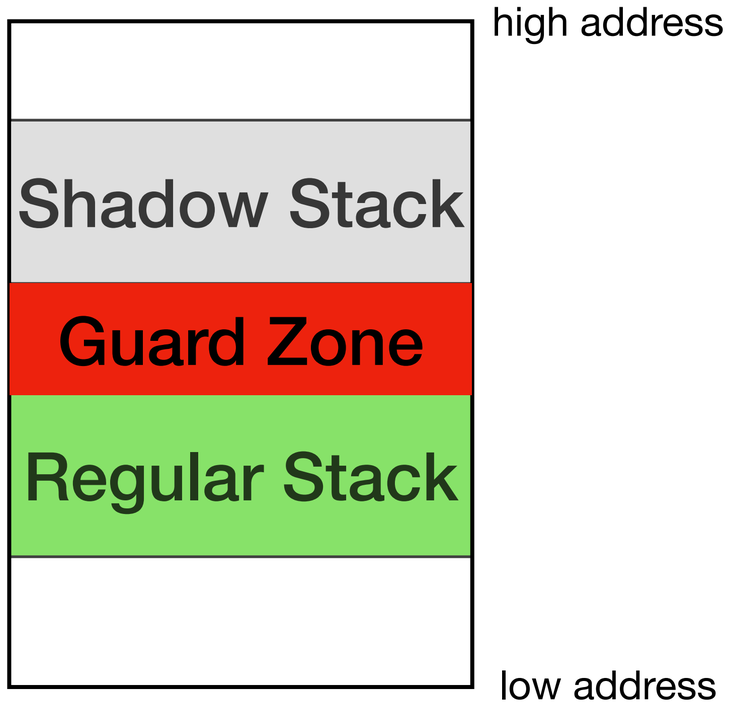
- A data guard refers to either address checking or address masking.
  - When which one is used is irrelevant.
- Introduce a pseudo-instruction “`r'=dGuard(r)`”
  - To hide implementation details
- An implementation should satisfy the following properties of `r'=dGuard(r)`”
  - If r is in DR, then r’ 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 often DR.

Shadow Stack for Return Address Integrity

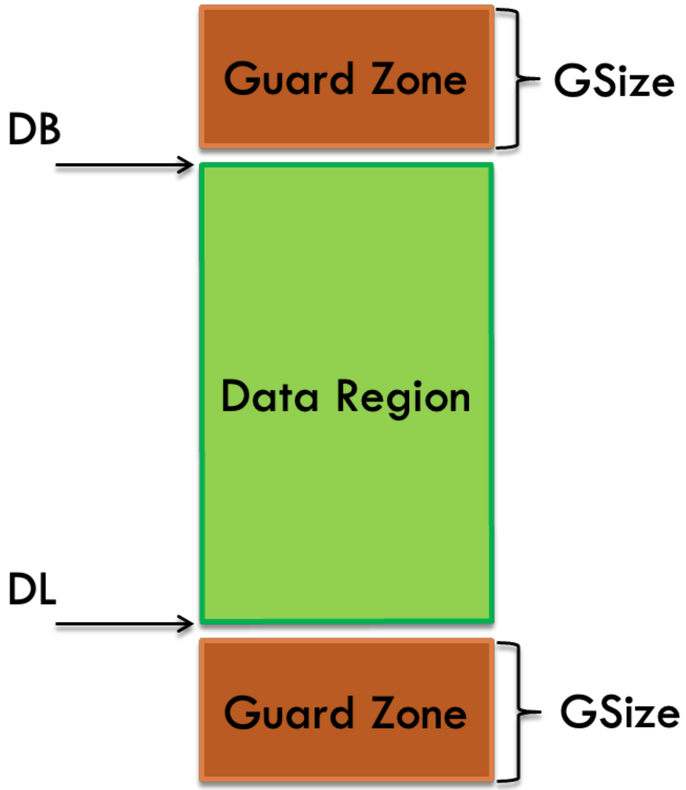
A separate stack dedicated to storing a copy of each return address

- A program can use the return address on the shadow stack
  - Checking the validity of the original return address
  - Directly using the copy on the shadow stack to return



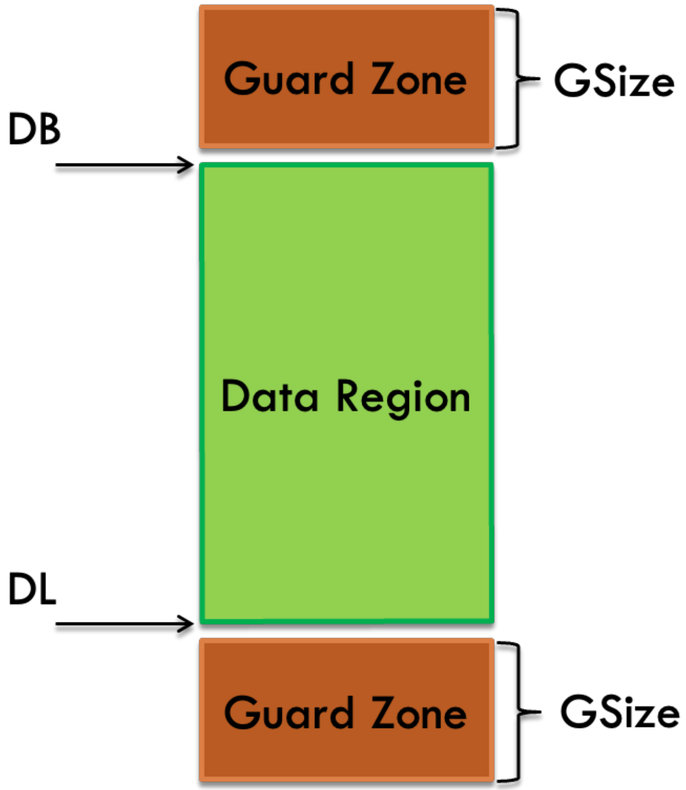


Optimization: Guard Zone/Page



- Place a guard zone before/after a data region.
- Guard zones are unmapped or not readable/writable.
  - Access to guard zones are trapped by hardware.
- Assume Guard Zone's size is GSize, a memory read/write is safe if the address is in [DB-GSize, DL+GSize].
- Also called red zone

Guard Zones Enable More Optimizations



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

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: `r6 = mem(r3 + 12)` becomes

```
r3 = dGuard(r3)
r6 = mem(r3 + 12)
```

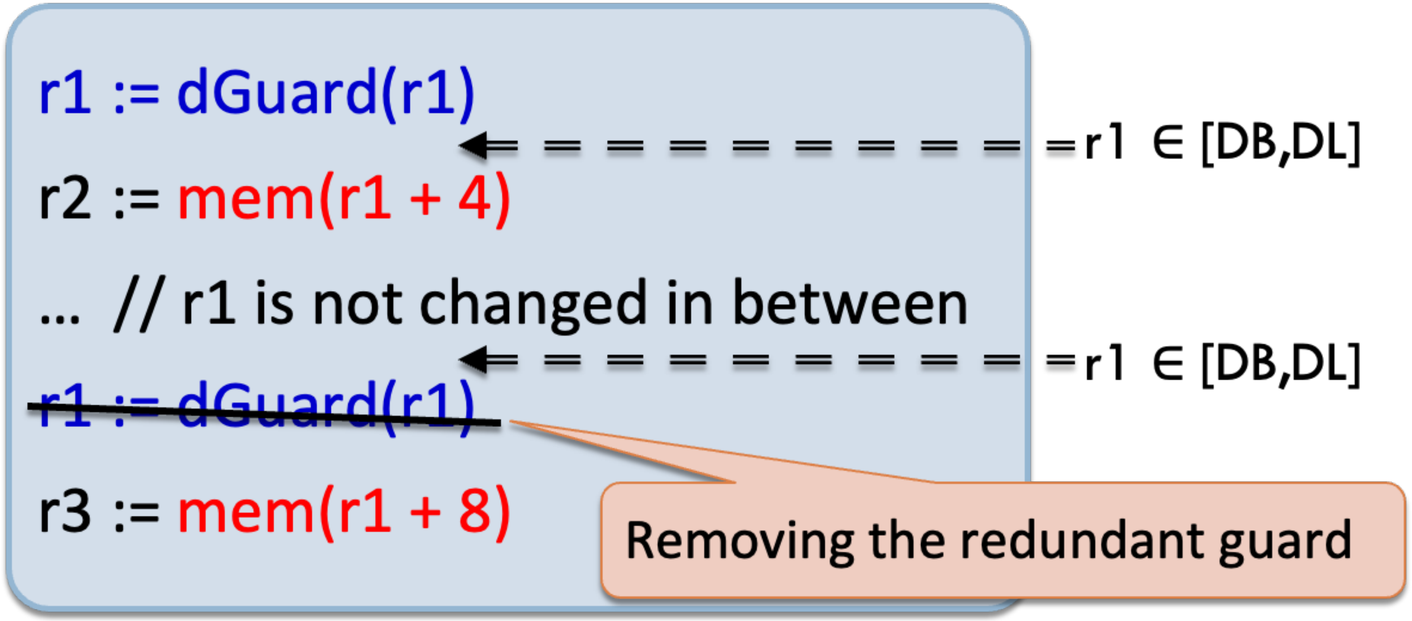
- Why is the above safe?
  - "`r3 := dGuard(r3)`" constrains r3 to be in DR and then `r3+12` must be in [DB-GSize, DL+GSize], assuming GSize ≥ 12.

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].

Optimization: Redundant Check Elimination

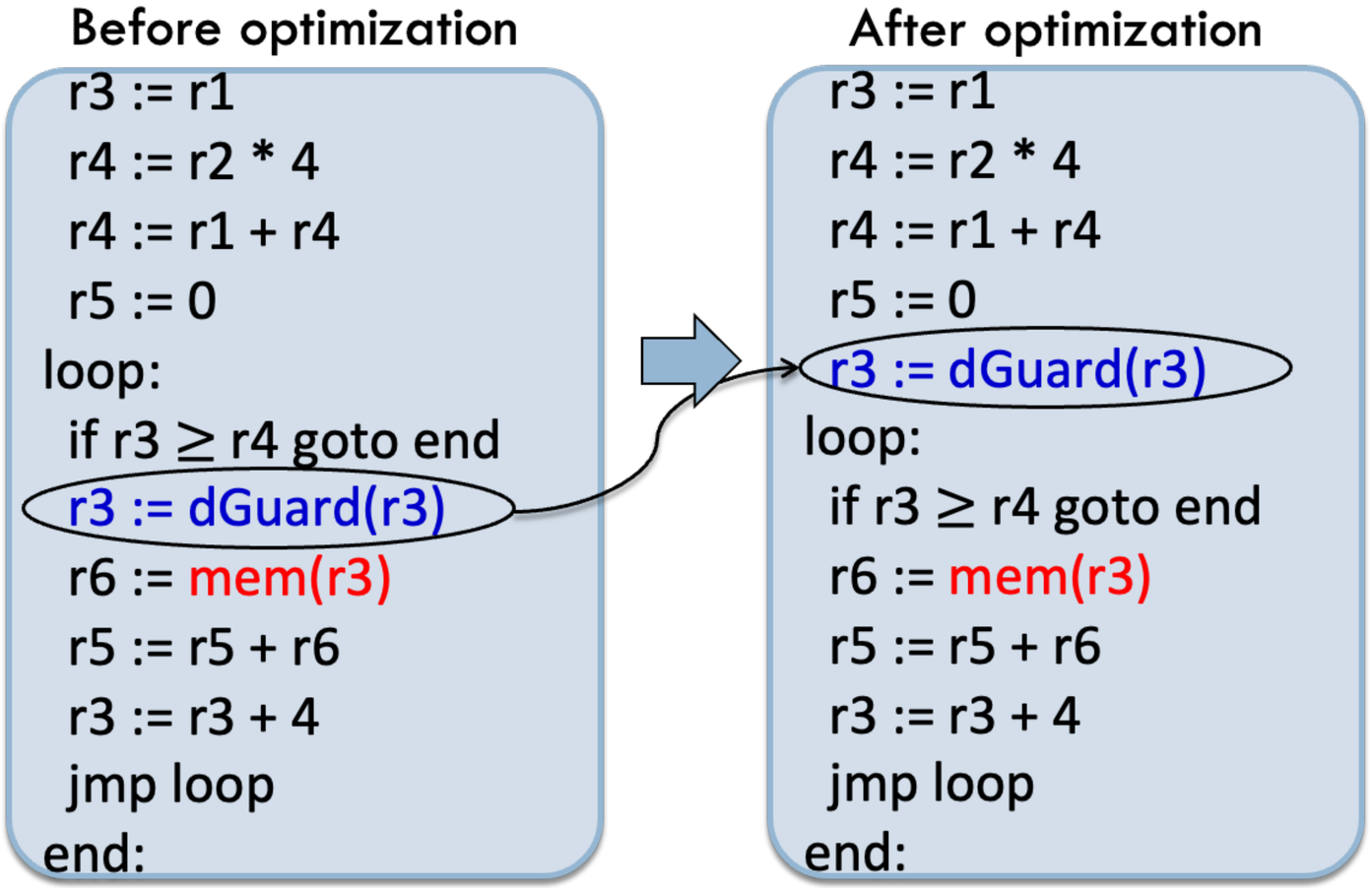
- Idea: perform range analysis to know the range of values of registers and use that to remove redundant data guards



Optimization: Loop Check Hoisting

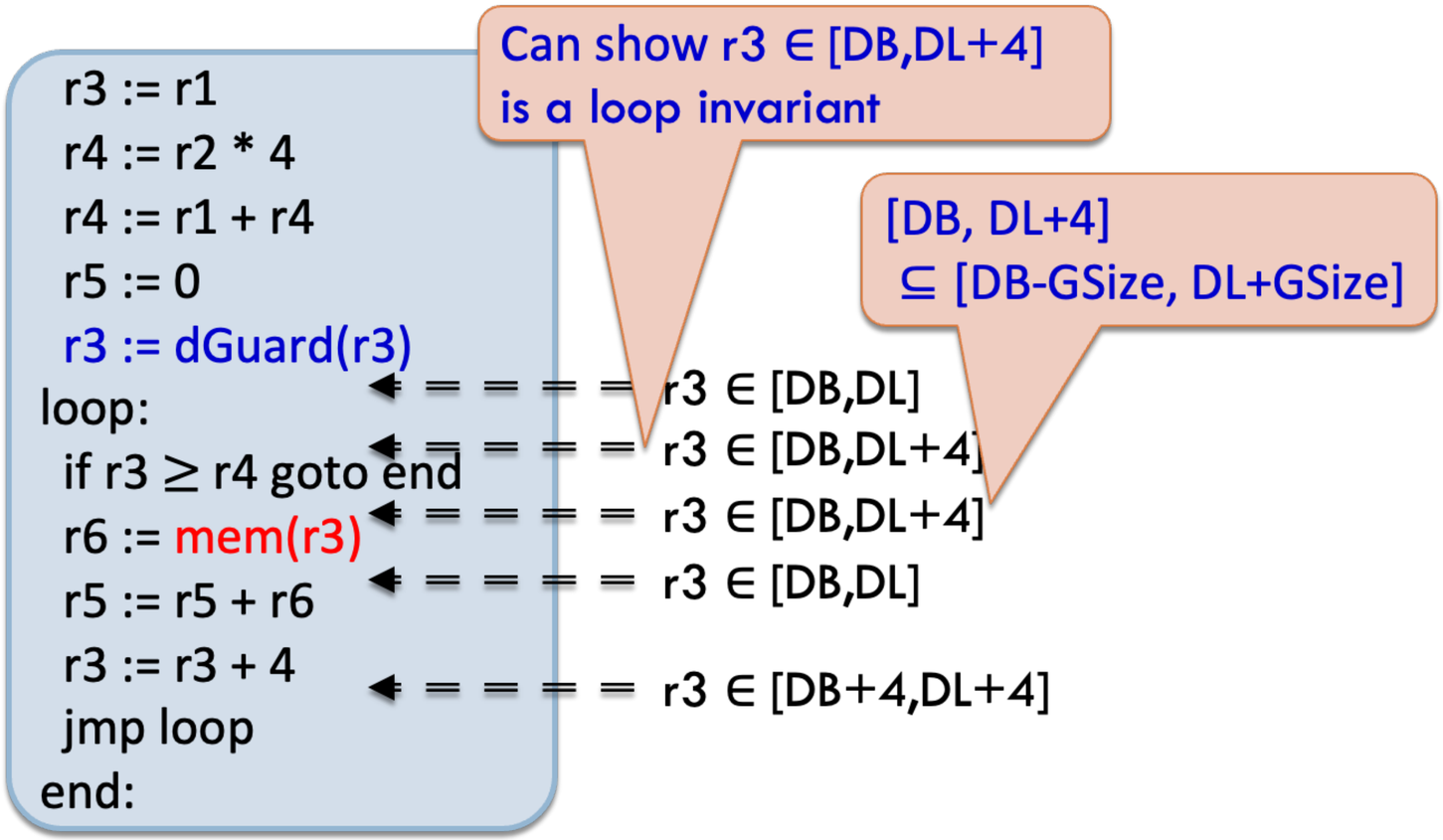
- Idea: A guard in a loop is hoisted outside of the loop.
  - The guard is performed only once per loop instead of once per loop iteration.
- Key observation
  - If addr $\in$ [DB-GSize, DL+GSize], then a successful (untrapped) memory operation via addr means addr $\in$ [DB, DL].

Loop Check Hoisting Example



* 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 Optimized Code Safe?



Optimization: Guard Changes Instead of Uses

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

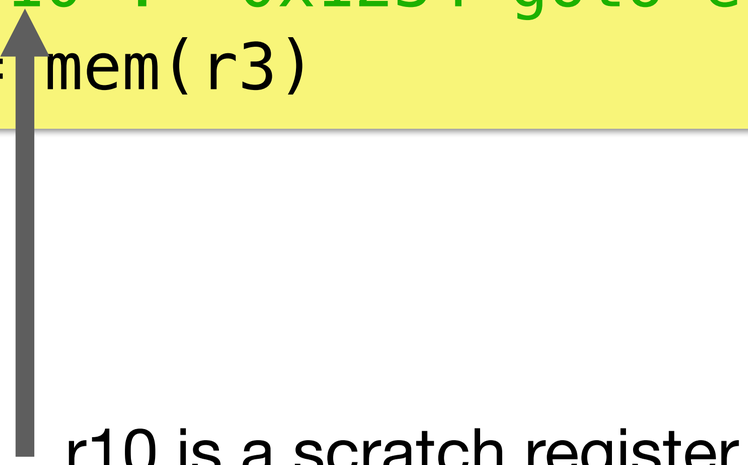
```
ebp = esp
ebp = dGuard(ebp)
```

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

Optimization: Data Region Specialization

- Example: `DB = 0x12340000; DL = 0x1234FFFF`
  - The data region ID is `0x1234`
- `r6 = mem(r3)` becomes

```
r10 = r3 >> 16 // right shift 16 bits to get the region id
if r10 != 0x1234 goto error
r6 = mem(r3)
```



r10 is a scratch register

Scratch Registers

- The SFI rewriting may require finding scratch registers to store intermediate results.
- If the old values of scratch registers need to be used later, we 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.
- 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 an unlimited number of virtual registers.
    - E.g. LLVM IR
    - A later register allocation phase maps those variables to registers or stack slots.

Anything Vulnerable about This Program?

```
  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:
```

What if a control flow hijacking (e.g., by corrupting an return address) causes the control flow to jump over dGuard and directly go to the memory access?

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

CFI is often needed for other security policies such as SFI.

How to Enforce Control-flow Integrity

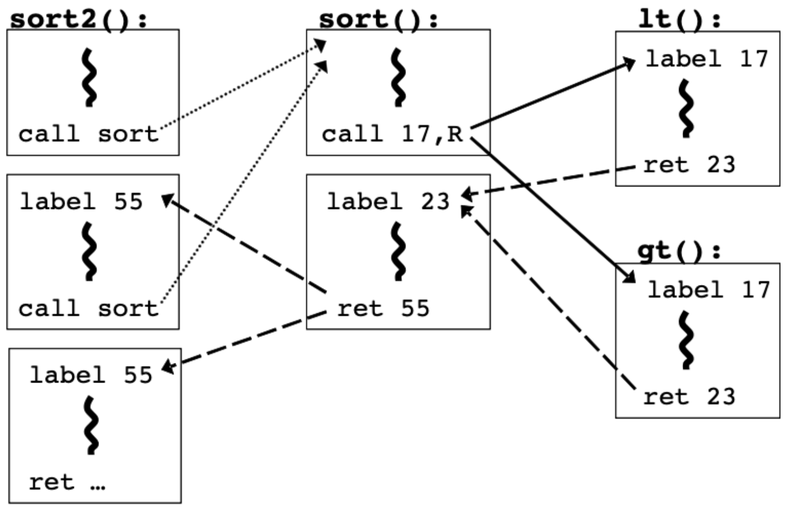
- Compute a CFG
- For indirect control flow transfers, compute their target destinations
  - Mostly via compiler or binary rewriting, but possible at run-time
- Before an indirect transfer, check the validity of the destination
- Two CFI policies:
  - Label-based and type-based

Label-based CFI

- Assign and insert a label (ID) before each indirect transfer destination
- Before executing an indirect transfer, check the destination's label
  - Similar to using stack canaries / shadow stacks

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Direct forward transfer

Indirect forward transfer

Backward transfer

Example of Label-based CFI

Bytes (opcodes)	x86 assembly code	Comment
FF E1	jmp ecx	; a computed jump instruction

CFI instrumentation

```
81 39 78 56 34 12    cmp  [ecx], 12345678h  ; compare data at destination
75 13                jne  error_label       ; if not ID value, then fail
8D 49 04             lea  ecx, [ecx+4]      ; skip ID data at destination
FF E1                jmp  ecx               ; jump to destination code
```

Bytes (opcodes)	x86 assembly code	Comment
8B 44 24 04 ...	mov eax, [esp+4]	; first instruction ; of destination code

CFI instrumentation

```
78 56 34 12          DD 12345678h           ; label ID, as data
8B 44 24 04          mov  eax, [esp+4]      ; destination instruction
...
```

Align-chunk Enforcement

- Divide the code into chunks of some size
  - E.g., 16 or 32 bytes
- Each chunk starts at an aligned address
  - So we can force an address to align by chunkSize with "`addr / chunkSize`"
- Make dangerous instructions 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 of Align-chunk Enforcement

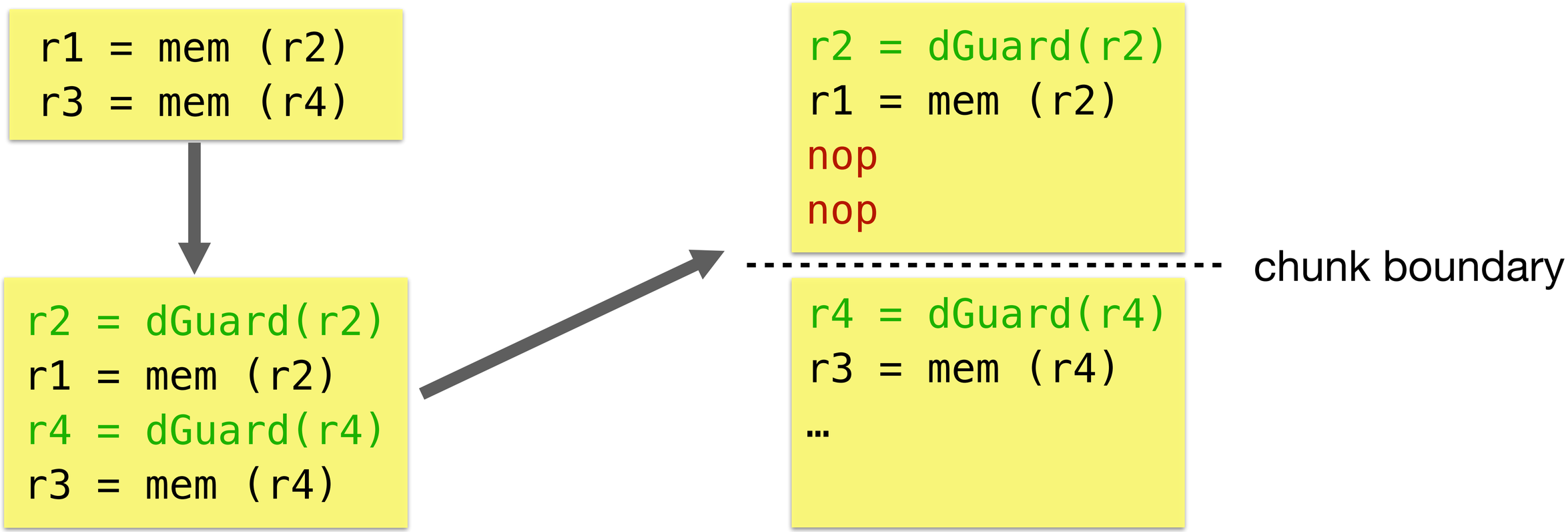
- Assume
  - CR is `[0x10000000, 0x1000FFFF]`; code region ID is `0x1000`
  - Chunk size is 16 bytes.
  - Zero-ID region `[0x00000000, 0x0000FFFF]` unmapped
- Then, "`jmp rax`" becomes

```
rax = rax & 0x1000FFF0
jmp rax
```

- Ensures that the target address is
  - in CR or zero-ID region
    - after &, r's upper 16 bits must be either 0x0000 or 0x1000
  - a chunk beginning
    - after &, r's lower four bits must all be 0, meaning it's 16-byte aligned

Downside of Align-chunk Enforcement

- All legitimate jump targets have to be aligned.
  - No-ops have to be inserted for that.
- E.g., assuming a 16-byte chunk, and each instruction is 4-byte long.



Downside of Align-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.

SFI Applications

- 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 & PNaCl)

- SFI service in Chrome
- 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
- Replaced by WebAssembly in 2017

DOOM in NaCl

NaCl: Code Verification

- Code is verified before running
  - Allow a restricted subset of x86 instructions
    - No unsafe instructions: 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 memory writes and computed jumps

SFI Review

- SFI policy: data-access policy; control-flow policy
- SFI enforcement: inlined reference monitoring (IRM)
- Enforcing data-access policy
  - Naïve enforcement and optimizations for the data-access policy
    - Data masking; guard zones; …
- Enforcing control-flow policy
  - Additional constraints: must prevent checks from being bypassed; must prevent jumping into middles of instructions

Hardware-assisted Memory Isolation

Memory Mapping of vim

```
$ cat /proc/147967/map
map_files/ maps
jie@fedora: /home/jie
$ cat /proc/147967/maps
564b3aef8000-564b3aefd000 r--p 00000000 00:20 160559                     /usr/bin/vim
564b3aefd000-564b3b235000 r-xp 00005000 00:20 160559                     /usr/bin/vim
564b3b235000-564b3b2a1000 r--p 0033d000 00:20 160559                     /usr/bin/vim
564b3b2a1000-564b3b2b5000 r--p 003a8000 00:20 160559                     /usr/bin/vim
564b3b2b5000-564b3b2e9000 rw-p 003bc000 00:20 160559                     /usr/bin/vim
564b3b2e9000-564b3b2f8000 rw-p 00000000 00:00 0
564b3b349000-564b3b75f000 rw-p 00000000 00:00 0                          [heap]
7fe39c600000-7fe3aa11d000 r--p 00000000 00:20 21612                      /usr/lib/locale/locale-archive
7fe3aa12a000-7fe3aa12e000 rw-p 00000000 00:00 0
7fe3aa12e000-7fe3aa130000 r--p 00000000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa130000-7fe3aa133000 r-xp 00002000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa133000-7fe3aa134000 r--p 00005000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa134000-7fe3aa135000 r--p 00005000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa135000-7fe3aa136000 rw-p 00000000 00:00 0
7fe3aa136000-7fe3aa138000 r--p 00000000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa138000-7fe3aa1a8000 r-xp 00002000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1a8000-7fe3aa1d0000 r--p 00072000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d0000-7fe3aa1d1000 r--p 00099000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d1000-7fe3aa1d2000 rw-p 0009a000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d2000-7fe3aa1fa000 r--p 00000000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa1fa000-7fe3aa363000 r-xp 00028000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa363000-7fe3aa3b1000 r--p 00191000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b1000-7fe3aa3b5000 r--p 001de000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b5000-7fe3aa3b7000 rw-p 001e2000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b7000-7fe3aa3c1000 rw-p 00000000 00:00 0
7fe3aa3c1000-7fe3aa3c3000 r--p 00000000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c3000-7fe3aa3c6000 r-xp 00002000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c6000-7fe3aa3c7000 r--p 00005000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c7000-7fe3aa3c8000 r--p 00005000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c8000-7fe3aa3c9000 rw-p 00006000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
```

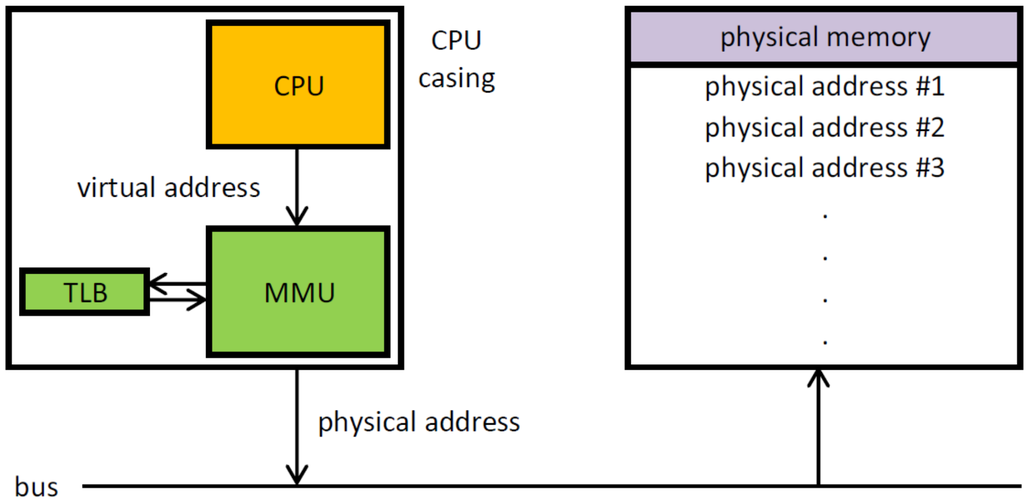
How is the memory access permission enforced?

Hardware-enforced Memory Access Configuration

- Memory access permissions, e.g., read/write/executable, checked and enforced by hardware
- Two primary types:
  - Memory Management Unit (MMU)
    - Supports virtual address
    - Mainly for general-purpose computing systems, such as desktops/smartphones
  - Memory Protection Unit (MPU)
    - Flat address space—no virtual address
    - Mainly for low-end embedded systems

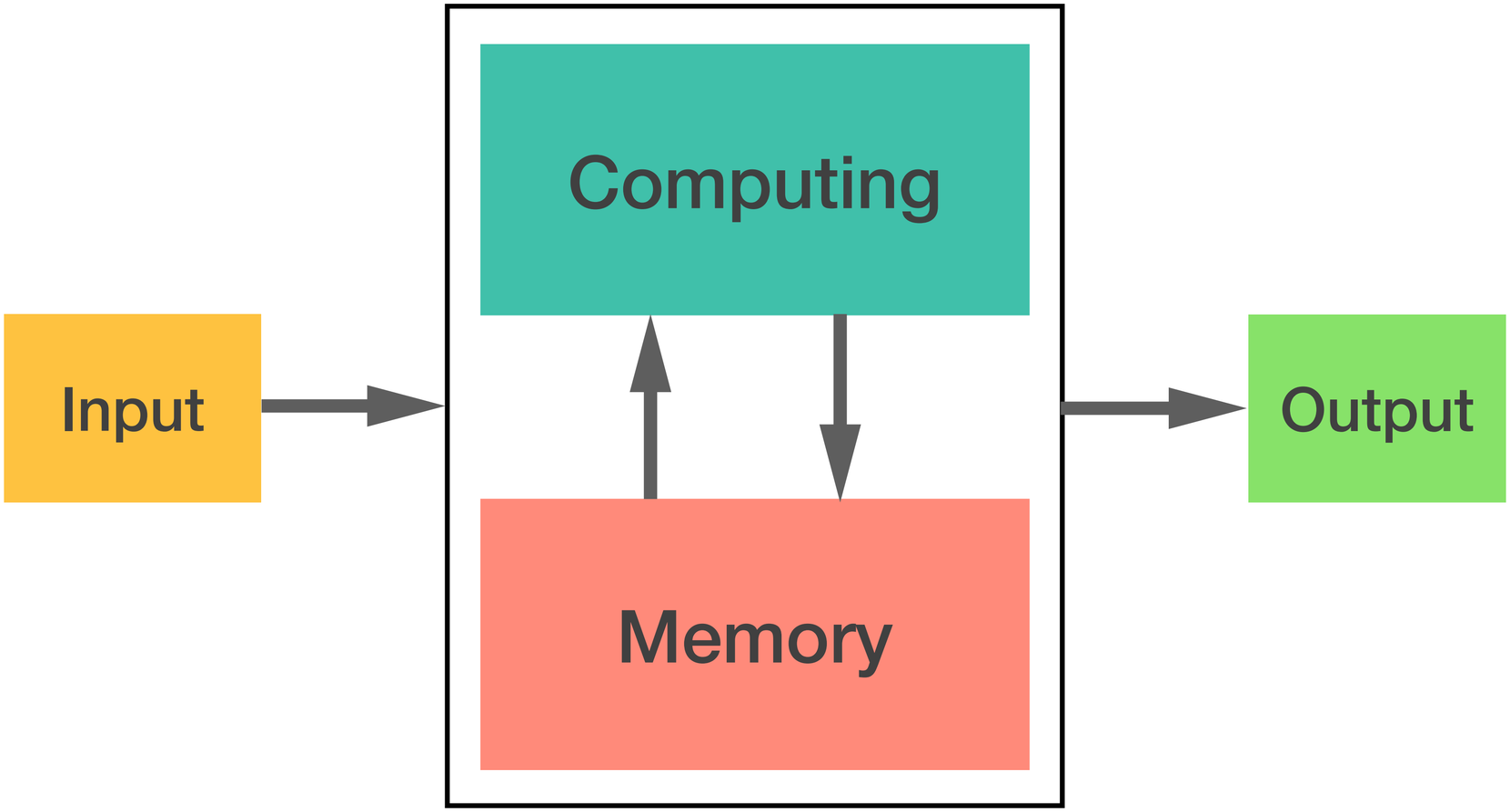
Memory Management Unit (MMU)

- A hardware unit that manages memory
  - Translating virtual addresses to physical addresses
  - Examining memory access permissions (memory protection)
  - Also known as paged MMU
    - Memory managed in fixed-size blocks called *pages*



CPU: Central Processing Unit
MMU: Memory Management Unit
TLB: Translation lookaside buffer

Architecture of Modern Computers



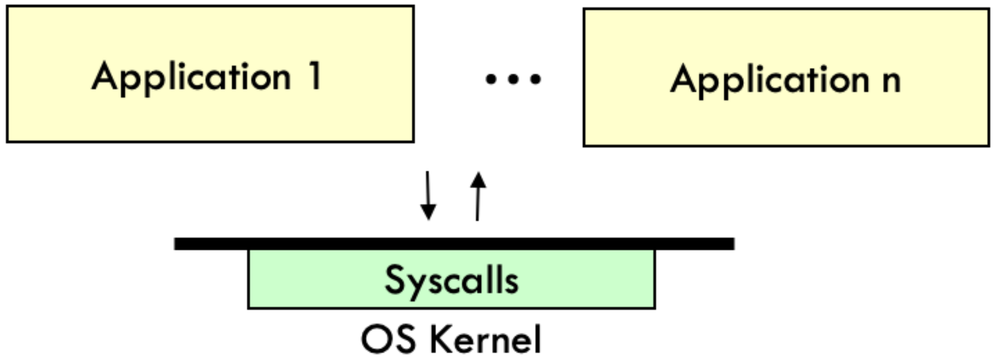
Memory Page

- A page is a fixed-size block of memory, determined by OS + architecture
  - 4 KB in Linux/AMD64 by default; 2 MB for huge pages;1 GB for gigantic pages
  - 16 KB in MacOS/AArch64; 2 MB for huge pages
- The smallest memory allocation unit requested by OS.
- All memory addresses in a page share the same properties.

Utilize MMU for Memory Isolation

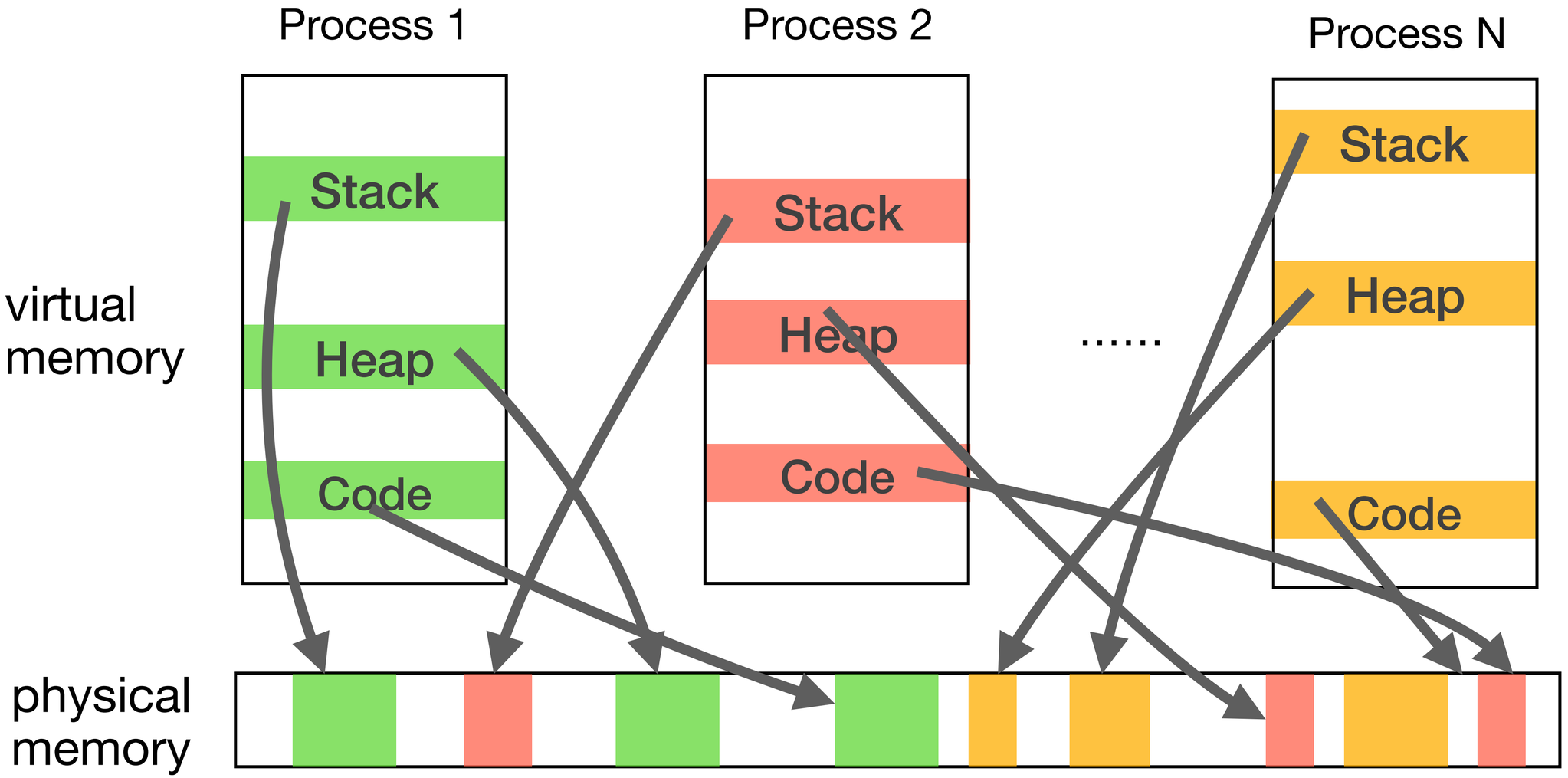
- Inter-process memory isolation
- Intra-process memory isolation

Example: Separation between OS and Applications



- One OS domain (the kernel mode)
  - Privileged: executed privileged instructions; 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.

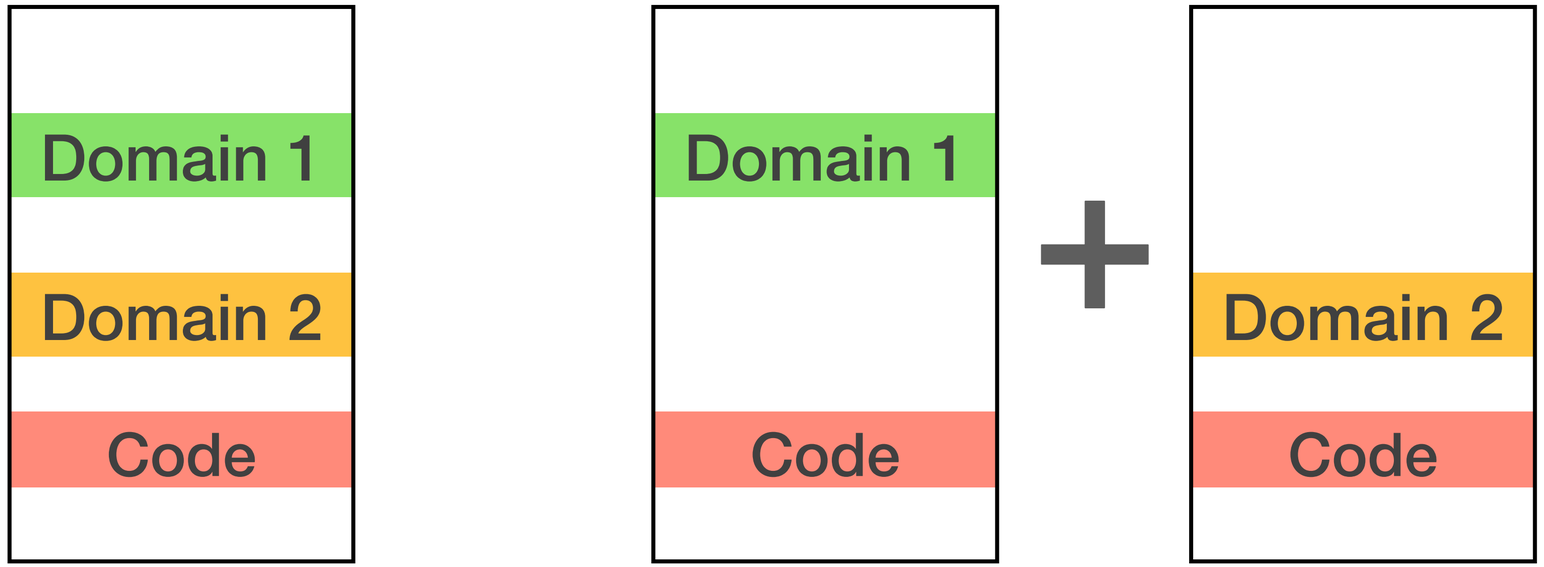
Isolation Between Processes



Virtual to physical memory mapping naturally creates *protection domains* between processes.

Use Inter-process Isolation to Implement Intra-process Isolation

- Put different memory domains into different processes
- Use inter-process communication (IPC) for different domains to “talk”



- Pro: Strong isolation enforced by hardware
- Cons: Extremely high performance cost
- Must be very careful about the communication interface

Page Table and Page Table Entries

- Page table is a data structure used by MMU to store mappings between virtual and physical memory addresses.
- Page table entry (PTE) is an entry in a page table representing a page.
  - Mapping from a virtual address to a physical page
  - Other information about the page

Example: Intel x64 PTE That Maps a 4-KB Page

63	62…59	58…52	51…M	M-1…12	11…9	8	7	6	5	4	3	2	1	0
XD	PK	Reserved	Reserved	PFN	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

- P: Present
- R/W: Read/Write
- PWT: Page-level write-through
- PCD: Page-level cache disable
- A: Accessed
- D: Dirty
- G: Global
- AVL: Available for software to define
- PFN: Physical frame number (physical address)
- PK: Protection key (if supported)
- XD: execute-disable

Configure Page Table Entry to Isolate Memory

- With data guards, memory access “`mem(r1) = r2`” becomes

```
r1 = dGuard(r1)
mem(r1) = r2
```

- Alternative: Before `mem(r1)`, set all memory domains outside of r1 to unwritable.
  - In *nix systems, use `mprotect()` syscall.

Configure Page Table Entry to Isolate Memory

- With data guards, memory access "`mem(r1) = r2`" becomes

```
r1 = dGuard(r1)
mem(r1) = r2
```

- Alternative: Before `mem(r1)`, set protected memory domains outside of r1 to unwritable, and resume the permission afterwards.
  - In *nix systems, use `mprotect()` syscall.

```
mprotect(protected_mem, PROT_NONE);
mem(r1) = r2
mprotect(protected_mem, PROT_READ | PROT_WRITE);
```

  - Pros: Strong protection
  - Cons: High performance penalty; introducing security hazards

Example: Intel x64 PTE That Maps a 4-KB Page

63	62…59	58…52	51…M	M-1…12	11…9	8	7	6	5	4	3	2	1	0
XD	PK	Reserved	Reserved	PFN	AVL	G	PAT	D	A	PCD	PWT	U/S	R/X	P

- P: Present
- R/W: Read/Write
- PWT: Page-level write-through
- PCD: Page-level cache disable
- A: Accessed
- D: Dirty
- G: Global
- AVL: Available for software to define
- PFN: Physical frame number (physical address)
- **PK: Protection key (if supported)**
- XD: execute-disable

Intel Memory Protection Key (MPK)

- A *protection key* represents an access permission configuration.
  - E.g., PK2 set to read-only, and PK5 set to read + write
- Memory pages are divided into different groups.
- A group of pages are associated with a protection key.
- Supports up to 16 protections keys (bits 59–62 in PTE)
  - I.e., 16 different protection domains
- Register `pkru` (Protection Key Rights for User Pages) for memory access checks
  - 32-bit register
  - Every two bits represents the memory access permission of one PK.
    - first bit: Access Disabled (AD) when set to 1
    - second bit: Write Disabled (WD) when set to 1

pkru Register

- E.g., a PTE's has PK13, and PK13 is 10
  - Meaning this page of memory is set to be read-only
- E.g., a PTE's has PK5, and PK5 is 00
  - Meaning this page of memory is set to be readable and writable

How to Manage MPK

- Use syscall `pkey_mprotect()`

```
int pkey_mprotect(void *addr, size_t len, int prot, int pkey);
```

  - Convenient, but slow (~1,100 CPU cycles)
- Directly manipulating `pkru` using the `wrpkru` instruction
  - Allows a program to change the memory access permissions for selected PK
    - `eax` contains the new PK value to be set
    - `ecx` and edx must be 0
  - Fast (~23 CPU cycles)
- `rdpkru` is used to read PKs into `eax`.

Configure Page Table Entry to Isolate Memory

- With data guards, memory access "`mem(r1) = r2`" becomes

```
r1 = dGuard(r1)
mem(r1) = r2
```

- Alternative: Before `mem(r1)`, set protected memory domains outside of r1 to unwritable, and resume the permission afterwards.
  - In *nix systems, use `mprotect()` syscall.

```
mprotect(protected_mem, PROT_NONE);
mem(r1) = r2
mprotect(protected_mem, PROT_READ | PROT_WRITE);
```

  - Pros: Strong protection
  - Cons: High performance penalty; introducing security hazards

Example of Protecting Memory Domain with MPK

- Assume protected memory domain is associated with `pk1`.
- For dangerous instruction "`mem(r1) = r2`", it becomes

```
xor %ecx, %ecx
xor %edx, %edx
rdpkru
or %eax, 0x00000004
wrpkru
mem(r1) = r2
… // recover original PKs
```

Anything vulnerable about this solution?

How to make sure your MPK gate instructions (those transitioning to/from a specific PK configuration) are respected?

Check the optional readings for this lecture.

MPK Summary

- 16 protection keys; allowing 16 protection domains
- PK represented by bits 59 to 62 in a PTE
- `pkru` register is used to check memory access permissions.
- `pkey_mprotect()` syscall for managing PK
- `rdpkru`/`wrpkru` instructions are used to read/write `pkru`.