Control-Flow Integrity

Yajin Zhou (http://yajin.org)

Zhejiang University

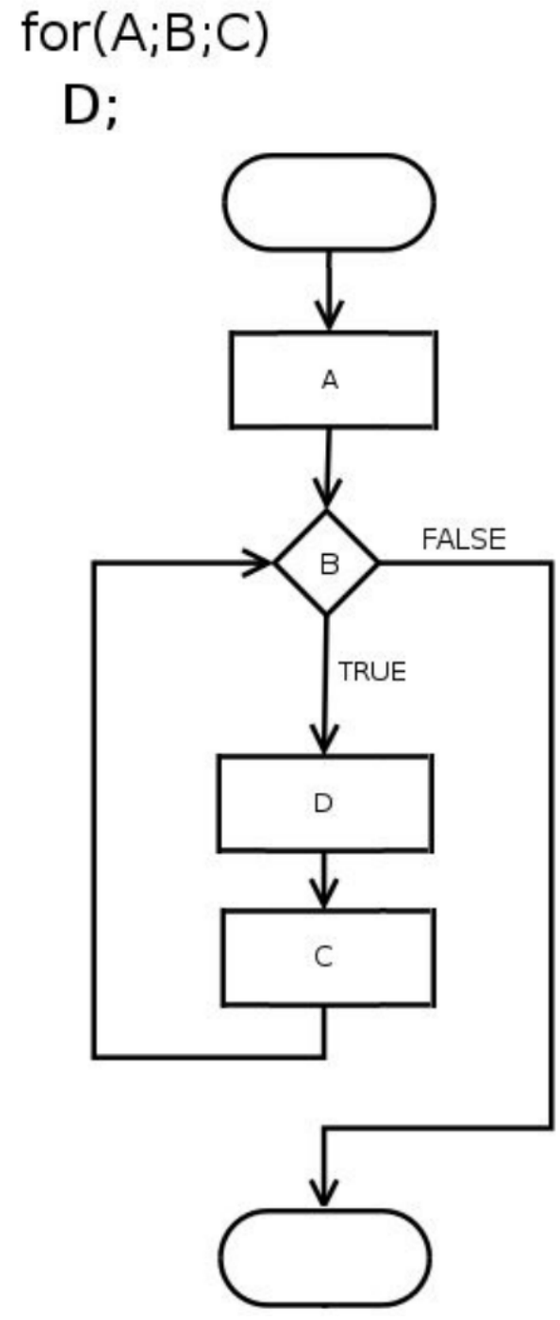
Credits: Prof. Kapravelos (ncsu)

Motivation

- Code injection
  - W^X
- Code reuse
  - ASLR
  - CFI

Program control flow

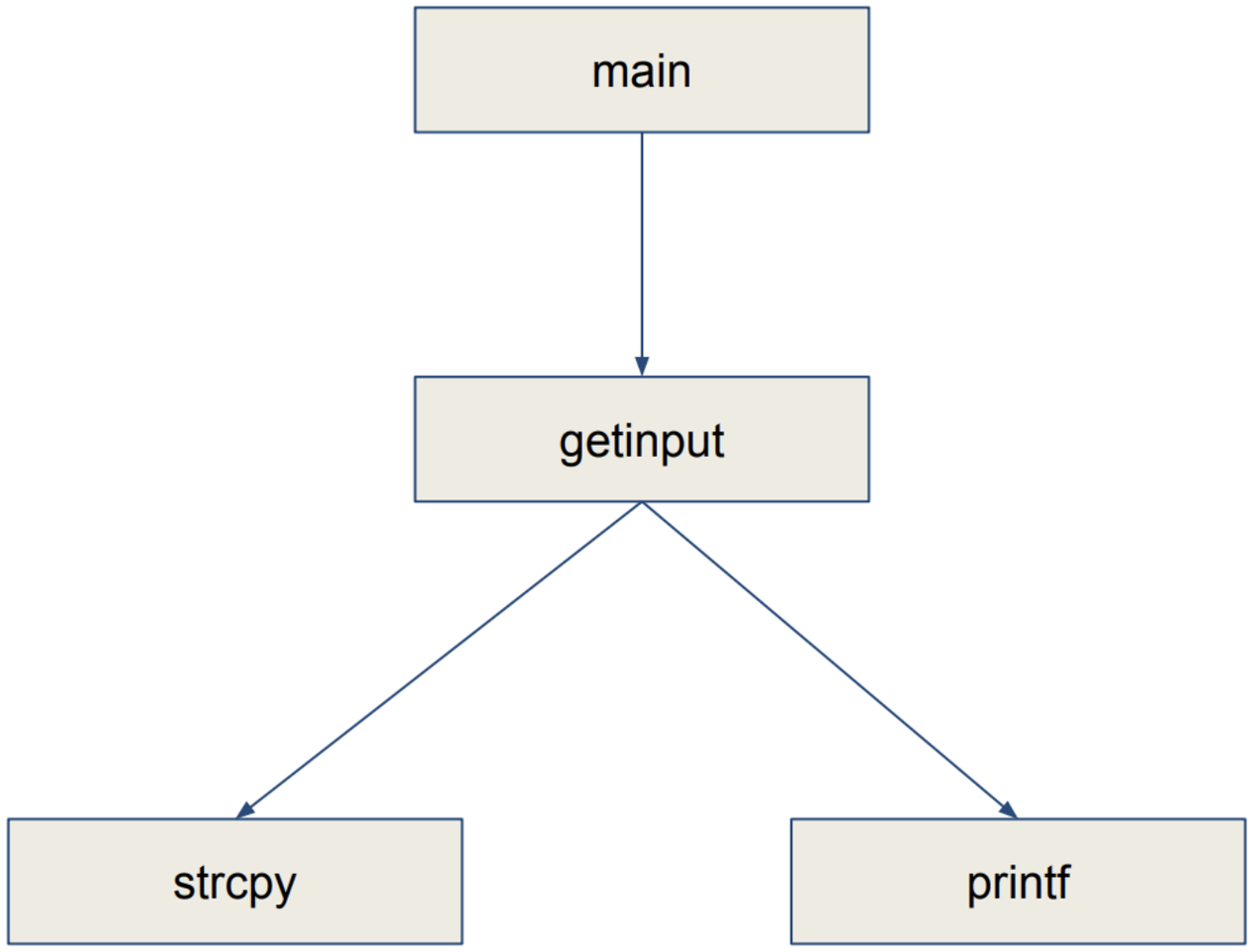
- Unconditional jumps
- Conditional jumps
- Loops
- Subroutines
- Unconditional halt



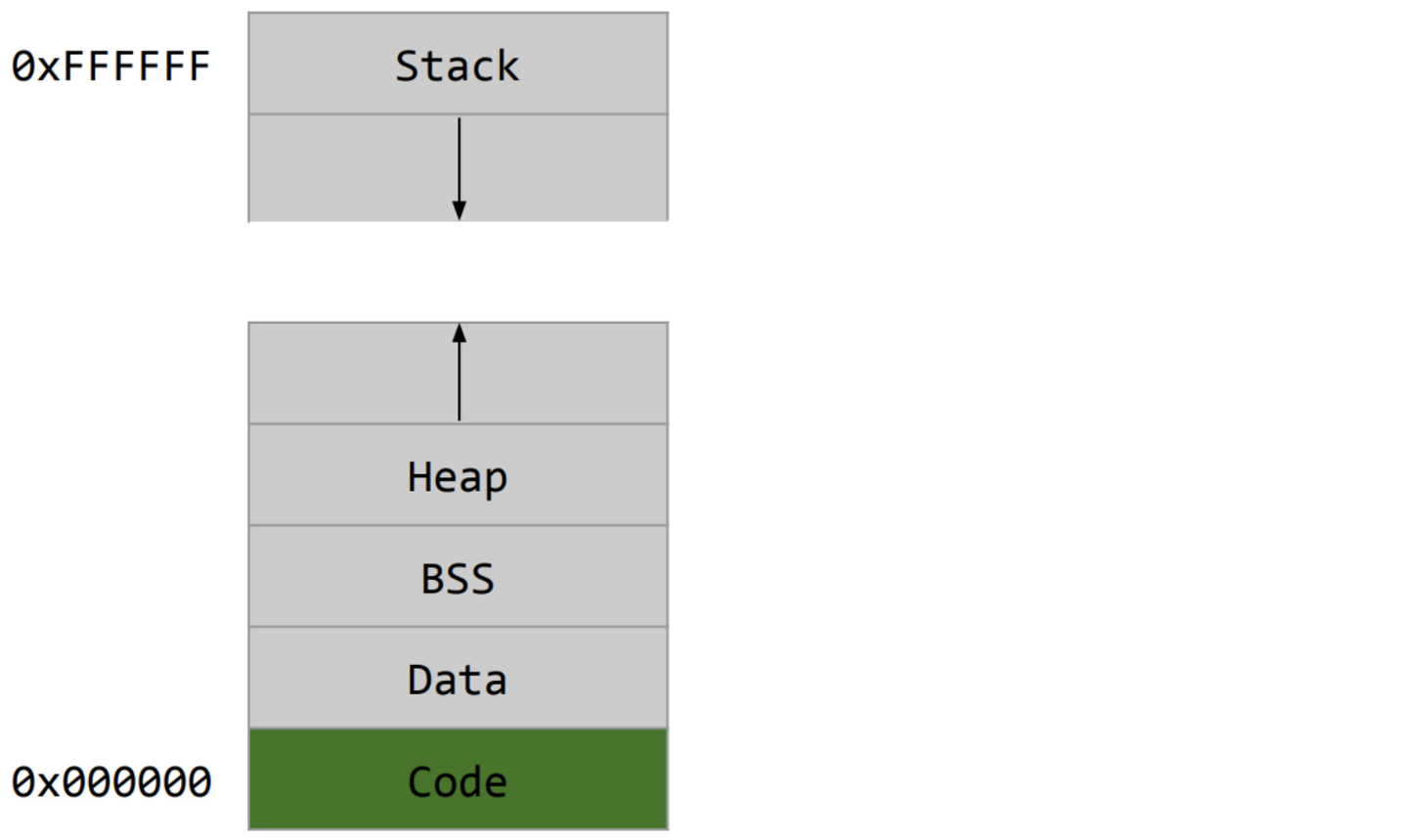
vuln.c

```c
#include <stdio.h>
#include <string.h>

void getinput(char *input) {
    char buffer[32];
    strcpy(buffer, input);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv) {
    getinput(argv[1]);
    return(0);
}
```

Simple call graph



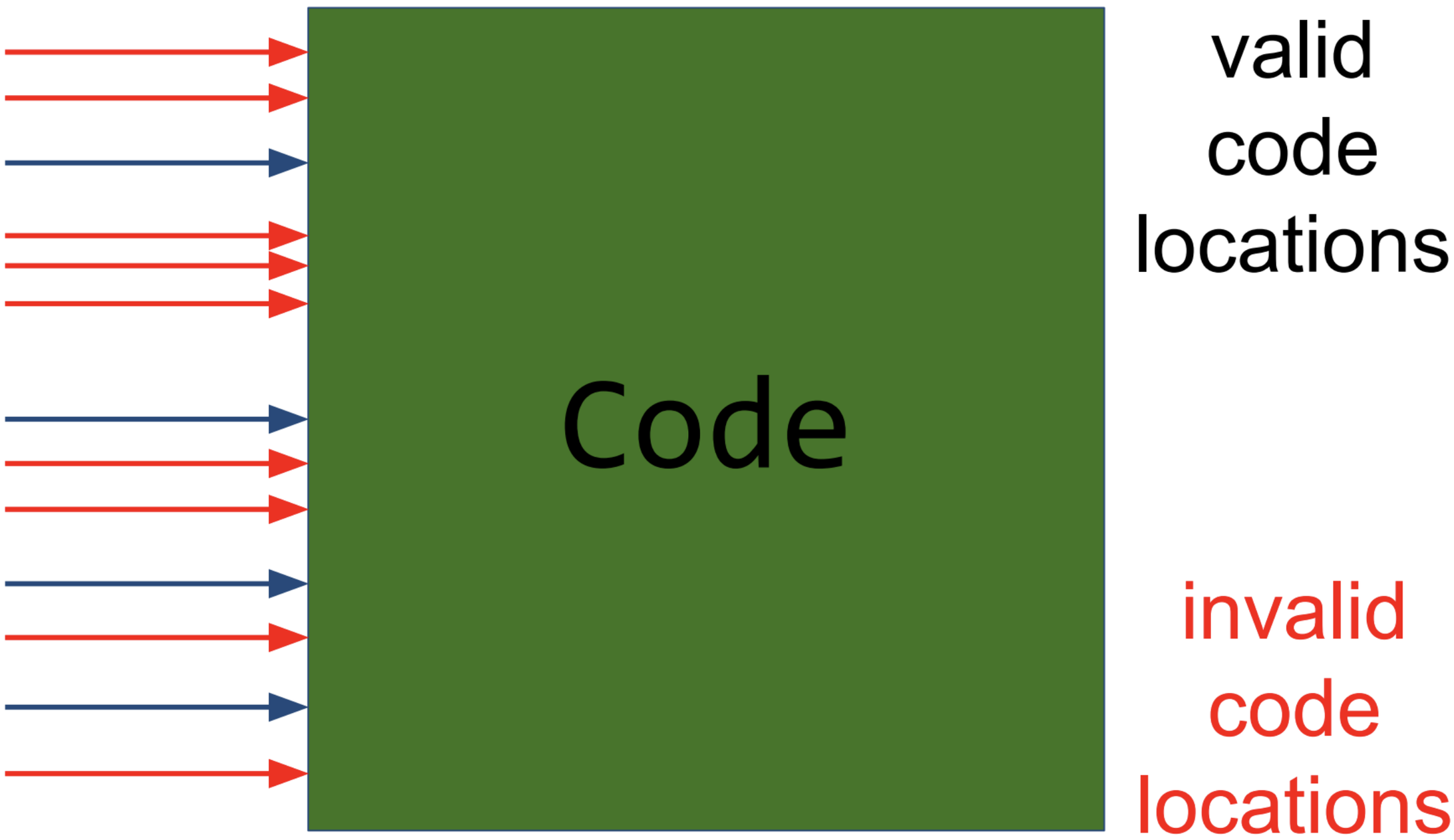
W^X



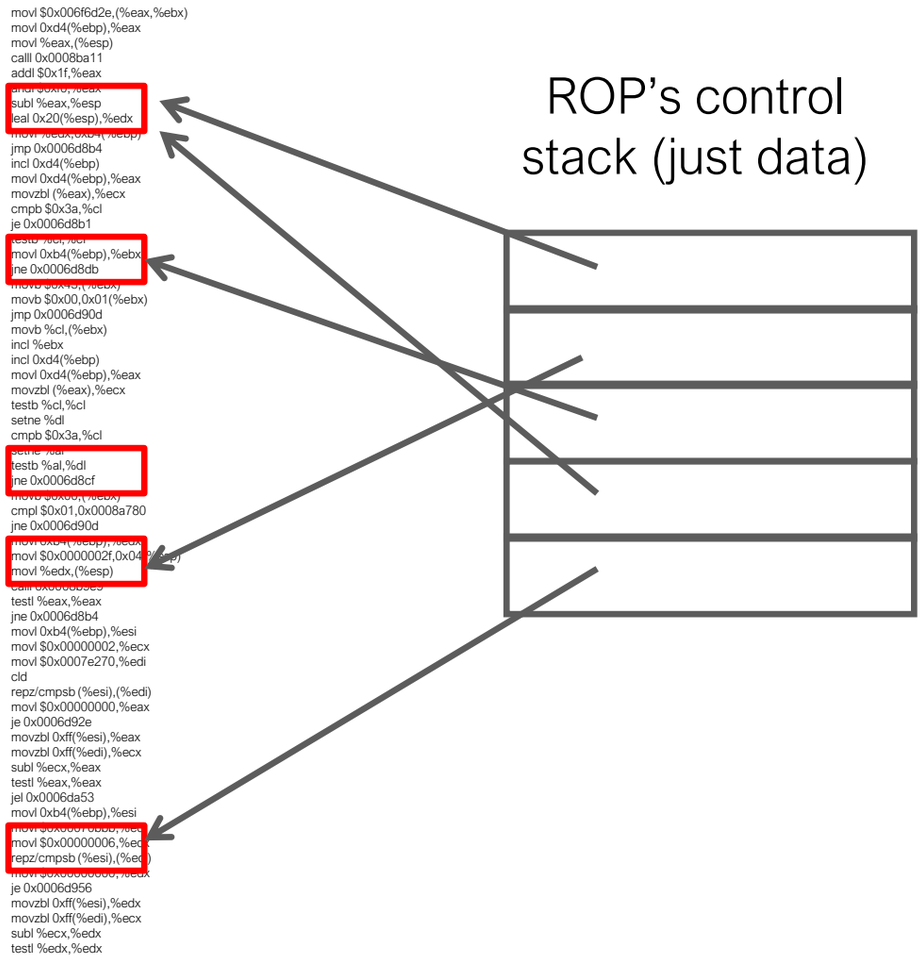
RW

RX

ROP



ROP



Fundamental problem with this execution model?

Code is not executed in the intended way!

How can we make sure that the program is executed in the intended way?
Control-Flow Integrity (CFI)

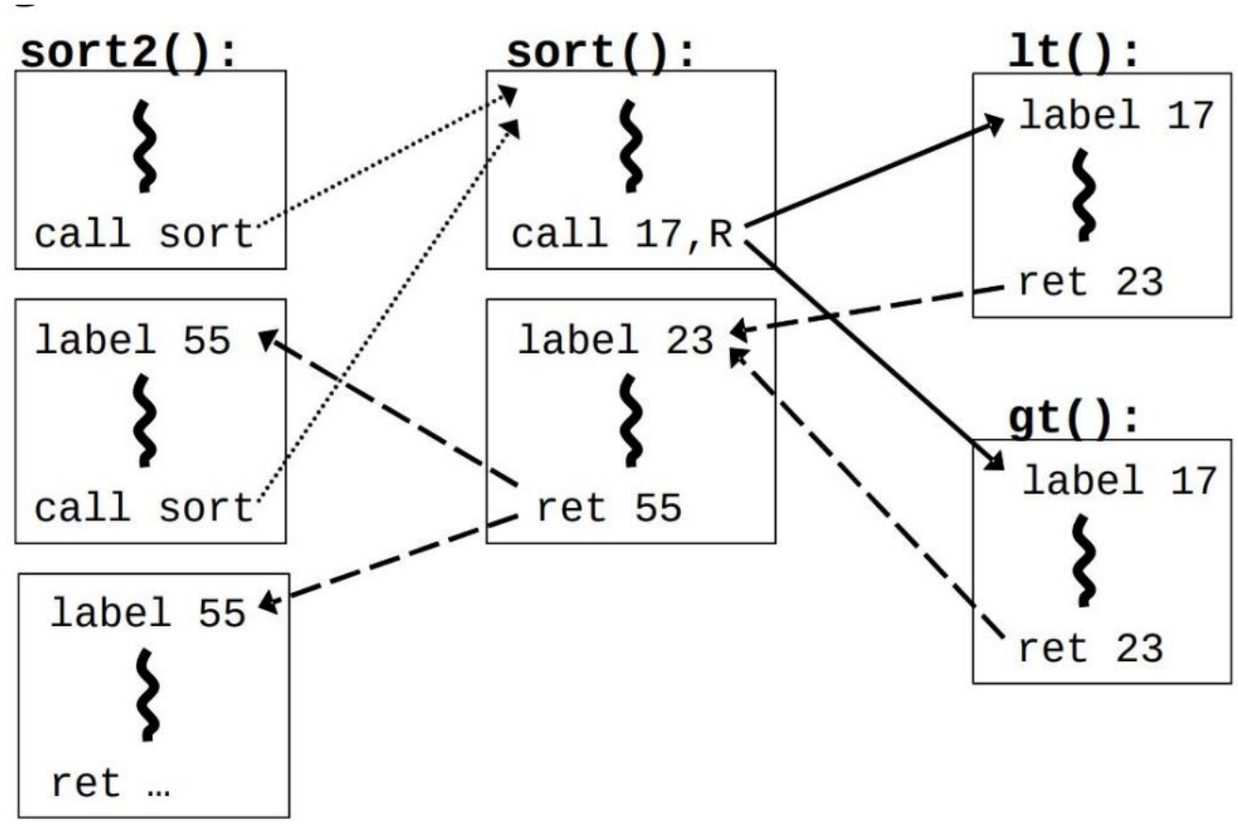
Control-Flow Integrity: CCS 05

Control-flow integrity

- CFI is a security policy
- Execution must follow a path of a Control-Flow Graph
- CFG can be pre-computed
  - source-code analysis
  - binary analysis
  - execution profiling
- Forward-edge and backward-edge
- But how can we enforce this extracted control-flow?

Enforcing CFI by Instrumentation

```
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 );
}
```



- LABEL ID
- CALL ID, DST
- RET ID

source: Control-Flow Integrity (link)

CFI Instrumentation Code

Source

Opcode bytes	Instructions		
FF E1	jmp	ecx	; computed jump

Destination

Opcode bytes	Instructions		
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
75 13	jne	error_label	; if != fail
8D 49 04	lea	ecx, [ecx+4]	; skip ID at dst
FF E1	jmp	ecx	; jump to dst

78 56 34 12	; data 12345678h		; ID
8B 44 24 04	mov	eax, [esp+4]	; dst
...			

or, alternatively, instrumented as (b):

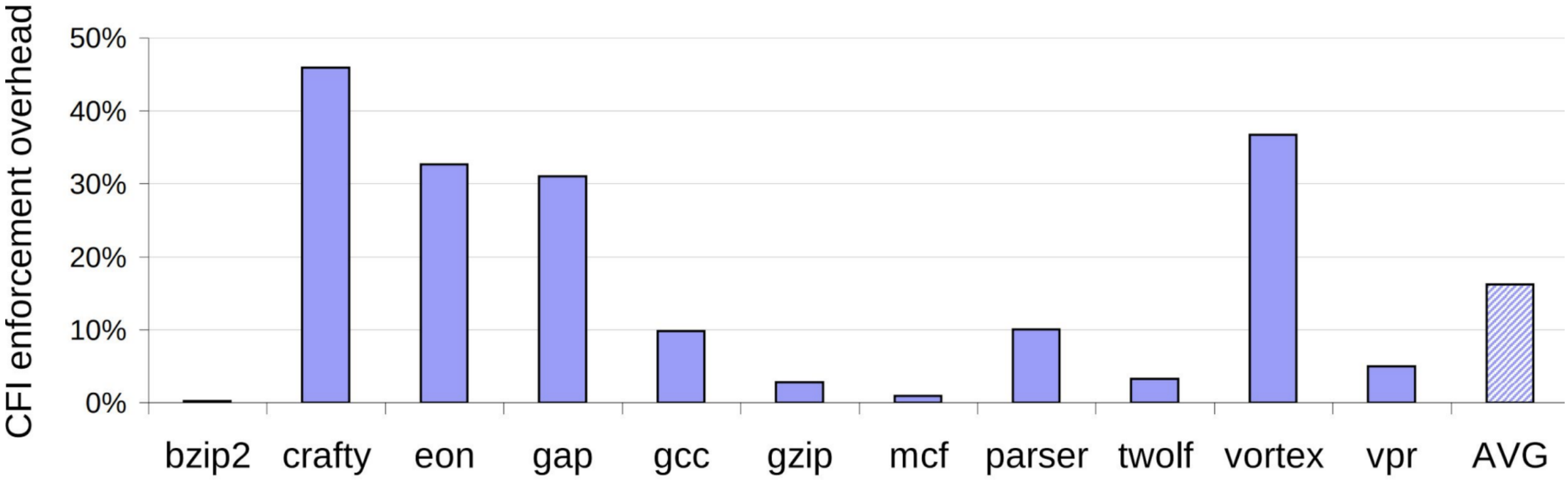
B8 77 56 34 12	mov	eax, 12345677h	; load ID-1
40	inc	eax	; add 1 for ID
39 41 04	cmp	[ecx+4], eax	; compare w/dst
75 13	jne	error_label	; if != fail
FF E1	jmp	ecx	; jump to label

3E 0F 18 05	prefetchnta		; label
78 56 34 12	[12345678h]		; ID
8B 44 24 04	mov	eax, [esp+4]	; dst
...			

- The extra code checks that the destination code is the intended jump location

source: Control-Flow Integrity (link)

Overhead


CFI enforcement overhead
50%
40%
30%
20%
10%
0%
bzip2
crafty
eon
gap
gcc
gzip
mcf
parser
twolf
vortex
vpr
AVG

Limitation

- Overhead is high
- Precise CFG construction is hard (or even impossible)

CFI in real systems

- Control Flow Guard
- LLVM
- Hardware features
  - Shadow stack
  - IBT: indirect branch tracking

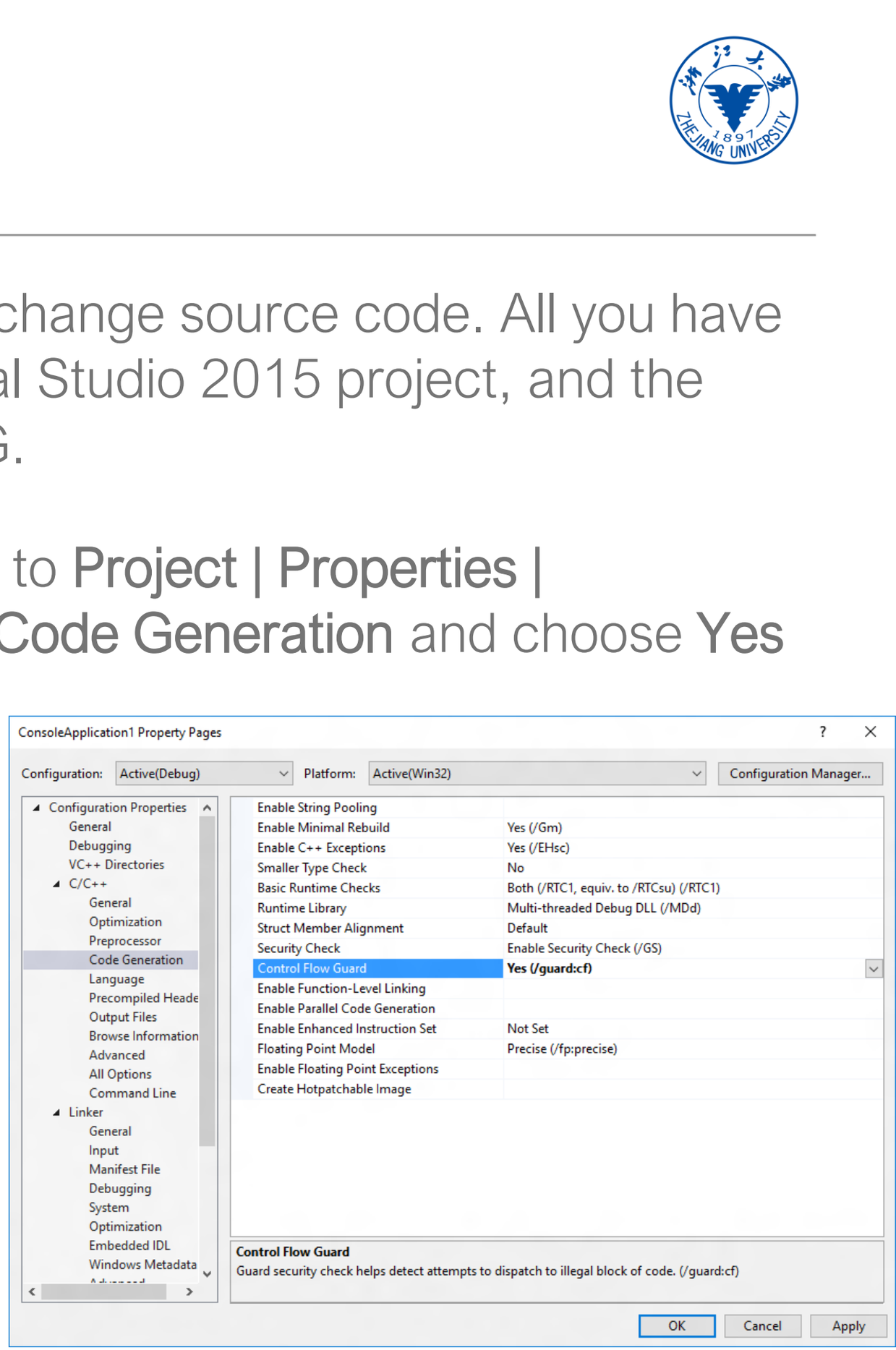
Control Flow Guard

- Windows 10 and Windows 8.1
- Microsoft Visual Studio 2015+
- Adds lightweight security checks to the compiled code
- Identifies the set of functions in the application that are valid targets for indirect calls
- The runtime support, provided by the Windows kernel:
  - Efficiently maintains state that identifies valid indirect call targets
  - Implements the logic that verifies that an indirect call target is valid

Control Flow Guard

- In most cases, there is no need to change source code. All you have to do is add an option to your Visual Studio 2015 project, and the compiler and linker will enable CFG.

- The simplest method is to navigate to **Project | Properties | Configuration Properties | C/C++ | Code Generation** and choose **Yes (/guard:cf)** for Control Flow Guard.



source: Control Flow Guard (link)

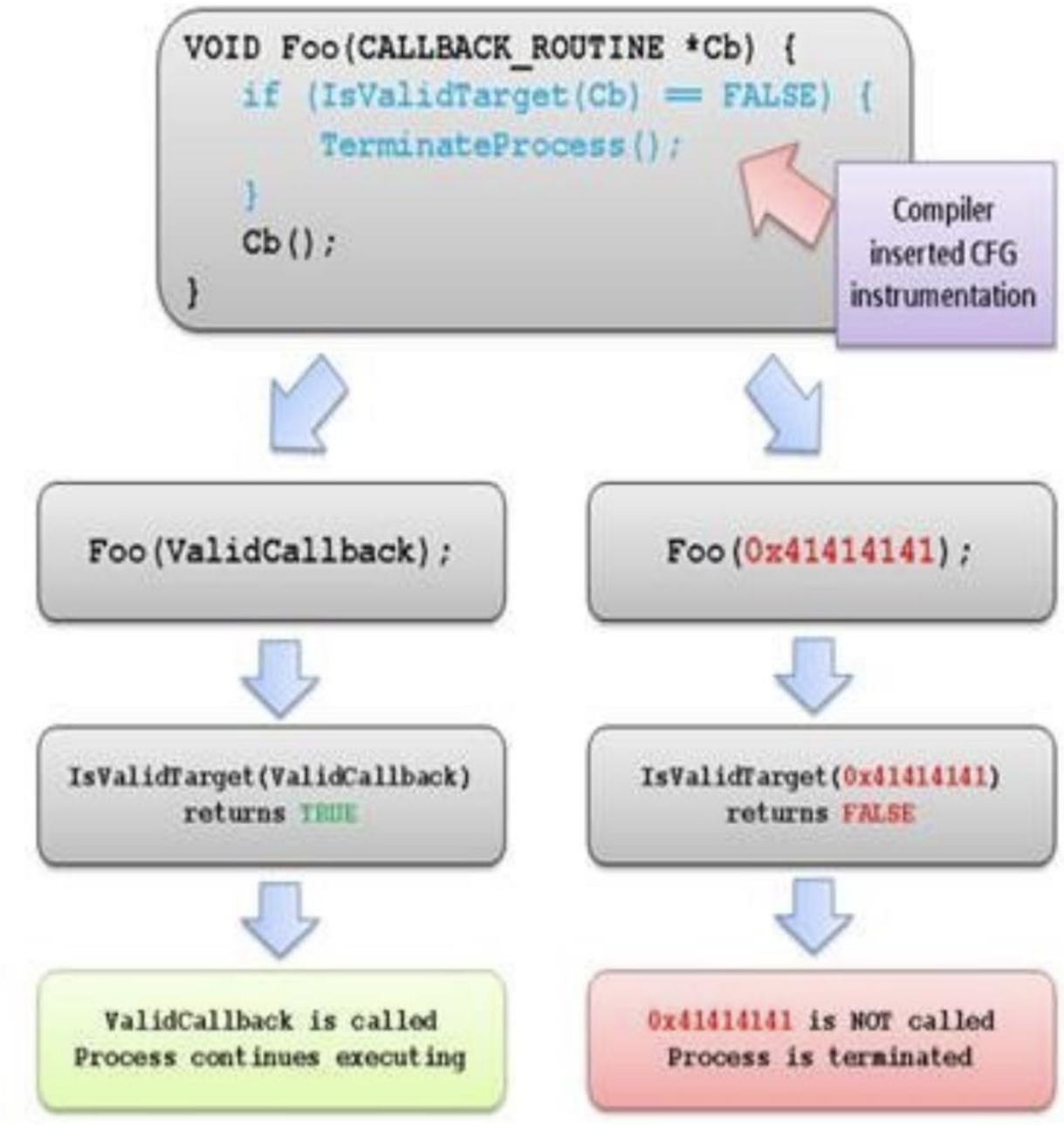
How Do I Tell That a Binary is under Control Flow Guard?

- Run the dumpbin tool (included in the Visual Studio 2015 installation) from the Visual Studio command prompt with the */headers* and */loadconfig* options: **dumpbin /headers.**

- "CF Instrumented" and "FID table present".

```
Section contains the following load config:

          000000A0 size
                 0 time date stamp
              0.00 Version
                 0 GlobalFlags Clear
                 0 GlobalFlags Set
                 0 Critical Section Default Timeout
                 0 Decommit Free Block Threshold
                 0 Decommit Total Free Threshold
   0000000000000000 Lock Prefix Table
                 0 Maximum Allocation Size
                 0 Virtual Memory Threshold
                 0 Process Heap Flags
                 0 Process Affinity Mask
                 0 CSD Version
              0000 Reserved
   0000000000000000 Edit list
   000000014023C008 Security Cookie
   000000001401C1A0 Guard CF address of check-function pointer
   000000001401C1A8 Guard CF address of dispatch-function pointer
   000000001401C2A8 Guard CF function table
               E95 Guard CF function count
          00003500 Guard Flags
                   CF Instrumented
                   FID table present
                   Protect delayload IAT
                   Delayload IAT in its own section
```

source: Control Flow Guard (link)



source: Control Flow Guard (link)

LLVM-CFI

- Clang includes an implementation of a number of control flow integrity (CFI) schemes
- Relies on link-time optimization: LTO
- LLVM-CFI implements the CFI check as a range check that maps into a table of trampolines. All **address taken functions** of the **same type are translated into a range where they are placed next to each other**. A CFI dispatch is then matched into a jump into an 8-byte aligned jump into the area of targets for the corresponding type.

```
void (*func)();

// func either points to bar or baz
if (usr == MAGIC)
  func = bar;
else
  func = baz;
```

Indirect branch tracking

- ENDBRANCH -> new CPU instruction
- marks valid indirect call/jmp targets in the program
- the CPU implements a state machine that tracks indirect jmp and call instructions
- when one of these instructions is seen, the state machine moves from IDLE to WAIT_FOR_ENDBRANCH state
- if an ENDBRANCH is not seen the processor causes a control protection fault

Challenges

- CFG construction
- A CFG is a graph that covers all valid executions of the program. Nodes in the graph are locations of control-flow transfers in the program and edges encode reachable targets.
- For forward edges, the CFG generation enumerates all possible targets, often leveraging information from the underlying source language.

Indirect functions

- For indirect function calls through a function pointer, the underlying analysis becomes more complicated as the target may not be known a-priori.

- Common source-based analyses use a type-based approach and, looking at the function prototype of the function pointer that is used, enumerate all matching functions.

  - Use address-taken to reduce valid set

  - For virtual calls, i.e., indirect calls in C++ that depend on the type of the object and the class relationship, the analysis can further leverage the type of the object to restrict the valid functions

Indirect functions

- So far, the constructed CFG is **stateless**, i.e., the *context of the execution* is not considered and each control-flow transfer is independent of all others.

- This over-approximates the number of valid targets with different granularities – since at runtime only one target is allowed for any possible transfer

Adaptive Call-site Sensitive Control Flow Integrity

```
1  typedef int (*Handler)(char *);
2  int proceed(Handler handler, char *root_path)
3  {
4    ...
5    return handler(root_path);
6  }
7
8  void auth()
9  {
10   char *user_name;
11   char *passwd;
12   char id[80];
13   int attempt = 5;
14   Handler handler;
15
16   while (attempt > 0) {
17     handler = &on_failure;
18     username = passwd = null;
19
20     scanf("%ms", &username);
21     scanf("%ms", &password);
22
23     passwd = salt_passwd(username, passwd);
24     sprintf(id, "%s;%s", username, passwd);
25
26     if (is_admin(id)) {
27       handler = &on_admin;
28       proceed(handler, user_home_dir);
29     } else {
30       proceed(handler, "/tmp");
31     }
32     attempt--;
33
34     //clear passwd, free user_name, passwd
35     ...
36   }
37 }
```

CFI-LB

- CFI-LB enforces a call-site sensitive CFI policy in which the targets of an indirect branch are validated in the context of callsites (i.e., return addresses on the stack).

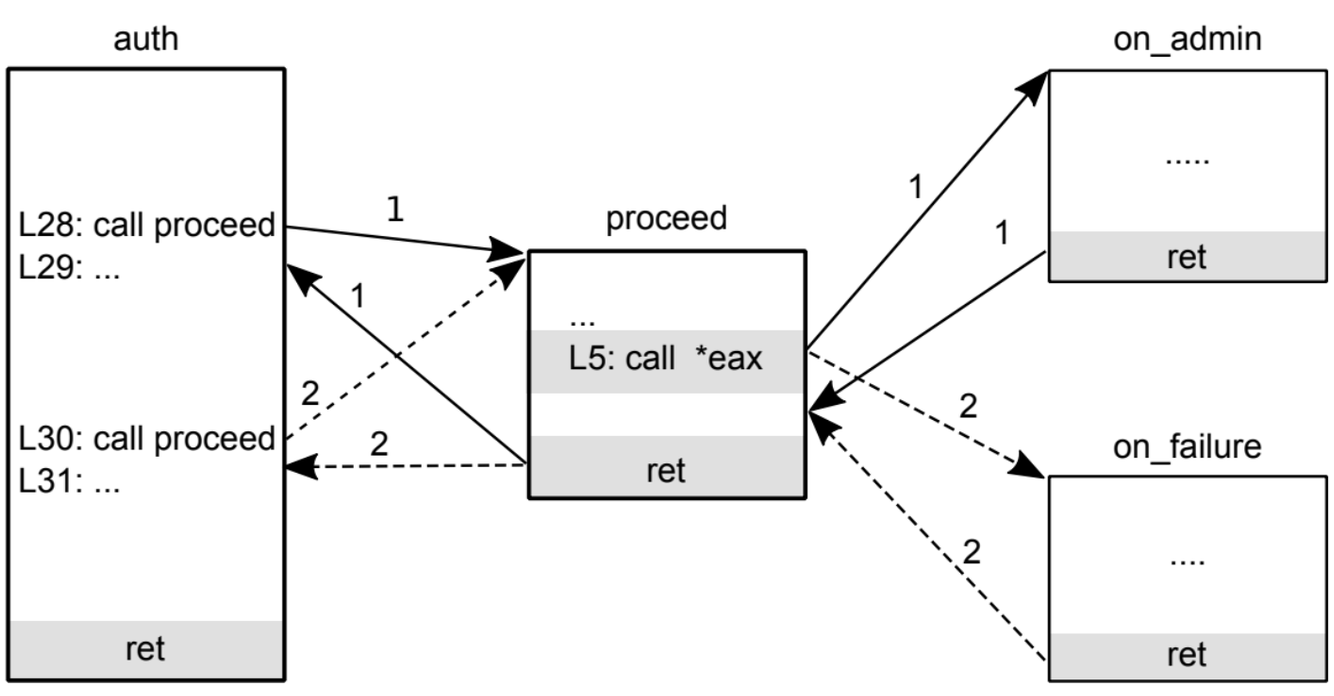


Fig. 3. CFG for the example in Fig. 2

Discussion

- Shared library
- Binary only
- Hardware assisted