Scribe Notes: Control Flow Integrity

Alex Thomas

February 18, 2021

Control Flow Graph

- Lines of code is a node in a graph a.k.a basic blocks or a sequence of code that don't branch/jump
- Any control flow (if statements, calling a function) are edges I.E. **If** statements may have 2 edges (taken or not taken)
- Built on compile-time

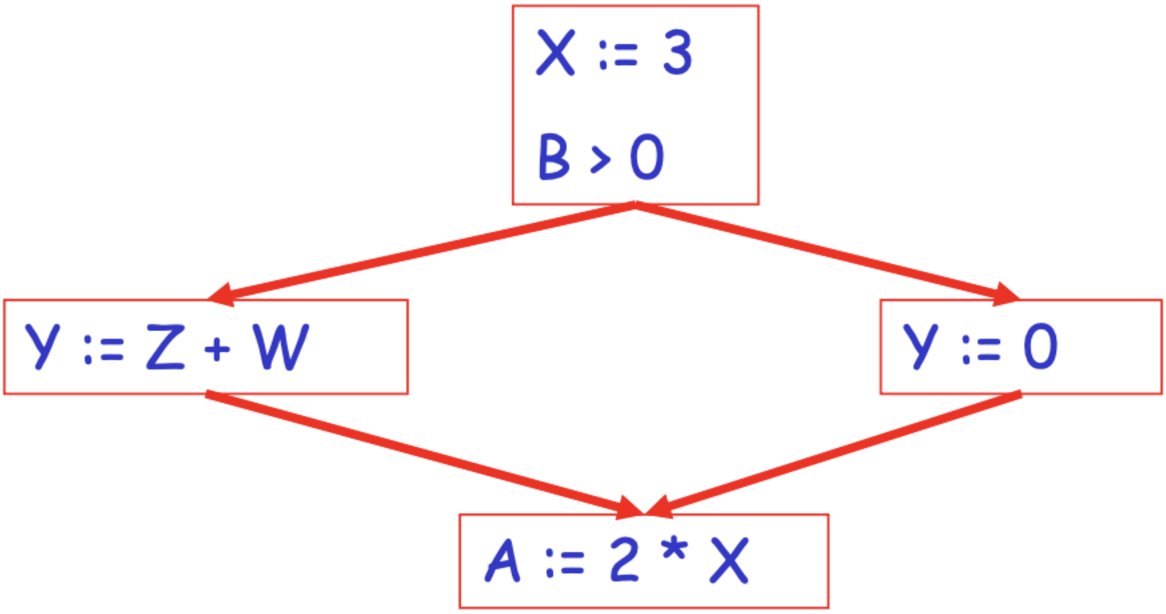

Figure 1: Illustration of CFG from CS164, Koushik Sen

Forward Edges

- Indirect jumps or indirect calls
- These are assembly instructions that jump to a register (the jump address is not fixed)

Backward edges

These are return instructions

Why did the CFI paper not protect against backward edges?

- Paper states that there are other techniques one can deploy to protect against forward edges
- **Performance cost** in trying to protect against backward edges.

- Why?
  - There are two calls
  - Direct calls to a fixed address (i.e. **call 0x1234**)
  - Indirect calls to a register value. (i.e. **jmp %rax**)
  - There are many more direct calls than indirect calls; each call has a return
  - **How to protect against return instructions?** *Use a shadow stack.*
    - Have a normal stack frame
    - Have another stack just for return addresses
    - Check if the return address and the normal stack upon popping match.
    - There is a 10% performance overhead, **but** there is hardware support from Intel that will mitigate this overhead to a 3.5% overhead

Coarse policy (one set of valid jump/call targets)

- Set of valid jump/call targets
- **How do we get all indirect targets?** *Can do some form of static analysis. Find all ways an address of a function pointer is set to a register. We can filter out further by static code analysis by looking at the function signatures.*

Precise Policy (one set of valid jump/call targets per indirect call)

- For each indirect call, find a set of all values the register can take

  Springboard

  - Naive:

    ```
    call *rax
    ```

    Transformed into..

    ```
    check rax; jump if error;
    call *rax
    ```

  - Array has one entry for each valid function that you can jump to. For example, if a target can jump to function **f**, **g**, or **h**

    ```
    0x1000 jump f
    0x1004 jump g
    0x1008: jump h
    0x100c: halt
    ...
    call *rax
    ...
    and *rax 0xF; or *rax 0x1000 # Check that %rax is within the springboard range
    ```

  - Enforces that rax is within the springboard by doing a bitmask

Attacks against CFI

Can CFI stop the following attacks?

Malicious Code Injection

- Attacker cannot overwrite existing code and if they did write code outside existing code, CFI would prevent this.

ROP Attacks

- It does not stop ROP attacks because return statements (backward edges) are outside the scope of CFI.
- Is this ok?
  - ASLR helps
  - Bounds checking helps, but has high overhead

Data-flow Attacks

- CFI doesn't protect data flow attacks

Critical Flow Attacks

- A path that can be very dangerous (launch nukes), but still a valid path with respect to the CFG
- Does not protect from these class of attacks as a malicious path still counts as a possible route

Control-flow Bending

- **__kernel_vsyscall**: *This is found in libc and every syscall is routed by calling this function*
- If an adversary is able to find a route to this path, an attacker can route any arbitrary syscall (i.e. execv)
- Virtually a path anywhere in code to __kernel_vsyscall
- May find arbitrary read, write, and call gadgets (subject to CFI policy)
- Coarse policy is vulnerable to control-flow bending
- Precise Policy for forward edges, but no backward edge protection ⇒ vulnerable to control-flow bending
- Precise policy for forward edges and backward edges ⇒ some are vulnerable, some are not

Bounds Checking vs CFI

- Bounds checking is inefficient, but has more reliable protection
- Bounds checking has limitation with object granularity (function pointer in a struct)
- Bounds checking does not do anything about use-after-free attacks (dangling pointers)
- CFI has both spatial and temporal safety; bounds checking is just spatial safety
- Compatibility can be bad for bounds checking; CFI has better compatibility
- How do "use-after-free" errors manifest?

  ```
  for(p = list; p != NULL; p = p->next){
       free(p)
  }
  ```

  Allocated data is freed, but the pointer is used to find the next linked list.

Temporal Memory Bugs

Electric Fence

- Put each object in its own page
  - A 3 byte object is on its own page (4096 bytes)
- If an object is deallocated, revoke all permissions to that page
- Any attempt to access deallocated object will cause a failure

- Can the page be reallocated to another object?
  - Nope!
- Cons
  - Wasteful memory
  - Internal fragmentation
  - Wasteful for virtual address space
  - Wasteful for physical memory
  - Must keep track of used virtual pages
  - Wasteful small objects (internal fragmentation)
  - Pollutes TLB and kernel data structures for virtual memory regions
  - Each object requires a separate translation

Dhurjati-Adve

```
Physical address P: [01 02 03 04 05]
Virtual address V1-> P, V2->P, V3->P, ...
```

- Objects are pushed into the same page
- All objects map to the same page, as aliasing can be done
- When I free O1, revoke permissions on V1
- Other objects may still map to the same page
- Memory allocator has a bitmap to keep track of the objects which are mapped to the same page. Alternatively, we can use a reference counting approach to implement physical page reuse
- Cons
  - Virtual address space is still wasted
  - Pollutes TLB and kernel data structures for virtual memory regions as we still must keep track of used virtual address space

Oscar (Dang et al)

- Monotonically increase your virtual address for allocated objects
- Any object allocated will never reuse virtual address space
- No longer have to keep track of used virtual address space as we just need to keep track of a single marker
- Stops any use-after-free bugs (since no address is ever reused)