NC STATE UNIVERSITY

CSC 405
Control-Flow Integrity

Adam Gaweda
agaweda@ncsu.edu

Alexandros Kapravelos
akaprav@ncsu.edu

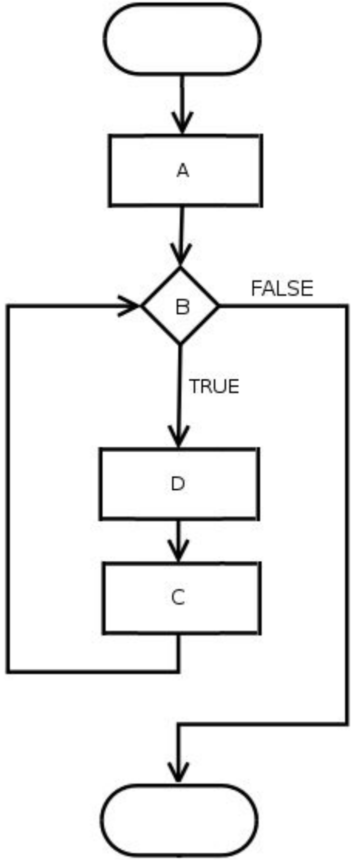
ROP & return-to-libc reusing existing code instead of injecting malicious code...

How can we stop this?

Program Control Flow

- Unconditional Jumps
- Conditional Jumps
- Loops
- Subroutines
- Unconditional Halts

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



vuln.c

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

// Same program from ROP lecture
void getinput(char *input) {
   char buffer[32];

   strcpy(buffer, input);
   printf("You entered: %s\n", buffer);
}

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

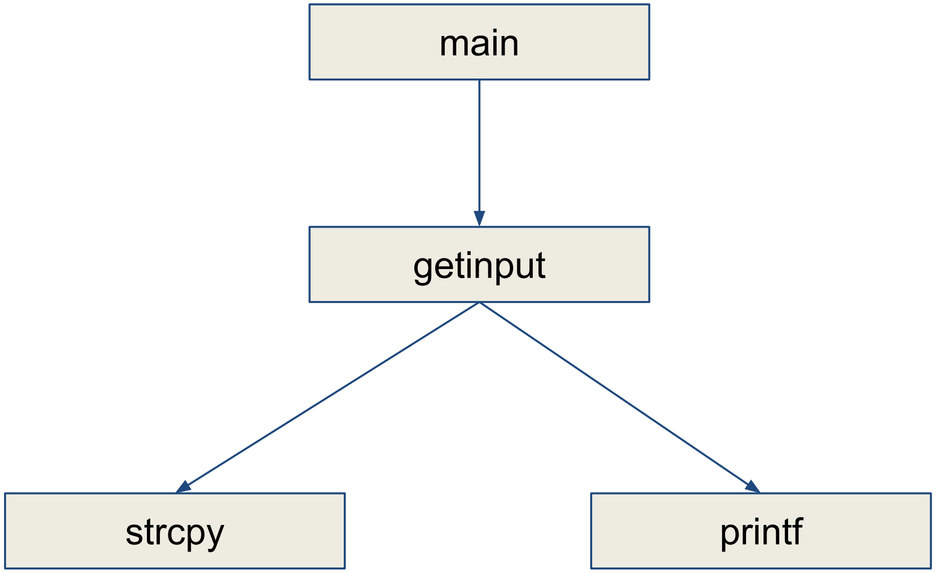
Simple Call Graph

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

// Same program from ROP lecture
void getinput(char *input) {
   char buffer[32];

   strcpy(buffer, input);
   printf("You entered: %s\n", buffer);
}

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



Function Locations

```
$ gcc vuln.c -o vuln
$ radare2 -A ./vuln
[0x000010a0]> afl
 0x00001070    1      11 sym.imp.strcpy
 0x00001080    1      11 sym.imp.__stack_chk_fail
 0x00001090    1      11 sym.imp.printf
 ...
 0x00001189    3     100 sym.getinput
 0x000011ed    1      45 main
 0x00001000    3      27 sym._init
[0x000010a0]>
```

Function Locations

```
$ gcc vuln.c -o vuln
$ radare2 -A ./vuln
[0x000010a0]> afl
 0x00001070    1      11 sym.imp.strcpy
                         ym              _chk_fail
                         ym
 ...
 0x00001189    3     100 sym.getinput
 0x000011ed    1      45 main
 0x00001000    3      27 sym._init
[0x000010a0]>
```

Size of Function in Bytes

Memory Address

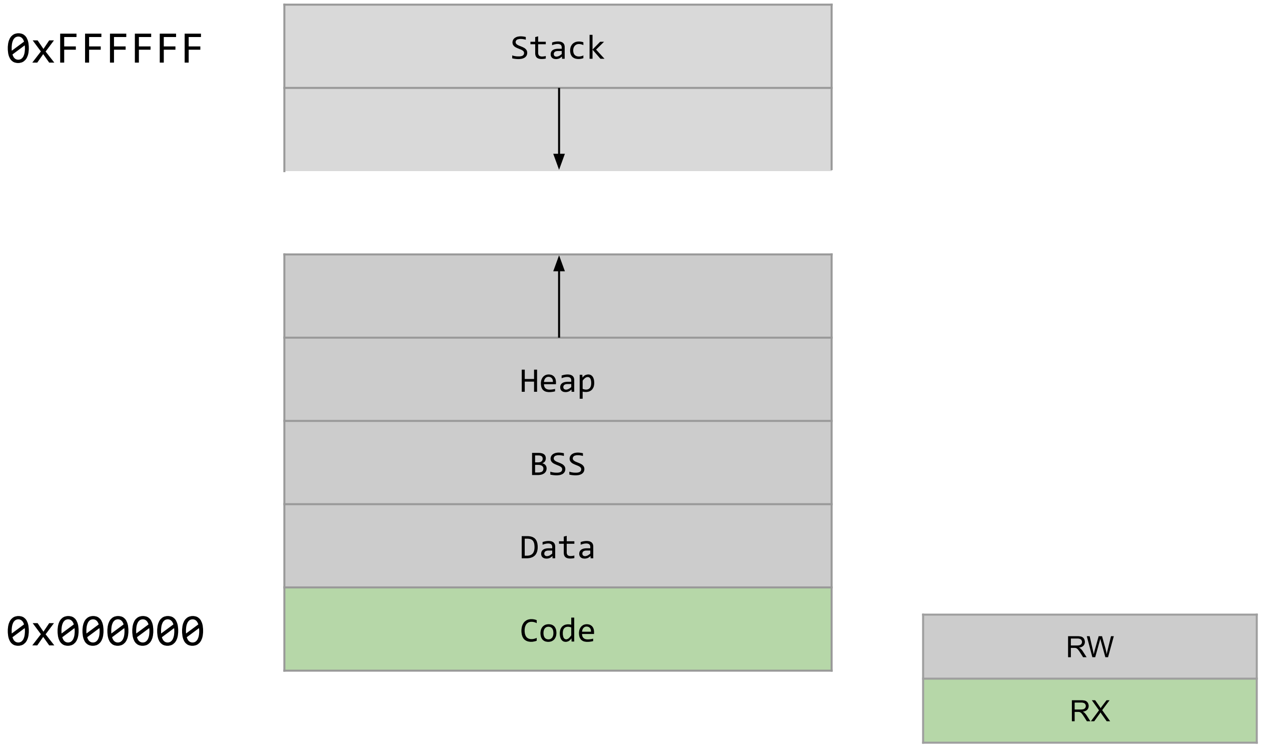
# of Basic Blocks
(code sequence with no branches in, except to the entry, and no branches out, except at the exit)

Name of function
(imp implies its imported)

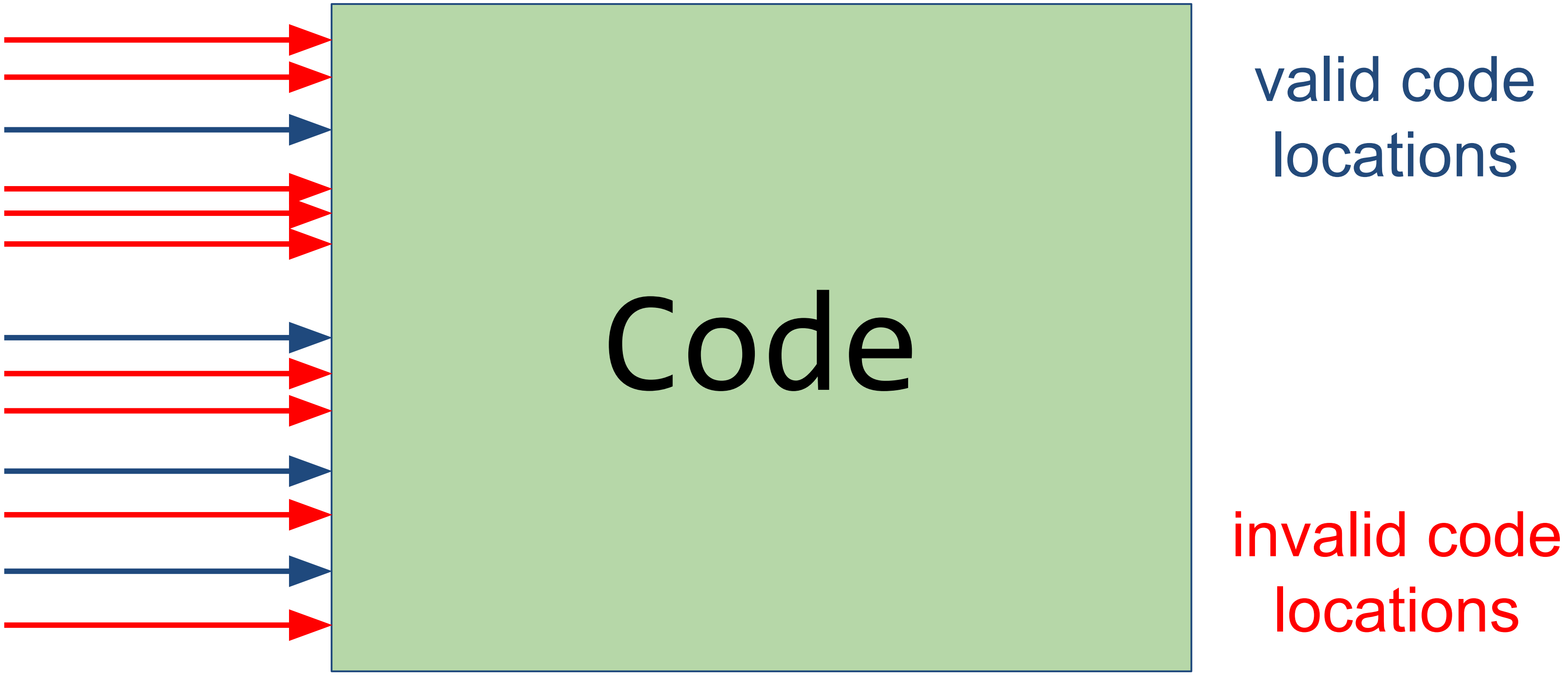
```c
void getinput(char *input) {
   char buffer[32];

   strcpy(buffer, input);
   printf("%s\n", buffer);
}
```

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?

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 Control-Flow Graph
- CFG can be pre-computed
  - source-code analysis
  - binary analysis
  - execution profiling
- But how can we enforce this extracted control-flow?

Building a Control-Flow Graph

1. Generate a .DOT file on compilation

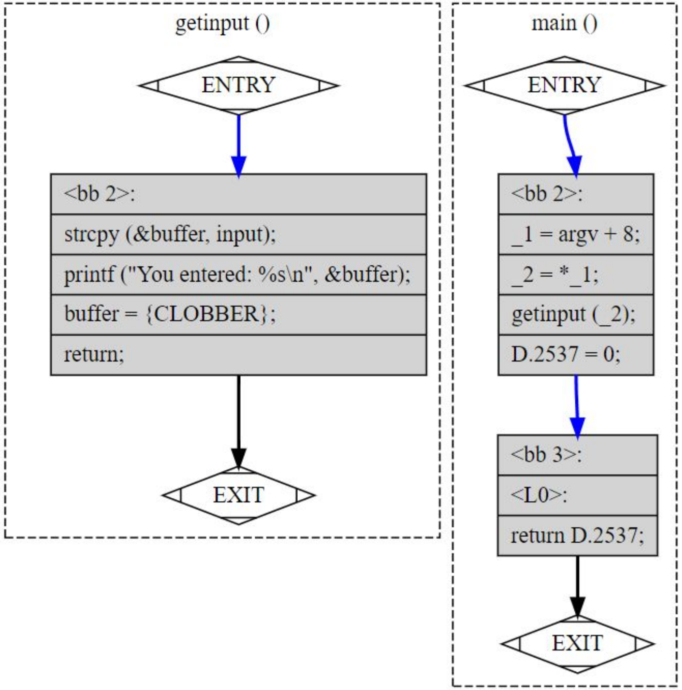
```
$ gcc -fdump-tree-all-graph -o vuln_graph/vuln vuln.c
```

Building a Control-Flow Graph

1. Generate a .DOT file on compilation

```
$ gcc -fdump-tree-all-graph -o vuln_graph/vuln vuln.c
```

2. Load the .DOT file into Graphviz or Edotor

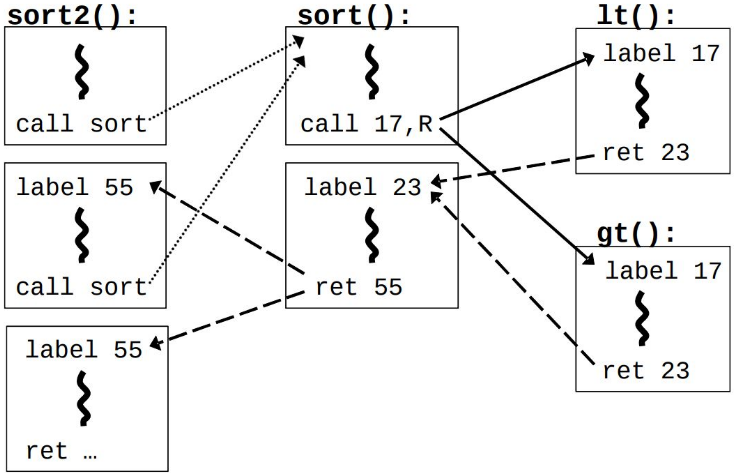


Enforcing CFI by Instrumentation

```
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` - Defines ID for code segment
- `CALL ID, DST` - Designate the ID you're expecting
- `RET ID` - Defines ID for code segment to return to

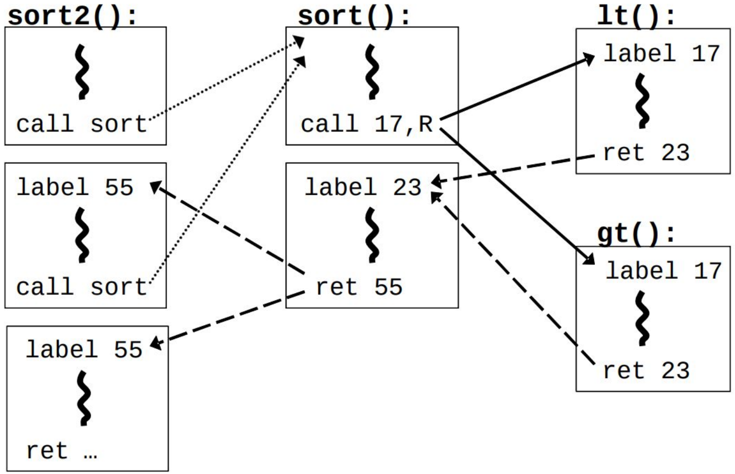
Source: [Control-Flow Integrity]

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 );
}
```



pointers to comparison functions

- `LABEL ID` - Defines ID for code segment
- `CALL ID, DST` - Designate the ID you're expecting
- `RET ID` - Defines ID for code segment to return to

Source: [Control-Flow Integrity]

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

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

Source: Control-Flow Integrity

CFI Instrumentation Code

Source / **Destination**

```
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]

CFI Instrumentation Code

Source (Opcode bytes / Instructions) — **Destination** (Opcode bytes / Instructions)

```
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

Still not implemented, but would ensure code flow

Source: Control-Flow Integrity

CFI Assumptions

- Unique IDs
  - must not be present anywhere in the code memory except in IDs and ID-checks
- Non-Writable Code (NWC)
  - must not be possible for the program to modify code memory at runtime
- Non-Executable Data (NXD)
  - must not be possible for the program to execute data as if it were code
- Jumps cannot go into the middle of instructions

CFI Assumptions

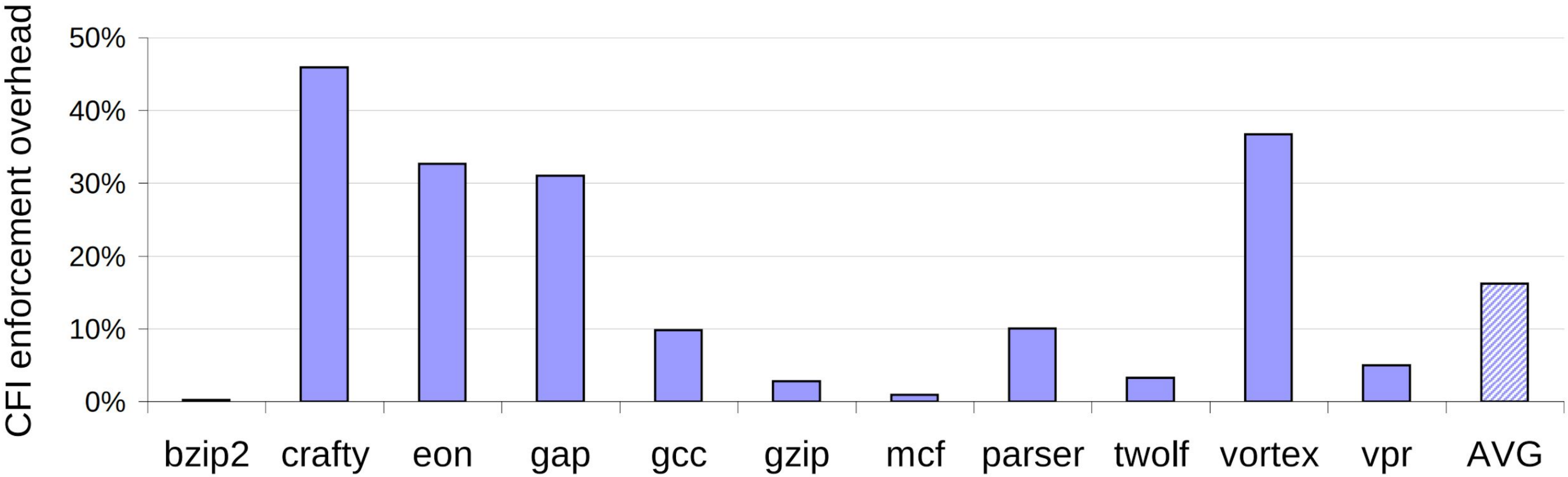
- Unique IDs
  - must not be present anywhere in the code memory except in IDs and ID-checks
- Non-Writable Code (NWC)
  - must not be possible for the program to modify code memory at runtime
- Non-Executable Data (NXD)
  - must not be possible for the program to execute data as if it were code
- Jumps cannot go into the middle of instructions

What code do you compile **everyday** that would cause problems with this?

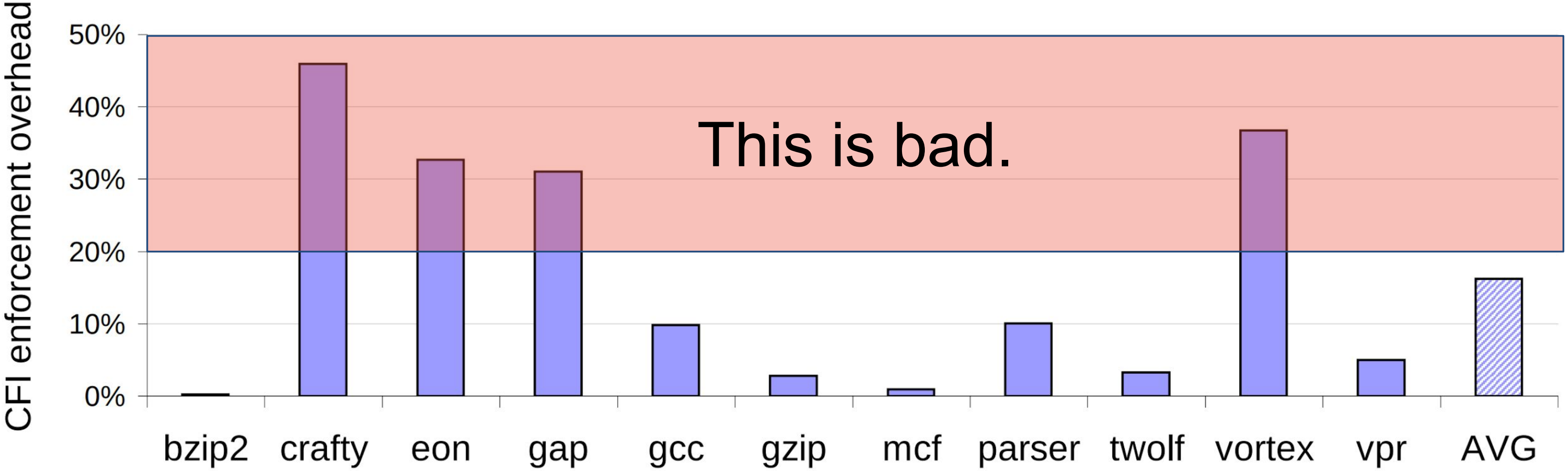
Attacker

- The paper assumes a 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 Control-Flow Graph
  - Attacker can only execute the normal flow of the program

CFI Enforcement Overhead



CFI Enforcement Overhead



Control-Flow Guard (semi-implemented)

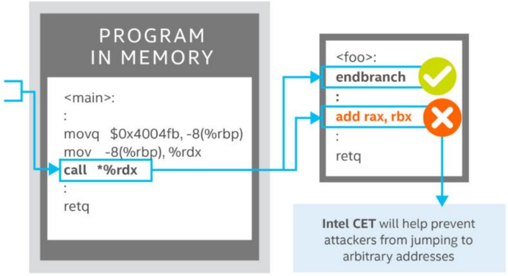
- Windows 10 and Windows 8.1
- Microsoft Visual Studio 2015+
- Adds lightweight security checks to the compiled code
- Identifies the set of functions in the application that are valid targets for indirect calls
- The runtime support, provided by the Windows kernel:
  - Efficiently maintains state that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

Intel® Control-Flow Enforcement Technology (Intel CET)

INTEL CET = INDIRECT BRANCH TRACKING (IBT) + SHADOW STACK (SS)

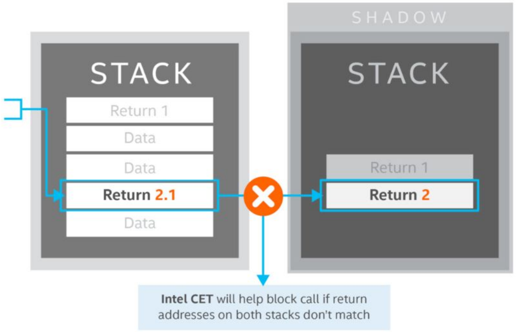
INDIRECT BRANCH TRACKING (IBT)

IBT delivers indirect branch protection to defend against jump/call oriented programming (JOP/COP) attack methods.



SHADOW STACK (SS)

SS delivers return address protection to defend against return-oriented programming (ROP) attack methods.



Intel CET helps protect against ROP/JOP/COP malware

Intel CET is built into the hardware microarchitecture and available across the family of products with that core. On Intel vPro® platforms with Intel® Hardware Shield, Intel CET further extends threat protection capabilities.

Control-Flow Enforcement Technology

- **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

- **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**)

Limitations of CFI?

Limitations of CFI?

What if your program has instructions that could be maliciously used?

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

Coarse-Grained CFI

- Trades security for better performance

- Any valid call location is accepted

However, this creates vulnerabilities…

[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"

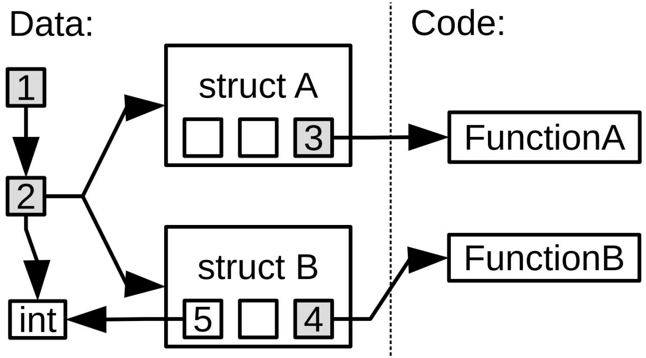
Which type of CFI did Intel choose to implement in hardware?

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

Coarse-grained CFI...

Code-Pointer Integrity

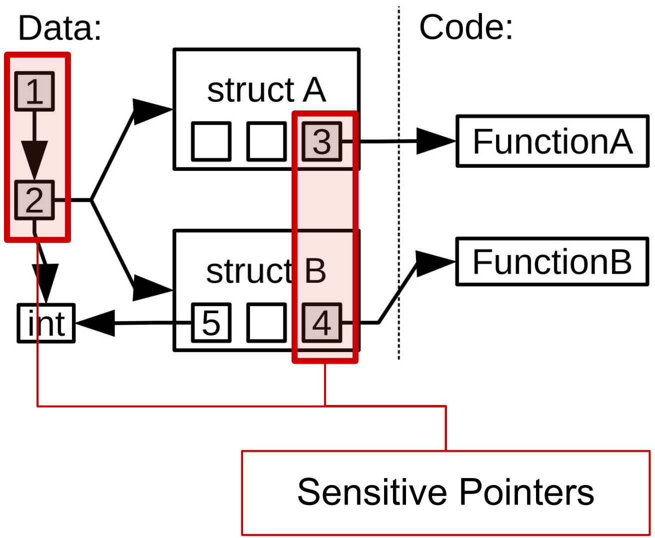
- Static Analysis
  - all sensitive pointers
  - all instructions that operate on them
- Instrumentation
  - store them in a separate, safe memory region
- Instruction-level Isolation Mechanism
  - prevents non-protected memory operations from accessing the safe region



Source: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-kuznetsov.pdf

Code-Pointer Integrity

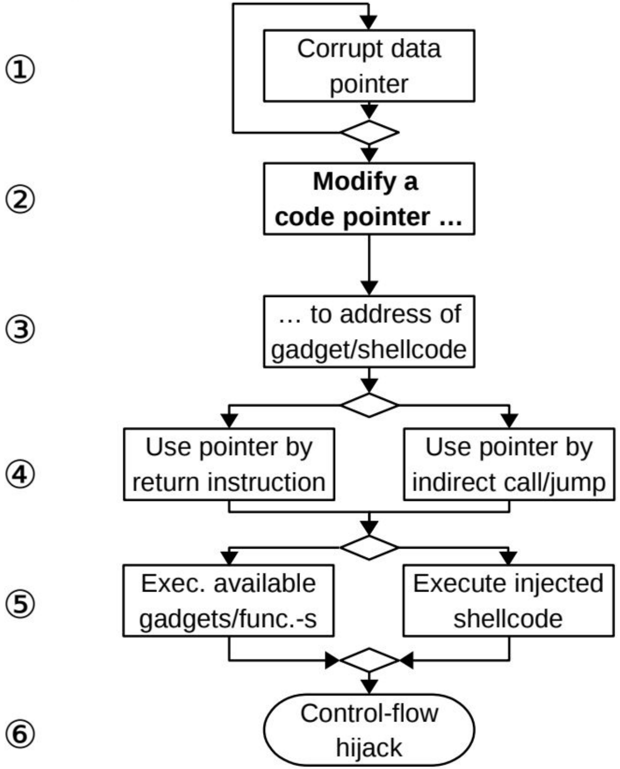
- Static Analysis
  - all sensitive pointers
  - all instructions that operate on them
- Instrumentation
  - store them in a separate, safe memory region
- Instruction-level Isolation Mechanism
  - prevents non-protected memory operations from accessing the safe region



Source: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-kuznetsov.pdf

Defense Overview and Overheads

Attack step



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

kBouncer

- Detect abnormal control transfers that take place during ROP code execution
  – Reviews last few jump calls to see if the average number of instructions execute is too small (gadgets are <10 instructions)

- **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

Source: https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_pappas.pdf

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

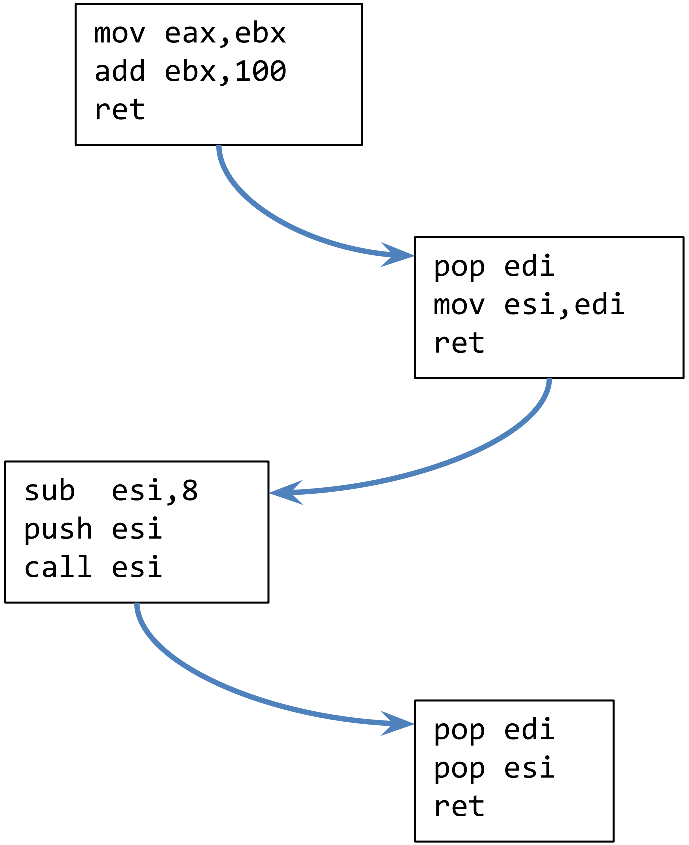
- Legitimate code:
  - `ret` transfers control to the instruction right after the corresponding call ➔ legitimate call site
- ROP code:
  - `ret` transfers control to the first instruction of the next gadget ➔ arbitrary locations
- Main idea:
  - Runtime monitoring of `ret` instructions' targets

Gadget Chaining

- Advanced ROP code may avoid illegal returns
  - Rely only on call-preceded gadgets (6% of all **ret** gadgets in the experiments)
  - "Jump-Oriented" Programming (non-**ret** gadgets)
- Look for a second ROP attribute:
  - Several short instruction sequences chained through 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)

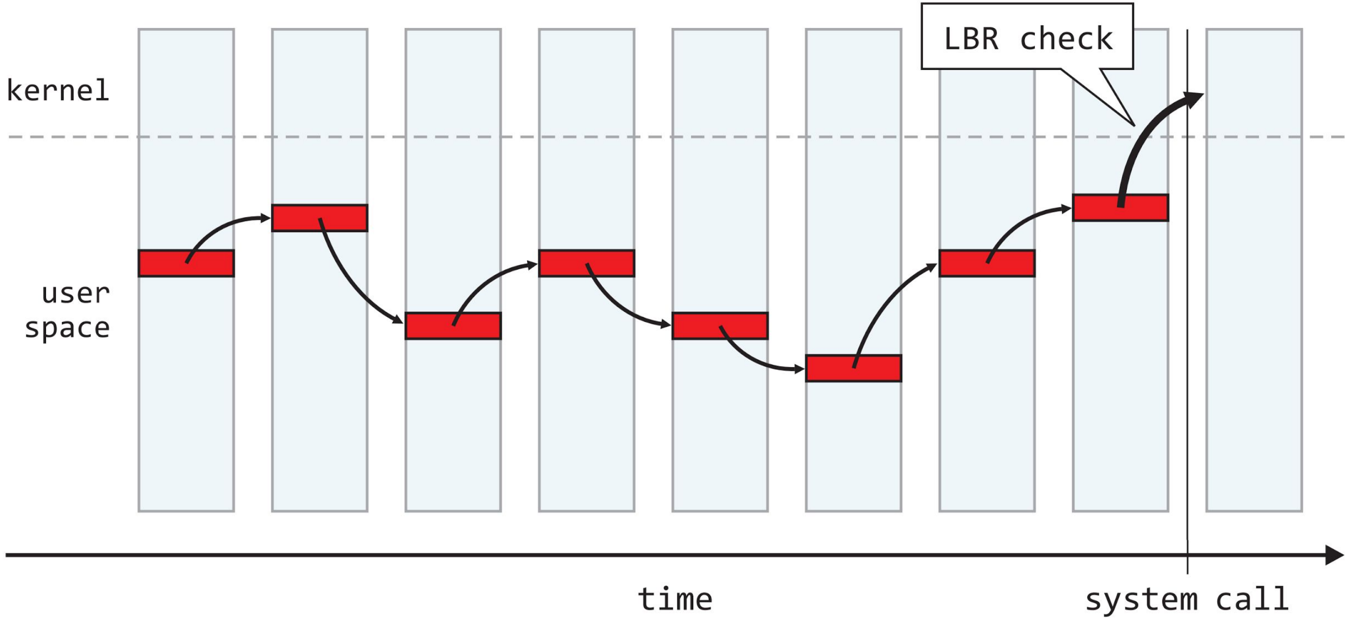
- Introduced in the Intel Nehalem (i5 and i7) architecture
- Stores the last 16 executed branches in a set of model-specific registers (MSR)
  - Can filter certain types of branches (relative/indirect calls/jumps, returns, ...)
- Multiple advantages
  - Zero overhead for recording the branches
  - Fully transparent to the running application
  - Does not require source code or debug symbols
  - Can be dynamically enabled for any running application

Monitoring Granularity

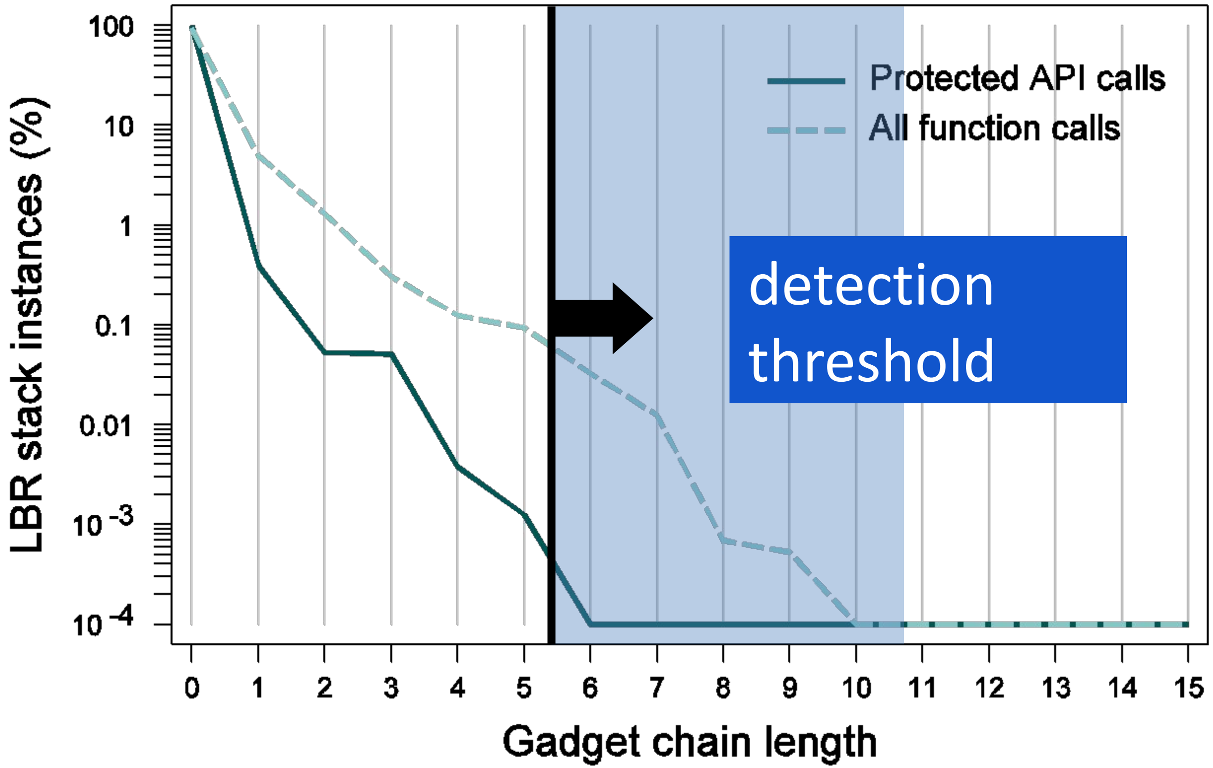
- Non-zero overhead for reading the LBR stack (accessible only from kernel level)
  - Lower frequency → lower overhead
  - Higher frequency → higher overhead
- ROP code can run at any point
  - Higher frequency → higher accuracy

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

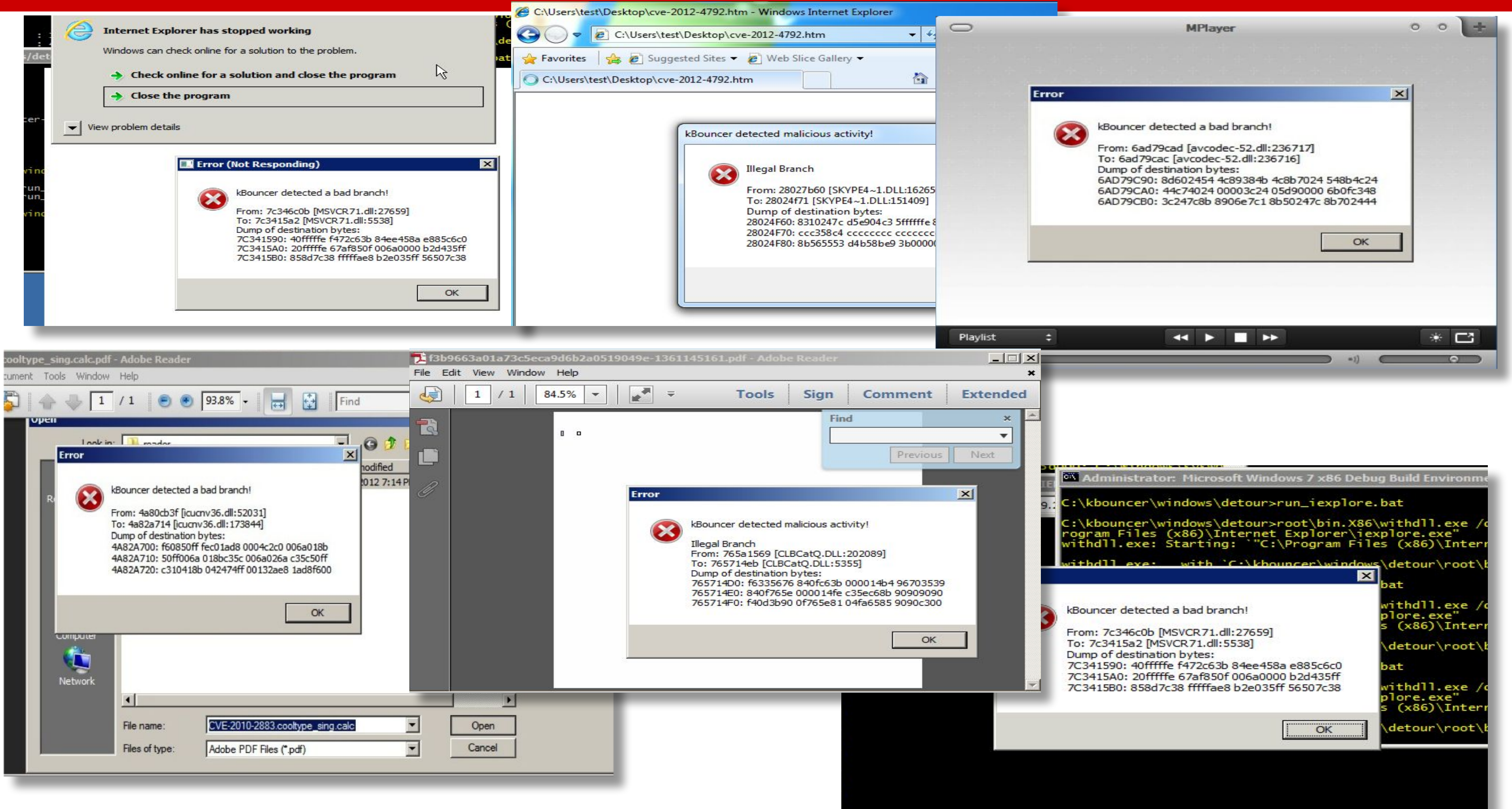


Dataset from: Internet Explorer, Adobe Reader, Flash Player, Microsoft Office

Effectiveness

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

NC STATE UNIVERSITY



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