

CSC 591

Systems Attacks and Defenses

Control-Flow Integrity

Alexandros Kapravelos

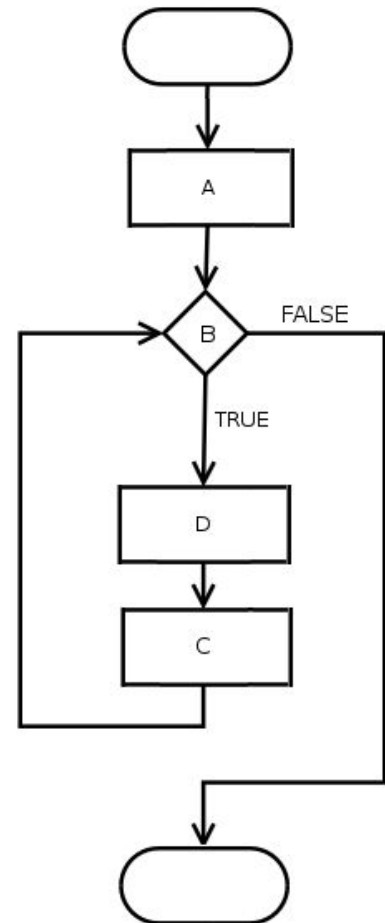
akaprav@ncsu.edu

ROP & return-to-libc reuse existing code instead of injecting malicious code. How can we stop this?

Program control flow

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

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



vuln.c

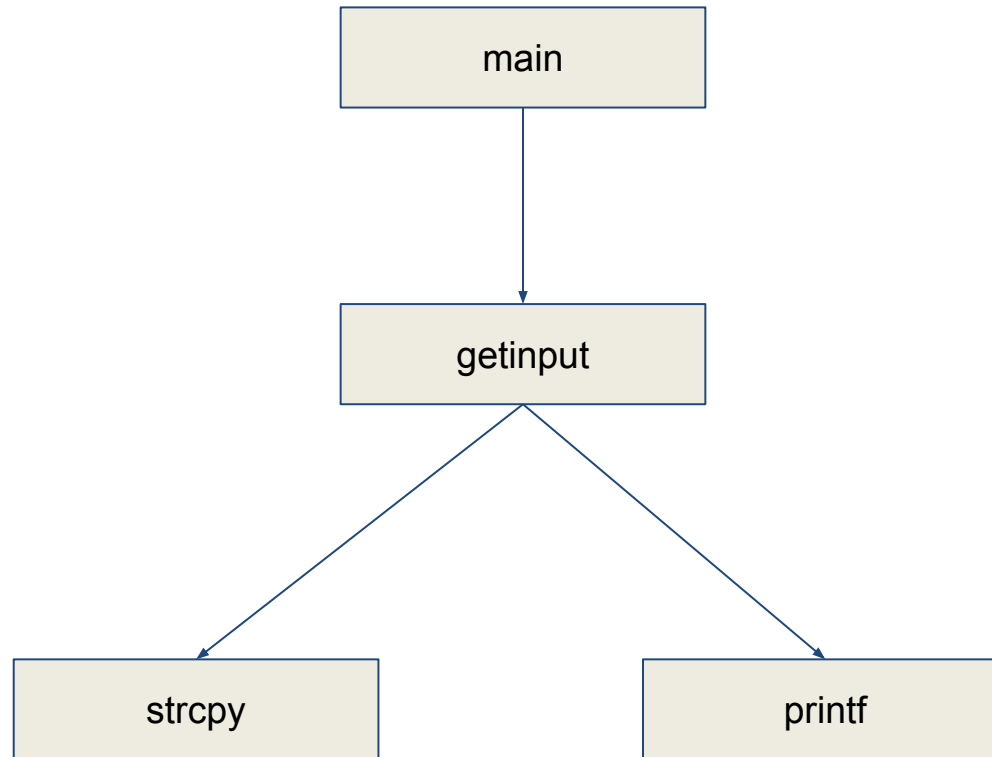
```
#include <stdio.h>
#include <string.h>

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



Functions locations

```
$ gcc vuln.c -o vuln
```

```
$ radare2 -A ./vuln
```

```
[0x004004e0]> afl
```

```
0x004004e0 42    1  sym._start
```

```
0x004004c0 6     1  sym.imp.__libc_start_main
```

```
0x00400631 41    1  sym.main
```

```
0x004005d6 91    3  sym.getinput
```

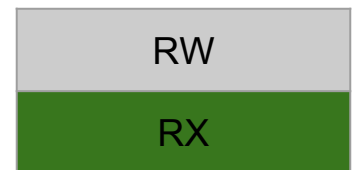
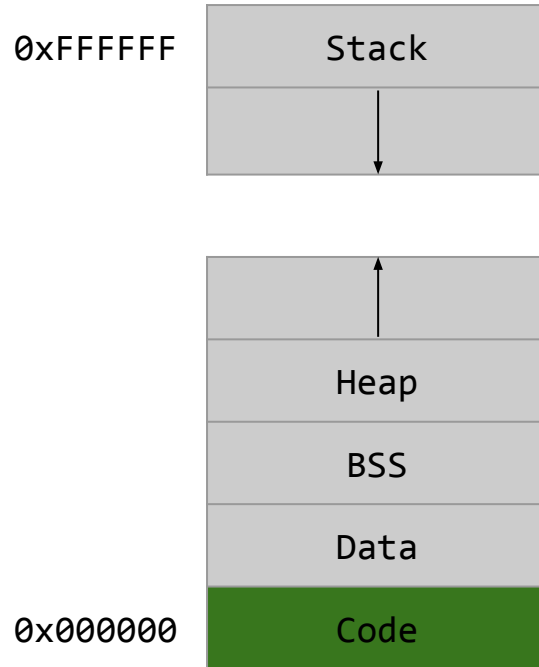
```
0x00400490 6     1  sym.imp.strcpy
```

```
0x004004b0 6     1  sym.imp.printf
```

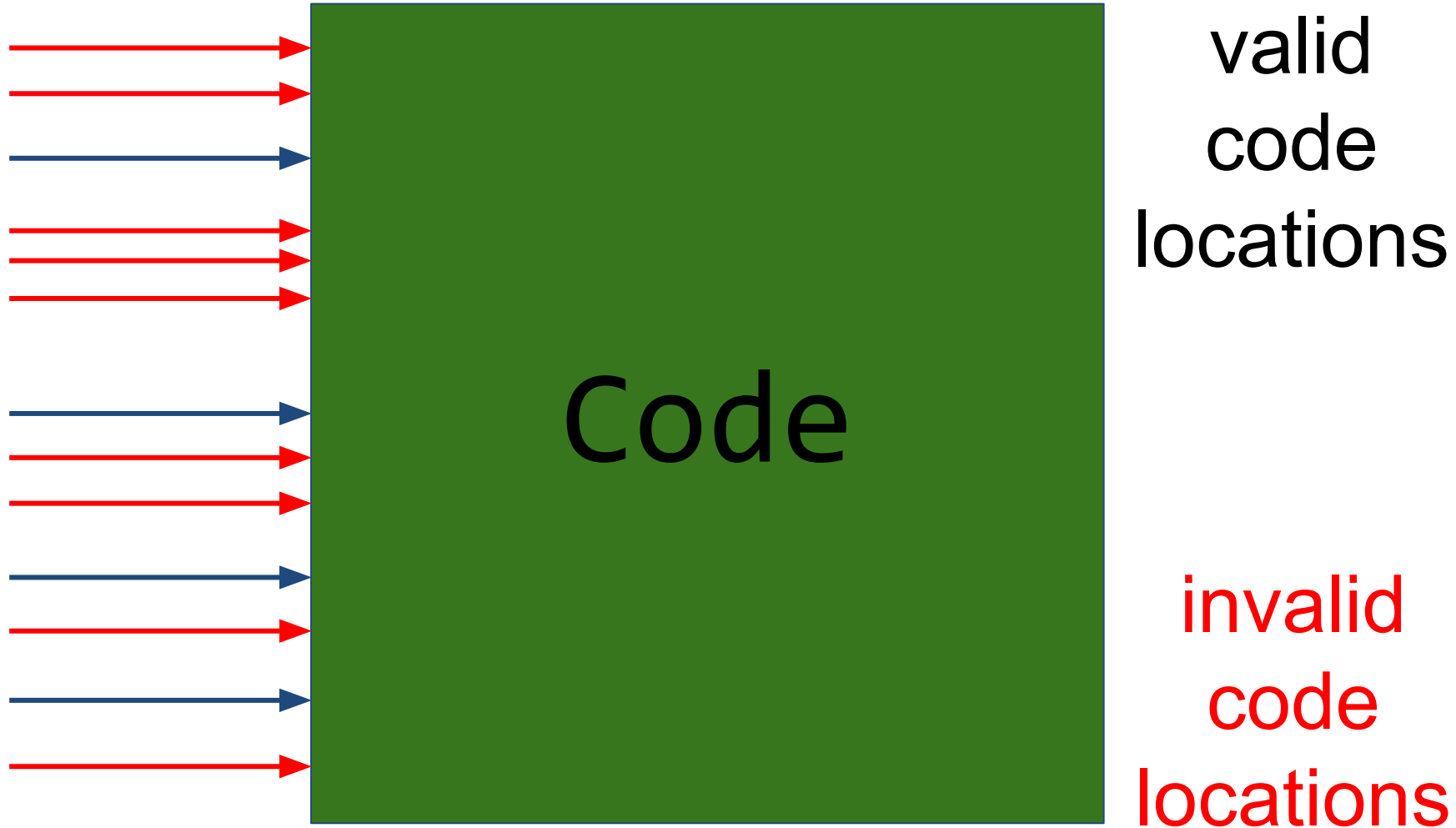
```
0x004004a0 6     1  sym.imp.__stack_chk_fail
```

```
[0x004004e0]>
```

NOEXEC (W^X)



NOEXEC (W^X)



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

- 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

- But how can we enforce this extracted control-flow?

Enforcing CFI by Instrumentation

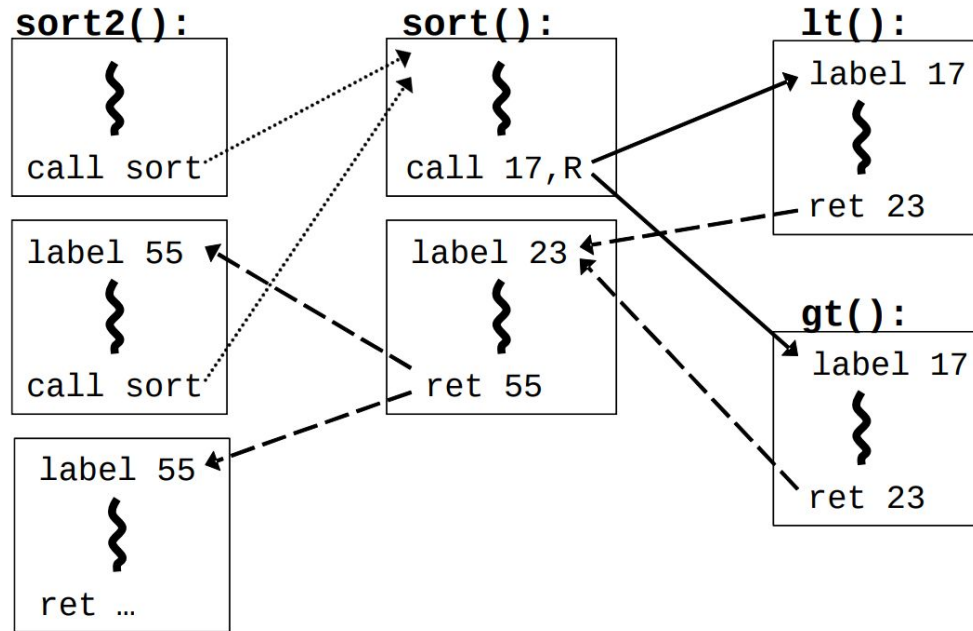
```

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

CFI Instrumentation Code

Opcode bytes	Source Instructions	Opcode bytes	Destination 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

CFI assumptions

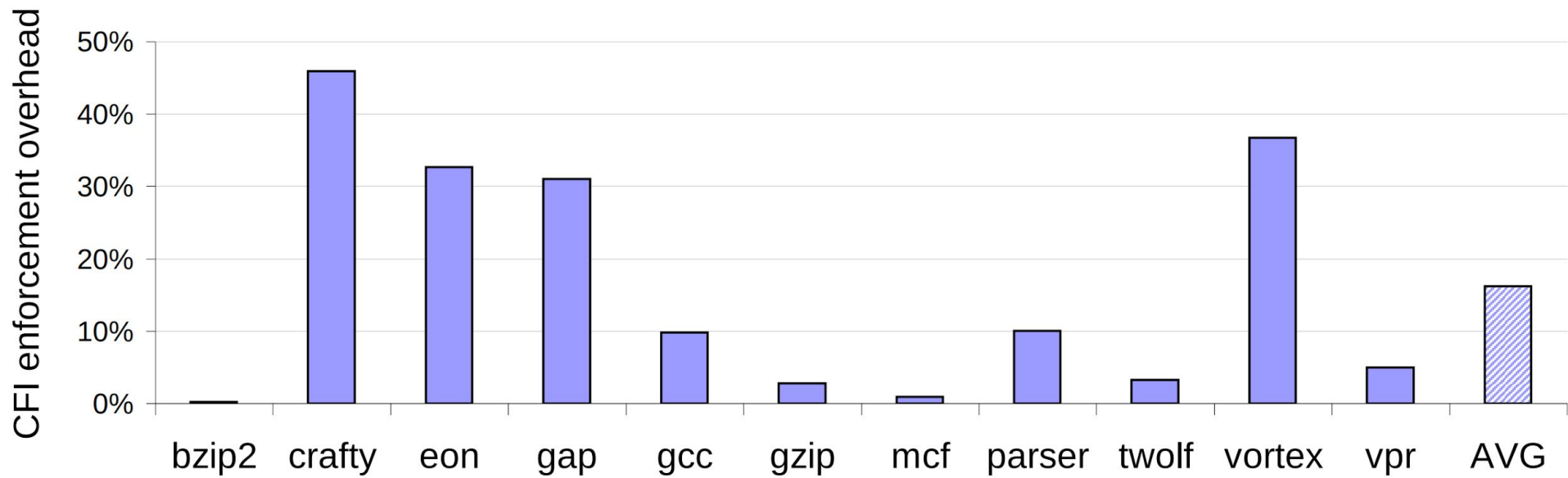
- Unique IDs
- Non-writable Code (NWC)
- Non-executable Data (NXD)
- Jumps cannot go into the middle of instructions

Attacker

- Powerful attacker model
 - Arbitrary control of all data in memory
 - Even hijack the execution flow of the program

- With CFI, execution will always follow the CFG

Overhead



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 enforcement technology

- Shadow stack
 - CALL instruction pushes the return address on both the data and shadow stack
 - RET instruction pops the return address from both stacks and compares them
 - if the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP)
- 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



Limitations of CFI?

Fine-grained CFI

- Precise monitoring of indirect control-flow changes
- caller-callee must match
- High performance overhead (~21%)
- Highest security

Coarse-grained CFI

- Trades security for better performance
- Any valid call location is accepted

[1] N. Carlini and D. Wagner, “ROP is still dangerous: Breaking modern defenses”

[2] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, “Stitching the gadgets: On the ineffectiveness of coarse grained control-flow integrity protection”

[3] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, “Out of control: Overcoming control-flow integrity”

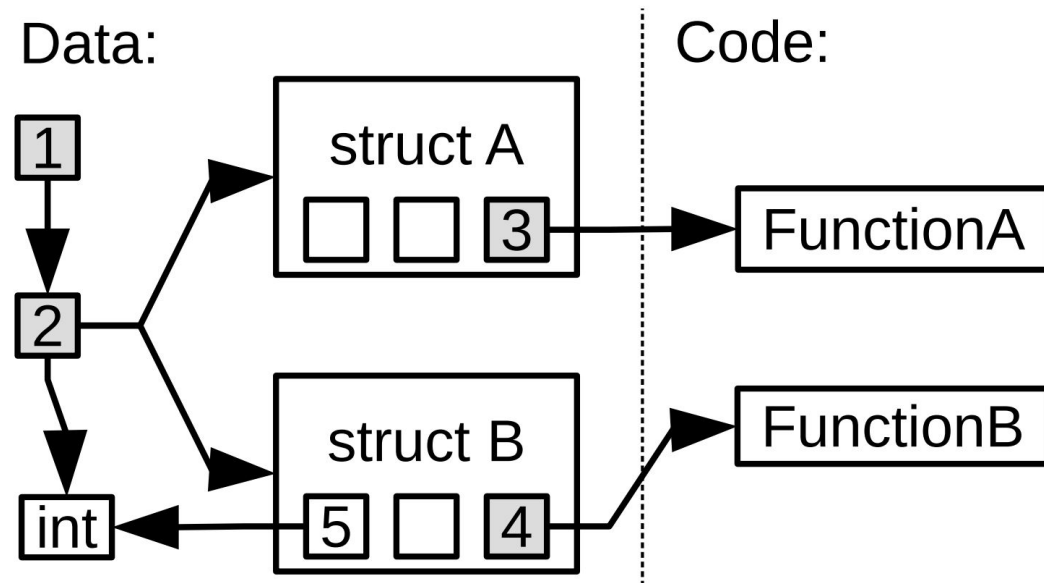
[4] E. Goktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, “Size does matter: Why using gadget chain length to prevent code-reuse attacks is hard”

Which type of CFI did Intel choose to implement in hardware?

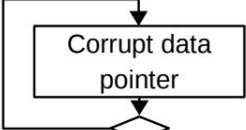
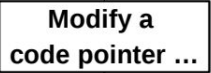
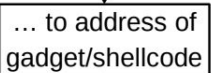
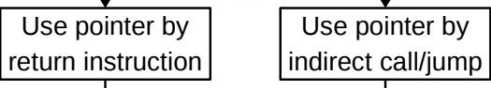

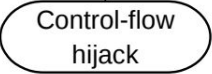
Coarse-grained CFI...

Code-Pointer Integrity

- Static analysis
 - all sensitive pointers
 - all instructions that operate on them
- Instrumentation
 - store them in a separate, safe memory region
- Instruction-level isolation mechanism
 - prevents non-protected memory operations from accessing the safe region



Defenses overview and overheads

	Attack step	Property	Mechanism	Stops all control-flow hijacks?	Avg. overhead
①		Memory Safety	SoftBound+CETS [34, 35] BBC [4], LBC [20], ASAN [43], WIT [3]	Yes No: sub-objects, reads not protected No: protects red zones only No: over-approximate valid sets	116% 110% 23% 7%
②		Code-Pointer Integrity (this work)	CPI CPS Safe Stack	Yes No: valid code ptrs. interchangeable No: precise return protection only	8.4% 1.9% ~0%
③		Randomization	ASLR [40], ASLP [26] PointGuard [13] DSR [6] NOP insertion [21]	No: vulnerable to information leaks No: vulnerable to information leaks No: vulnerable to information leaks No: vulnerable to information leaks	~10% 10% 20% 2%
④		Control-Flow Integrity	Stack cookies CFI [1] WIT (CFI part) [3] DFI [10]	No: probabilistic return protection only No: over-approximate valid sets No: over-approximate valid sets No: over-approximate valid sets	~2% 20% 7% 104%
⑤		Non-Executable Data	HW (NX bit) SW (Exec Shield, PaX)	No: code reuse attacks No: code reuse attacks	0% few %
⑥		High-level policies	Sandboxing (SFI) ACLs Capabilities	Isolation only Isolation only Isolation only	varies varies varies

kBouncer

- Detection of abnormal control transfers that take place during ROP code execution
- *Transparent*
 - Applicable on third-party applications
 - Compatible with code signing, self-modifying code, JIT, ...
- *Lightweight*
 - Up to 4% overhead when artificially stressed, practically zero
- *Effective*
 - Prevents real-world exploits

ROP Code Runtime Properties

- Illegal ret instructions that target locations not preceded by call sites
 - Abnormal condition for legitimate program code
- Sequences of relatively short code fragments “chained” through any kind of indirect branch
 - Always holds for any kind of ROP code

Illegal Returns

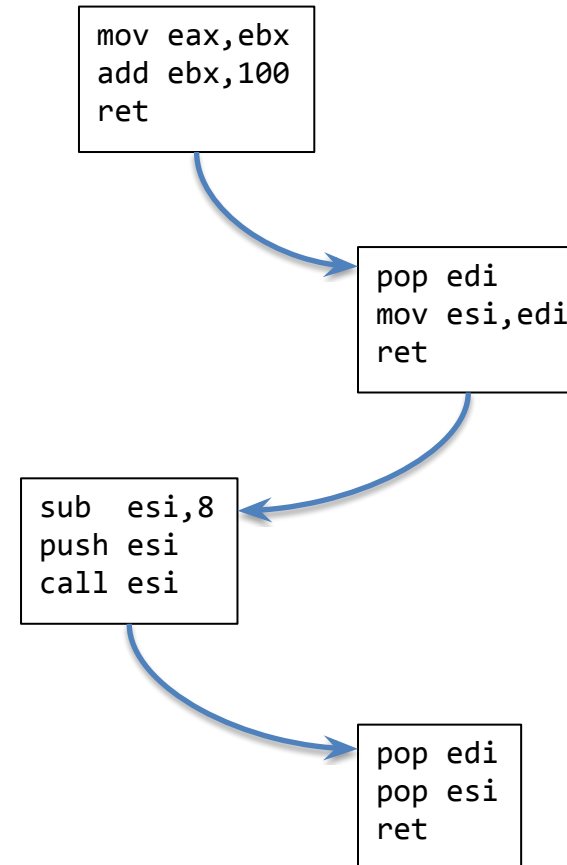
- Legitimate code:
 - ret transfers control to the instruction right after the corresponding call → legitimate call site
- ROP code:
 - ret transfers control to the first instruction of the next gadget → arbitrary locations
- Main idea:
 - Runtime monitoring of ret instructions' targets

Gadget Chaining

- Advanced ROP code may avoid illegal returns
 - Rely only on call-preceded gadgets
(just 6% of all ret gadgets in our experiments)
 - “Jump-Oriented” Programming (non-ret gadgets)
- Look for a second ROP attribute:
Several short instruction sequences chained through
(any kind of) indirect branches

Gadget Chaining

- Look for consecutive indirect branch targets that point to gadget locations
- Conservative gadget definition: up to 20 instructions
 - Typically 1-5



Last Branch Record (LBR)

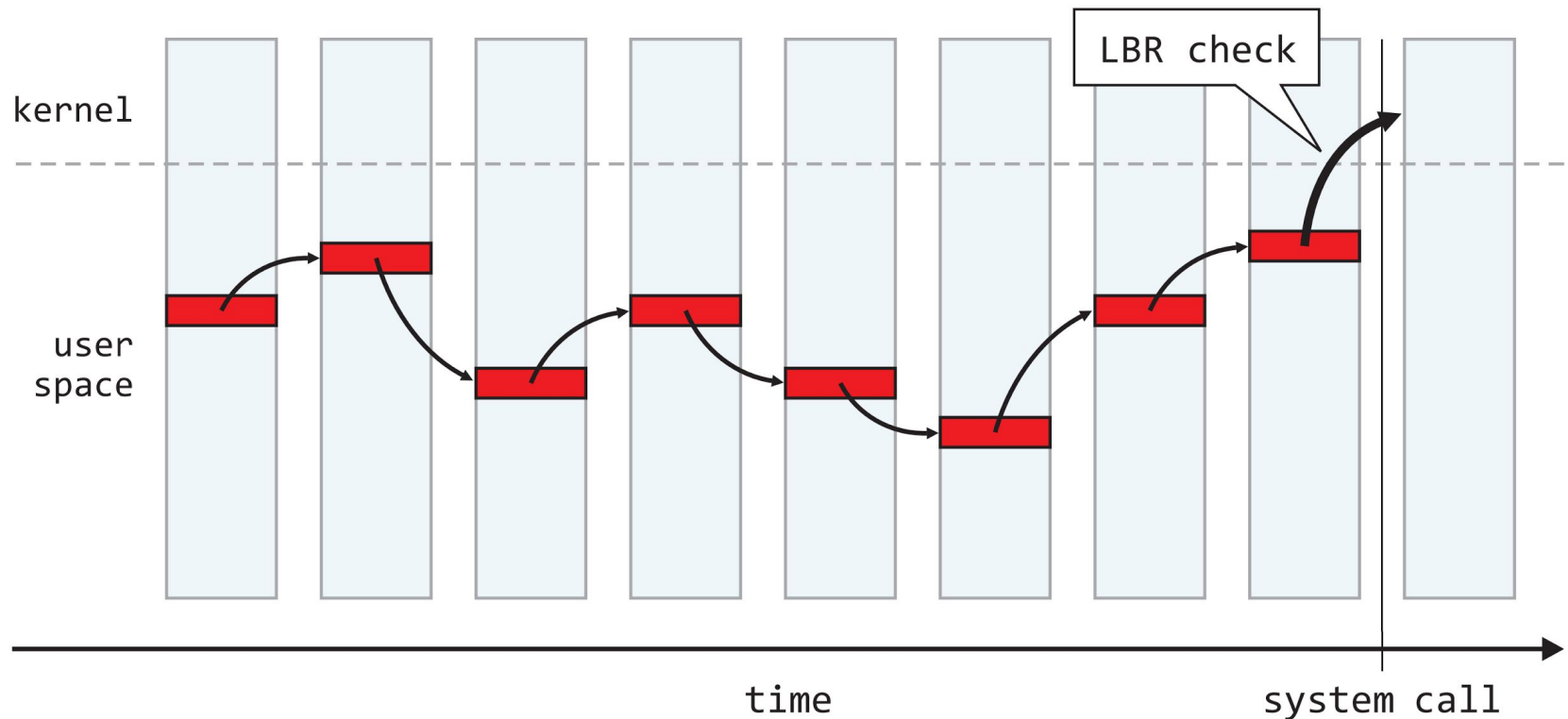
- Introduced in the Intel Nehalem architecture
- Stores the last 16 executed branches in a set of model-specific registers (MSR)
 - Can filter certain types of branches (relative/indirect calls/jumps, returns, ...)
- Multiple advantages
 - Zero overhead for recording the branches
 - Fully transparent to the running application
 - Does not require source code or debug symbols
 - Can be dynamically enabled for any running application

Monitoring Granularity

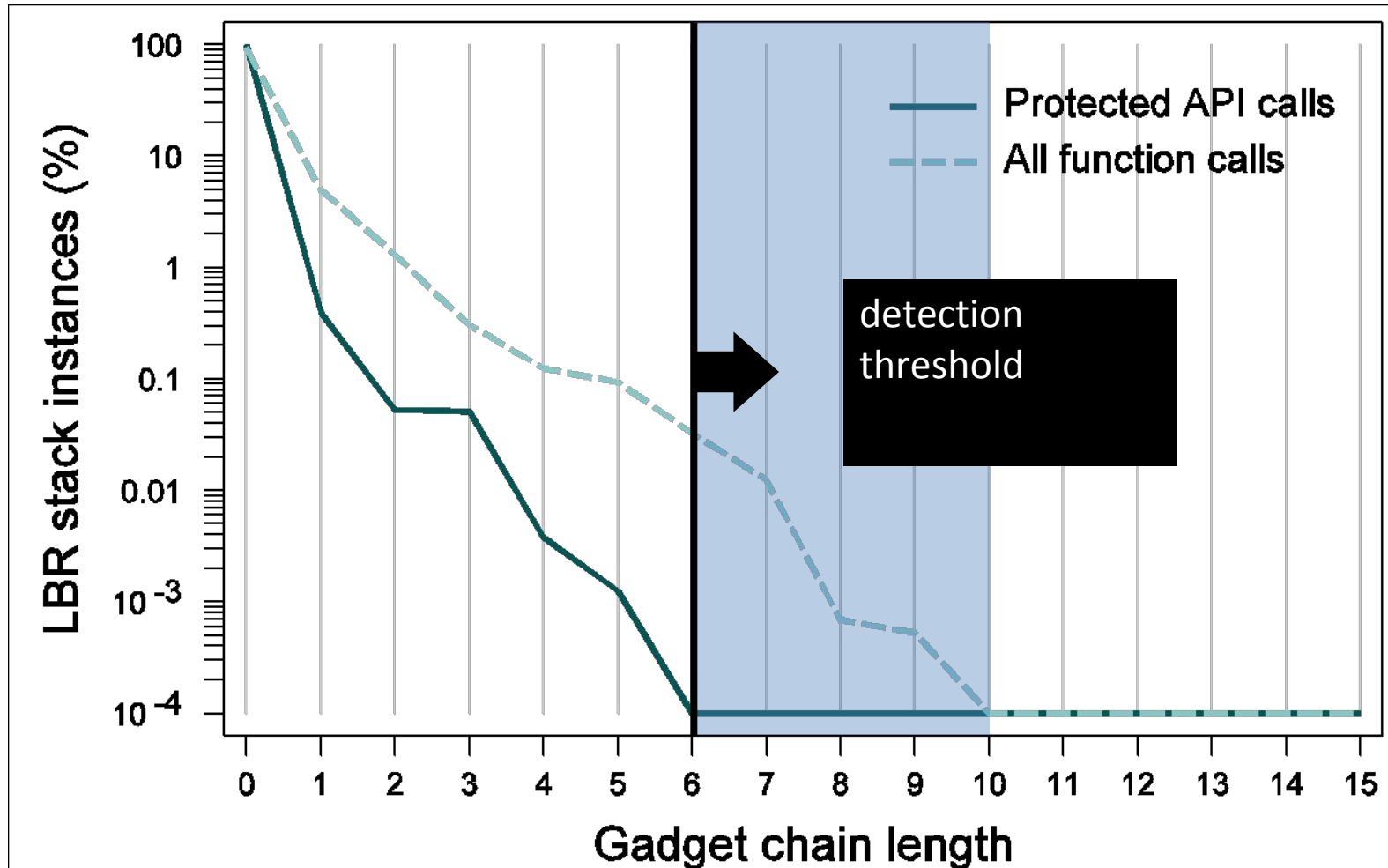
- Non-zero overhead for reading the LBR stack (accessible only from kernel level)
 - Lower frequency → lower overhead
- ROP code can run at any point
 - Higher frequency → higher accuracy

Monitoring Granularity

- Meaningful ROP code will eventually interact with the OS through system calls
 - Check for abnormal control transfers on system call entry



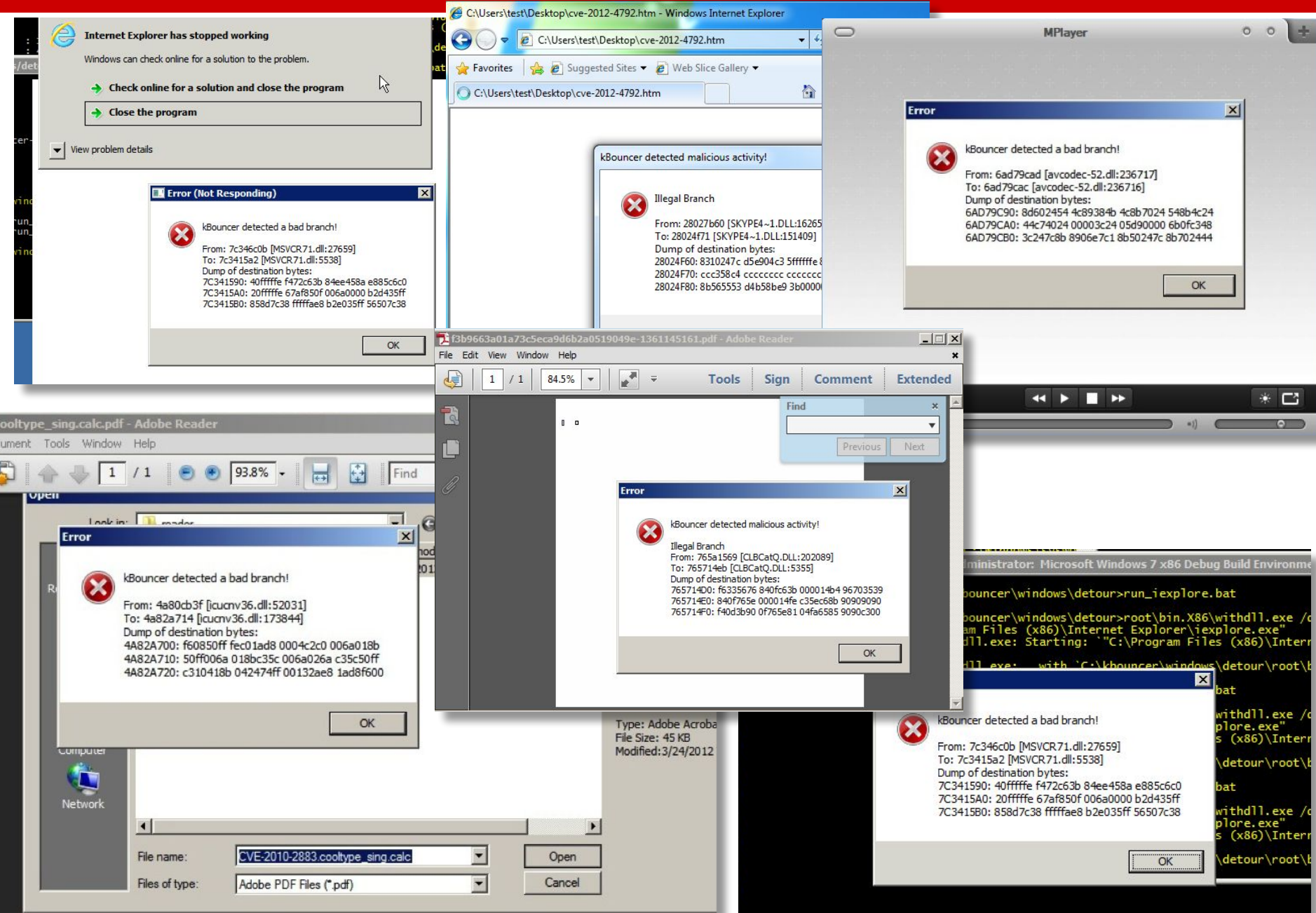
Gadget Chaining: Legitimate Code



* Dataset from: Internet Explorer, Adobe Reader, Flash Player, Microsoft Office (Word, Excel and PowerPoint)

Effectiveness

- Successfully prevented real-world exploits in
 - Adobe Reader XI (zero-day!)
 - Adobe Reader 9
 - Mplayer Lite
 - Internet Explorer 9
 - Adobe Flash 11.3
 - ...



Limitations

- Indirect branch tracing only checks the last 16 gadgets, up to 20 instructions
 - Still possible to find longer call-preceded or non-return gadgets

kBouncer



The BlueHat Prize
Winners
Announced

Your Security Zen

Six year old PDF loop bug affects most major PDF implementations

Found with afl + libfuzzer

create a PDF file with cross-referencing xref tables

A naive parser would get caught in an endless loop

Denial of service!

Mozilla Firefox

Chrome

Microsoft Edge

Ghostscript

Evince