Control-Flow Integrity

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: Prof. Kapravelos (ncsu)

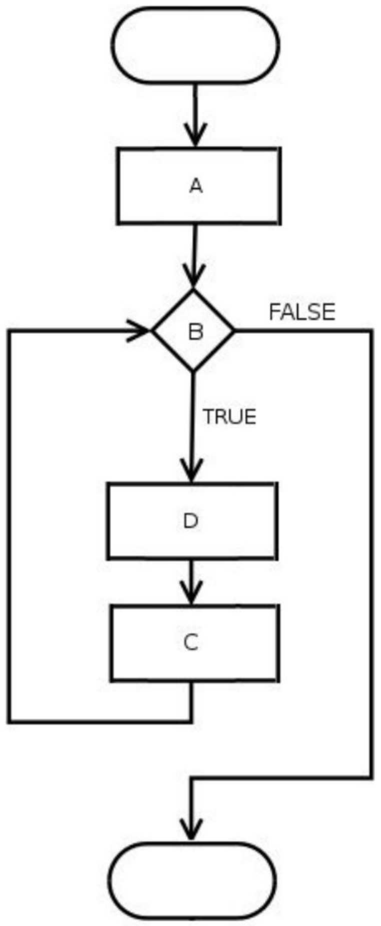
Motivation

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

Program control flow

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

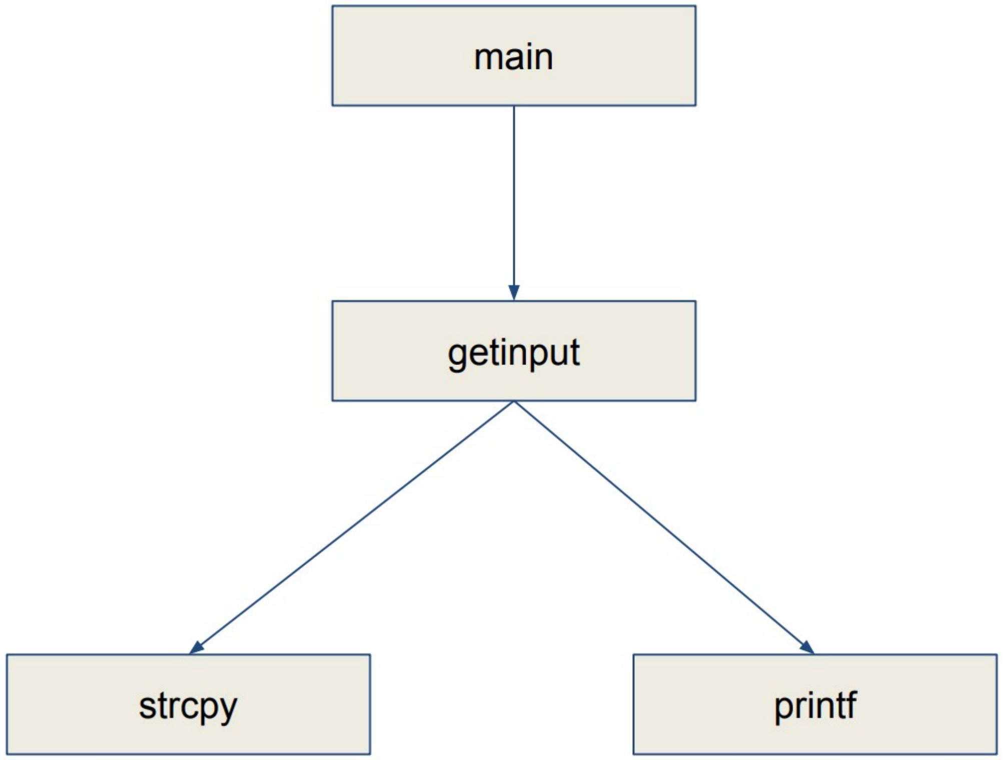
```
for(A;B;C)
  D;
```



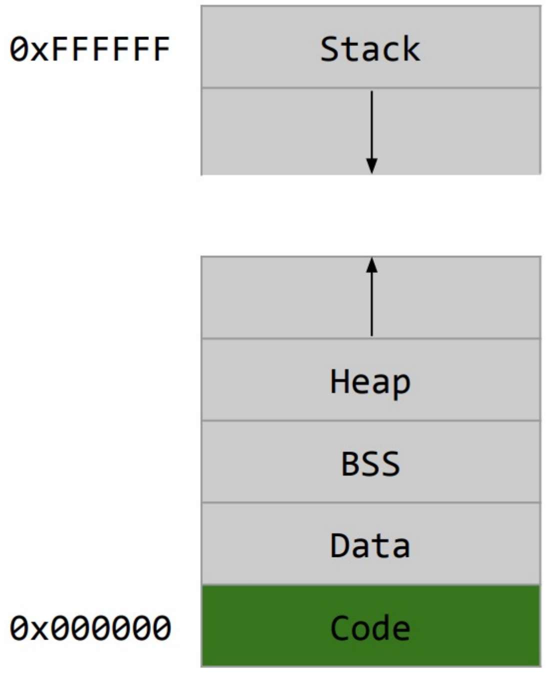
vuln.c

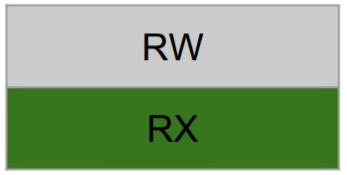
```c
#include <stdio.h>
#include <string.h>

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

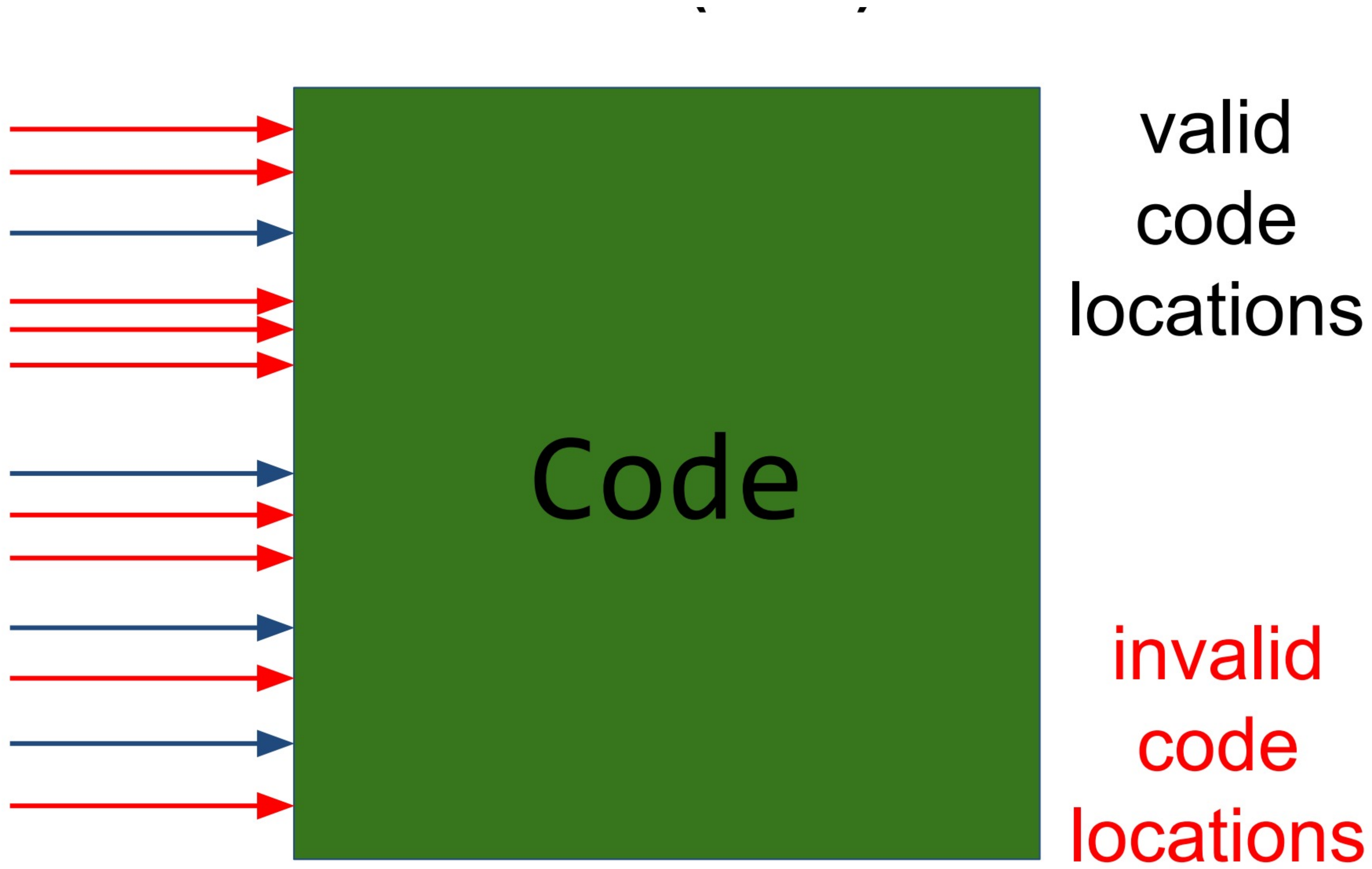

$$\begin{tikzcd}
 & \text{main} \arrow[d] & \\
 & \text{getinput} \arrow[dl] \arrow[dr] & \\
\text{strcpy} & & \text{printf}
\end{tikzcd}$$

W^X

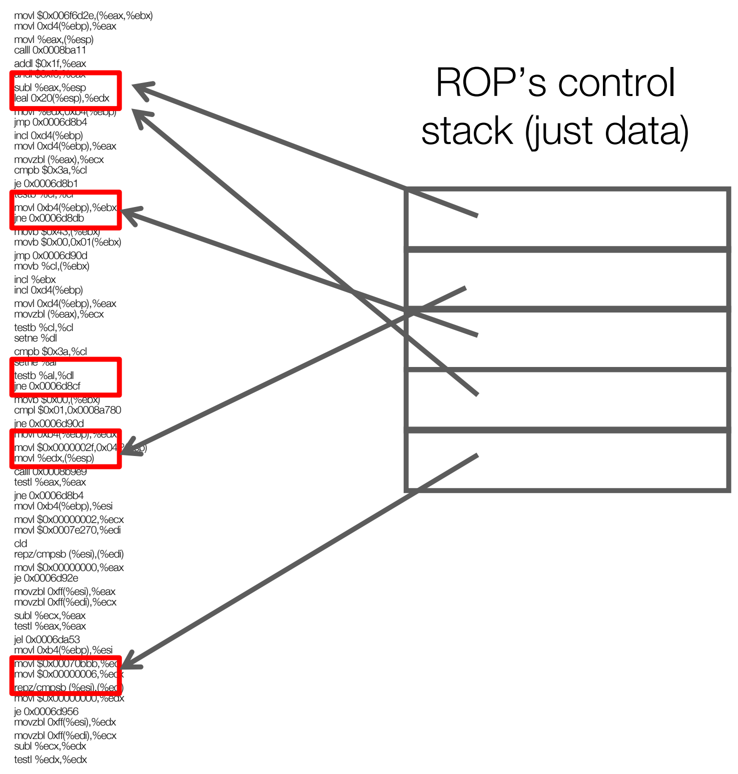




ROP



ROP



```
movl $0x006f6d2e,(%eax,%ebx)
movl 0xd4(%ebp),%eax
movl %eax,(%esp)
calll 0x0008ba11
addl $0x1f,%eax
andl $0xf0,%eax
subl %eax,%esp
leal 0x20(%esp),%edx
movl %edx,0xb4(%ebp)
jmp 0x0006d8b4
incl 0xd4(%ebp)
movl 0xd4(%ebp),%eax
movzbl (%eax),%ecx
cmpb $0x3a,%cl
je 0x0006d8b1
testb %cl,%cl
movl 0xb4(%ebp),%ebx
jne 0x0006d8db
movb $0x43,(%ebx)
movb $0x00,0x01(%ebx)
jmp 0x0006d90d
movb %cl,(%ebx)
incl %ebx
incl 0xd4(%ebp)
movl 0xd4(%ebp),%eax
movzbl (%eax),%ecx
testb %cl,%cl
setne %dl
cmpb $0x3a,%cl
setne %al
testb %al,%dl
jne 0x0006d8cf
movb $0x00,(%ebx)
cmpl $0x01,0x0008a780
jne 0x0006d90d
movl 0xb4(%ebp),%edx
movl $0x0000002f,0x04(%esp)
movl %edx,(%esp)
calll 0x0008b9e9
testl %eax,%eax
jne 0x0006d8b4
movl 0xb4(%ebp),%esi
movl $0x00000002,%ecx
movl $0x0007e270,%edi
cld
repz/cmpsb (%esi),(%edi)
movl $0x00000000,%eax
je 0x0006d92e
movzbl 0xff(%esi),%eax
movzbl 0xff(%edi),%ecx
subl %ecx,%eax
testl %eax,%eax
jel 0x0006da53
movl 0xb4(%ebp),%esi
movl $0x00070bbb,%ec
movl $0x00000006,%ecx
repz/cmpsb (%esi),(%edi)
movl $0x00000000,%edx
je 0x0006d956
movzbl 0xff(%esi),%edx
movzbl 0xff(%edi),%ecx
subl %ecx,%edx
testl %edx,%edx
```

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)

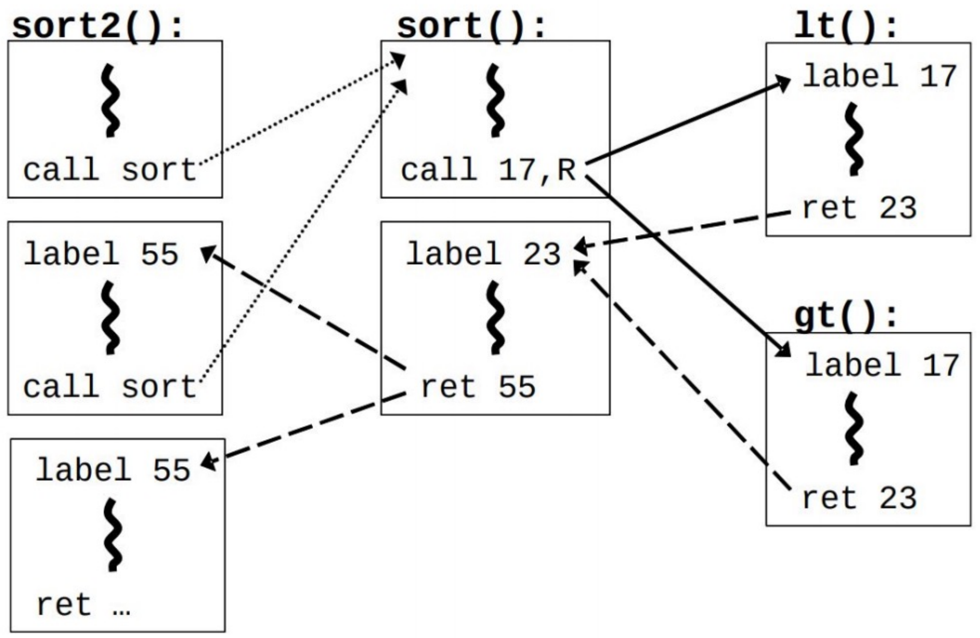
Control-Flow Integrity: CCS 05

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

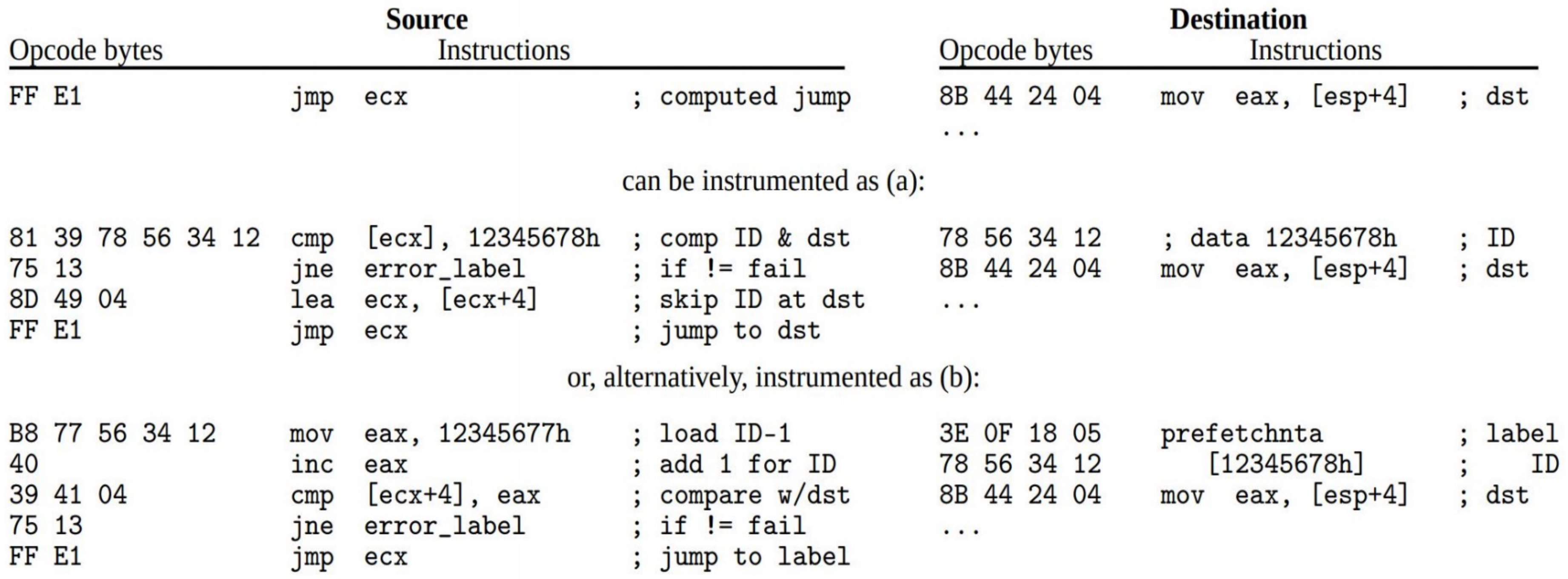
```
bool lt(int x, int y) {
    return x < y;
}

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

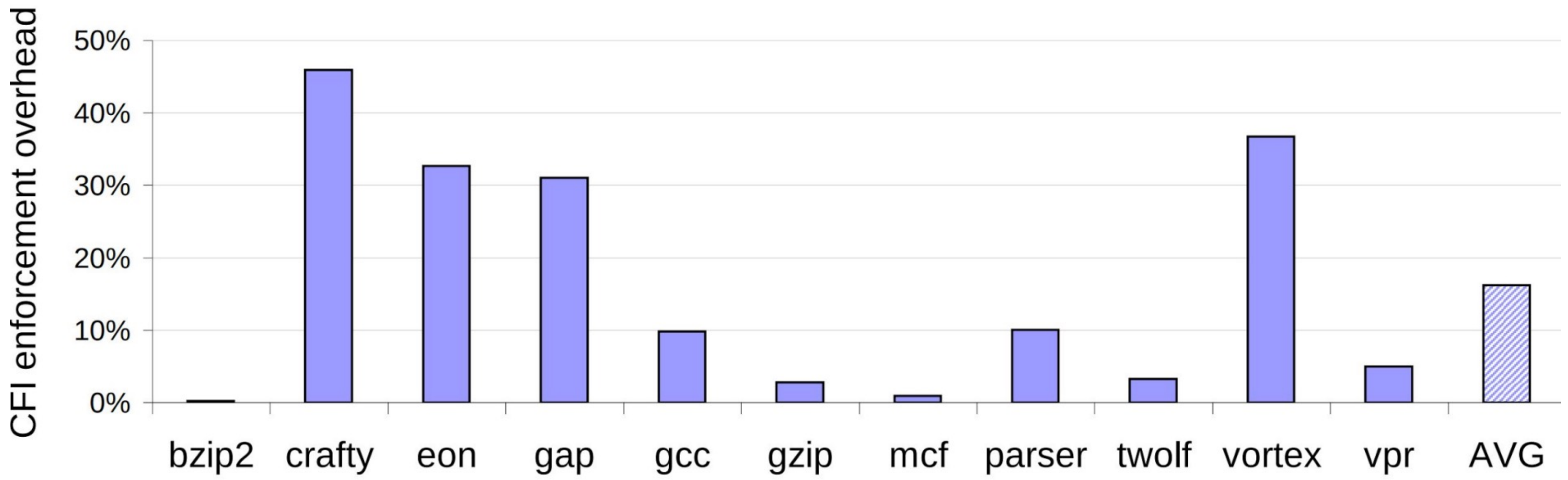


```
                          Source                                              Destination
Opcode bytes                   Instructions                     Opcode bytes          Instructions
FF E1               jmp  ecx               ; computed jump      8B 44 24 04   mov  eax, [esp+4]   ; dst
   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

- 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

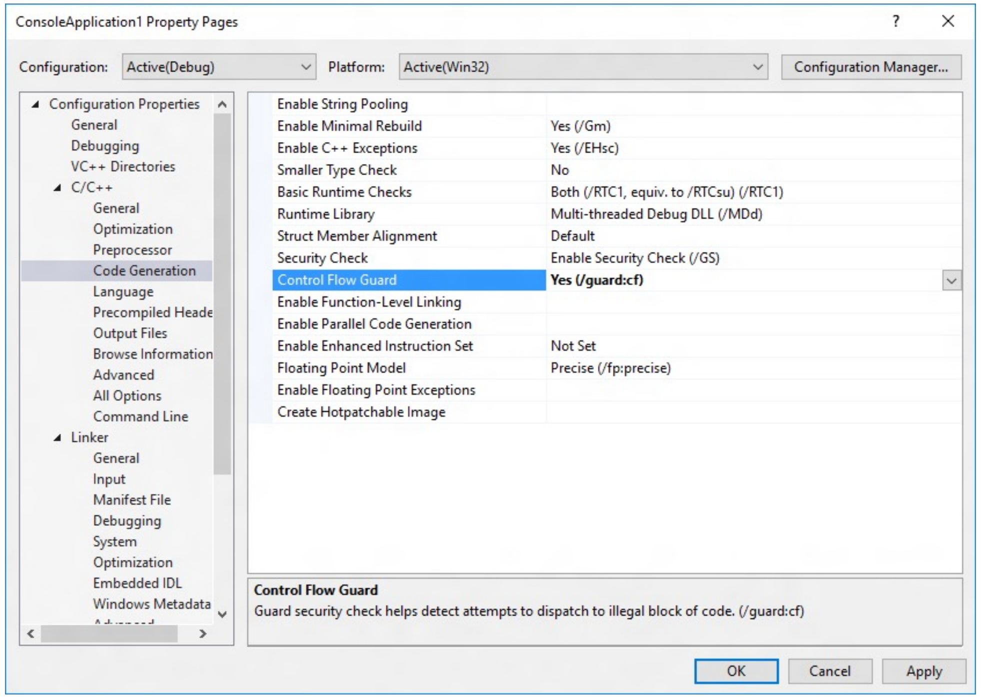
- Control Flow Guard
- LLVM
- Hardware features
  - Shadow stack
  - IBT: indirect branch tracking

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.


ConsoleApplication1 Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

Enable String Pooling	
Enable Minimal Rebuild	Yes (/Gm)
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Both (/RTC1, equiv. to /RTCsu) (/RTC1)
Runtime Library	Multi-threaded Debug DLL (/MDd)
Struct Member Alignment	Default
Security Check	Enable Security Check (/GS)
Control Flow Guard	**Yes (/guard:cf)**
Enable Function-Level Linking	
Enable Parallel Code Generation	
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	
Create Hotpatchable Image	

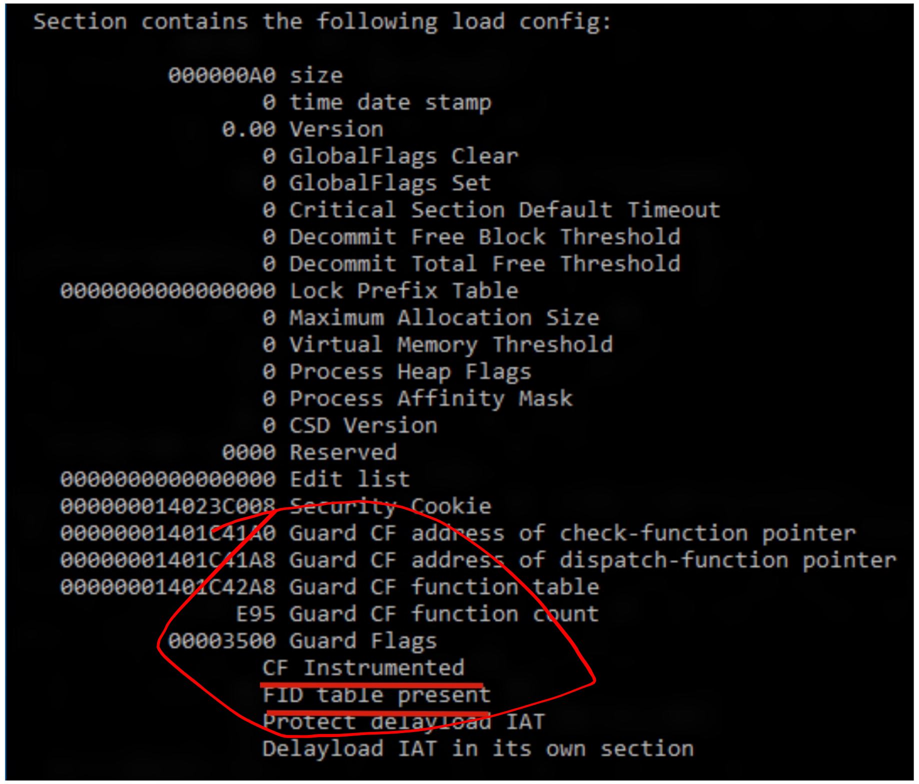
Control Flow Guard
Guard security check helps detect attempts to dispatch to illegal block of code. (/guard:cf)

source: Control Flow Guard (link)

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

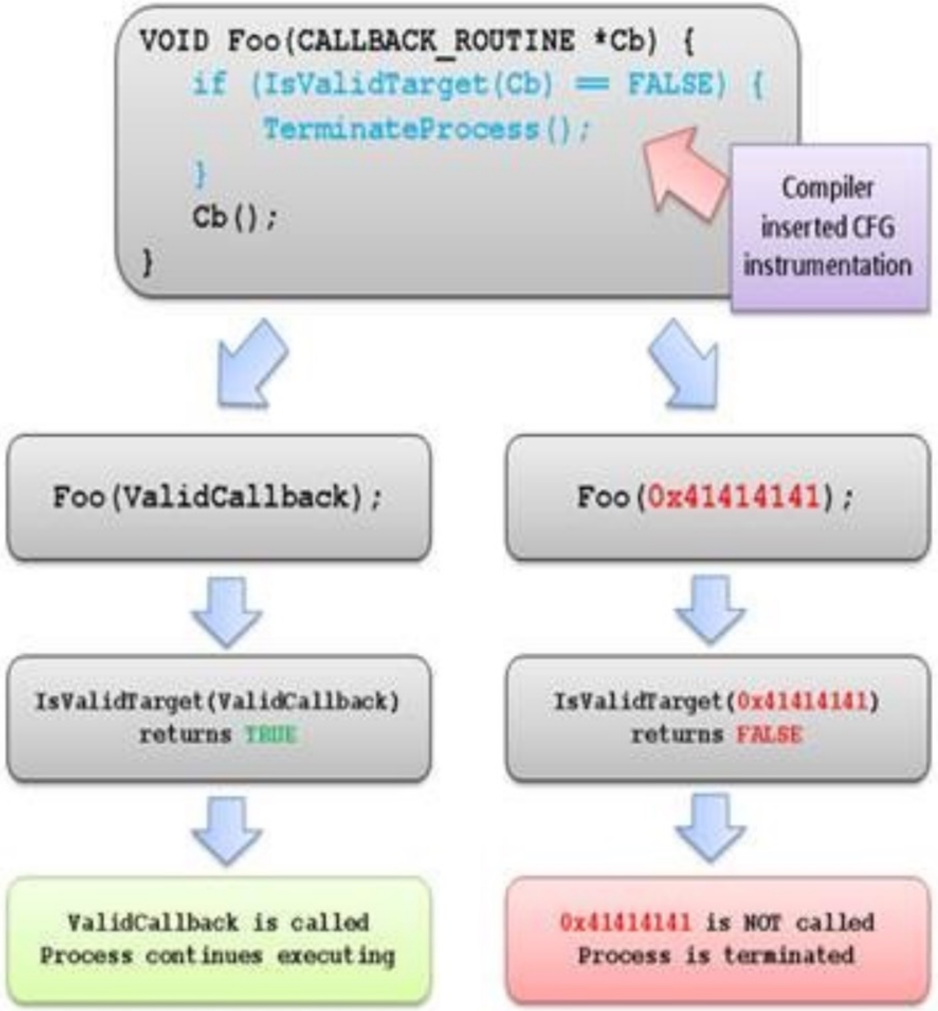
- Run the dumpbin tool (included in the Visual Studio 2015 installation) from the Visual Studio command prompt with the */headers* and */loadconfig* options: **dumpbin /headers.**

- "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
  00000001401C41A0 Guard CF address of check-function pointer
  00000001401C41A8 Guard CF address of dispatch-function pointer
  00000001401C42A8 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)



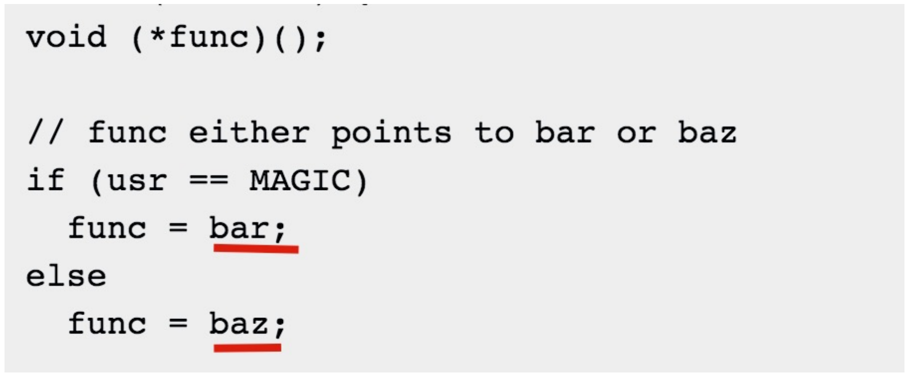
```
VOID Foo(CALLBACK_ROUTINE *Cb) {
    if (IsValidTarget(Cb) == FALSE) {
        TerminateProcess();
    }
    Cb();
}
```

Compiler inserted CFG instrumentation
Foo(ValidCallback);
Foo(0x41414141);
IsValidTarget(ValidCallback) returns TRUE
IsValidTarget(0x41414141) returns FALSE
ValidCallback is called Process continues executing
0x41414141 is NOT called Process is terminated

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;
```

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();

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

  // forward edge CFI check
  // depending on the precision of CFI:
  // a) all functions {bar, baz, buz, bez, foo} are allowed
  // b) all functions with prototype "void (*)()" are allowed,
  //    i.e., {bar, baz, buz}
  // c) only address taken functions are allowed, i.e., {bar, baz}
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

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

```c
typedef int (*Handler)(char *);
int proceed(Handler handler, char *root_path)
{
  ...
  return handler(root_path);
}

void auth()
{
  char *user_name;
  char *passwd;
  char id[80];
  int attempt = 5;
  Handler handler;

  while (attempt > 0) {
    handler = &on_failure;
    username = passwd = null;

    scanf("%ms", &username);
    scanf("%ms", &password);

    passwd = salt_passwd(username, passwd);
    sprintf(id, "%s;%s", username, passwd);

    if (is_admin(id)) {
      handler = &on_admin;
      proceed(handler, user_home_dir);
    } else {
      proceed(handler, "/tmp");
    }
    attempt--;

    //clear passwd, free user_name, passwd
    ...
  }
}
```

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).

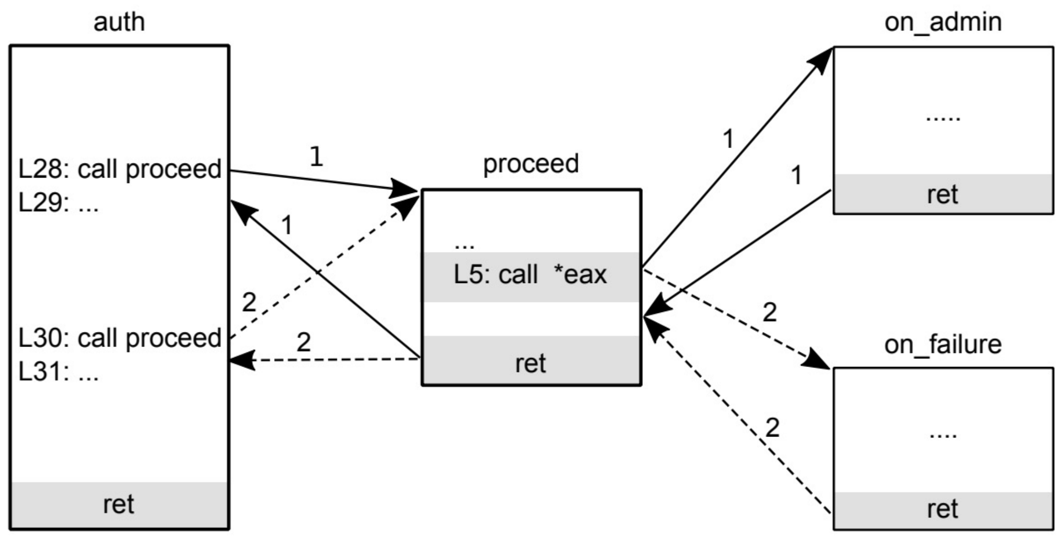


Fig. 3. CFG for the example in Fig. 2

Discussion

- Shared library
- Binary only
- Hardware assisted