

DAC Tutorial
6 June, Austin, TX, USA

The Continuing Arms Race: A Journey in the World of Runtime Exploits and Defenses

Lucas Davi, Ahmad-Reza Sadeghi

CRISP, Technische Universität Darmstadt
**Intel Collaborative Research Institute for Secure
Computing at TU Darmstadt, Germany**

Special Session Announcement

- ◆ **Secure IoT: Utopia, Alchemy, or Possible Future?**
 - ◆ Organizers: Ahmad-Reza Sadeghi (TU Darmstadt) and Yier Jin (Univ. of Central Florida)
 - ◆ Chair: Anand Rajan (Intel Corp.)
 - ◆ Co-Chair: Saverio Fazzari (Booz Allen Hamilton, Inc.)
- ◆ **THURSDAY June 09, 10:30am - 12:00pm | 18AB**
- ◆ **Talks**
 - ◆ Things, Trouble, Trust: On Building Trust in IoT Systems
 - ◆ Exploring risk and mapping the Internet of Things with Autonomous Drones
 - ◆ Can IoT be Secured: Emerging Challenges in Connecting the Unconnected

Motivating Problem



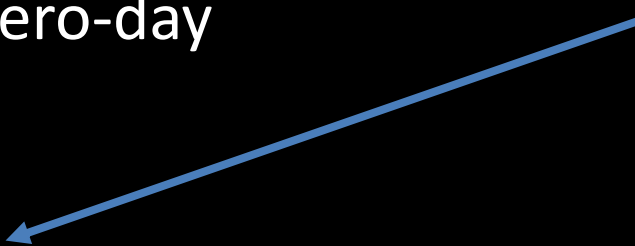
- Software increasingly sophisticated and complex
- Various developers involved
- Native Code
- Many program bugs

Large attack surface for runtime exploits on diverse platforms

Introduction

- ◆ Vulnerabilities
 - ◆ Programs continuously suffer from program bugs, e.g., a buffer overflow
 - ◆ Memory errors
 - ◆ CVE statistics; zero-day
- ◆ Runtime Attack
 - ◆ Exploitation of program vulnerabilities to perform malicious program actions
 - ◆ Control-flow attack; runtime exploit

Focus in this
tutorial



Three Decades of Runtime Attacks

Morris Worm
1988

return-into-
libc
Solar Designer
1997

Return-oriented
programming
Shacham
CCS 2007

Continuing Arms
Race

Code
Injection
AlephOne
1996

Borrowed
Code Chunk
Exploitation
Krahmer
2005

...

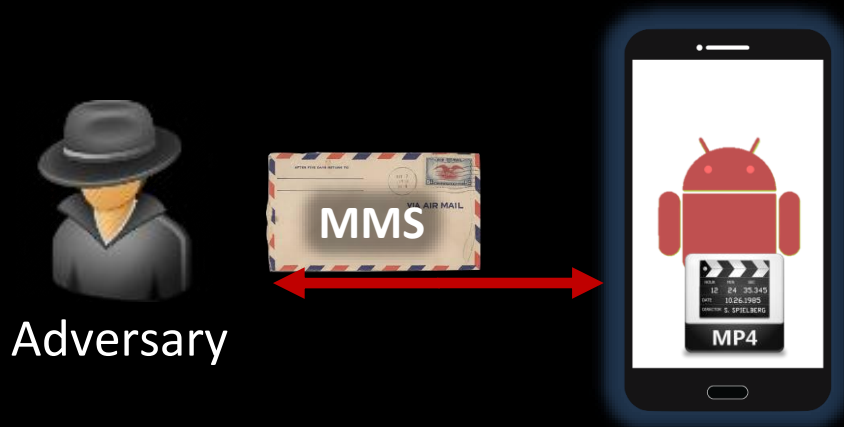
Are these attacks relevant?



Recent Attacks

Stagefright [Drake, BlackHat 2015]

These issues in Stagefright code critically expose 95% of Android devices, an estimated 950 million devices



Cisco Router Exploit [2016]

Million CISCO ASA Firewalls potentially vulnerable to attacks



Relevance and Impact

High Impact of Attacks

- Web browsers repeatedly exploited in pwn2own contests
- Zero-day issues exploited in Stuxnet/Duqu [Microsoft, BH 2012]
- iOS jailbreak

Industry Efforts on Defenses

- Microsoft EMET (Enhanced Mitigation Experience Toolkit) includes a ROP detection engine
- Microsoft Control Flow Guard (CFG) in Windows 10
- Google's compiler extension VTV (virtual table verification)

Hot Topic of Research

- A large body of recent literature on attacks and defenses

**But runtime exploits have also some
“good” side-effects**



Apple iPhone Jailbreak

Disable signature verification and escalate privileges to root



Request

http://www.jailbreakme.com/_/iPhone3,1_4.0.pdf



- 1) Exploit PDF Viewer Vulnerability by means of **Return-Oriented Programming**
- 2) Start Jailbreak
- 3) Download required system files
- 4) Jailbreak Done

Tutorial Outline

1. Lecture on Runtime Exploits

- ◆ Introduction
- ◆ Selected Background on ARM
- ◆ Code Injection
- ◆ Code-Reuse Attacks
- ◆ Modern Defense Techniques and Their Limitations
- ◆ Hardware-Assisted Protection Schemes

2. Hands-on Lab (Runtime attacks against Android-ARM)

BASICS

What is a runtime attack ?



Big Picture: Program Compilation



Source Code
C



```
COPY ( buffer[8], *usr_input )
```

Compile

Executable
binary

```
mov reg0[0-3], reg1[0-3]  
mov reg0[4-n], reg1[4-n]
```

reg0

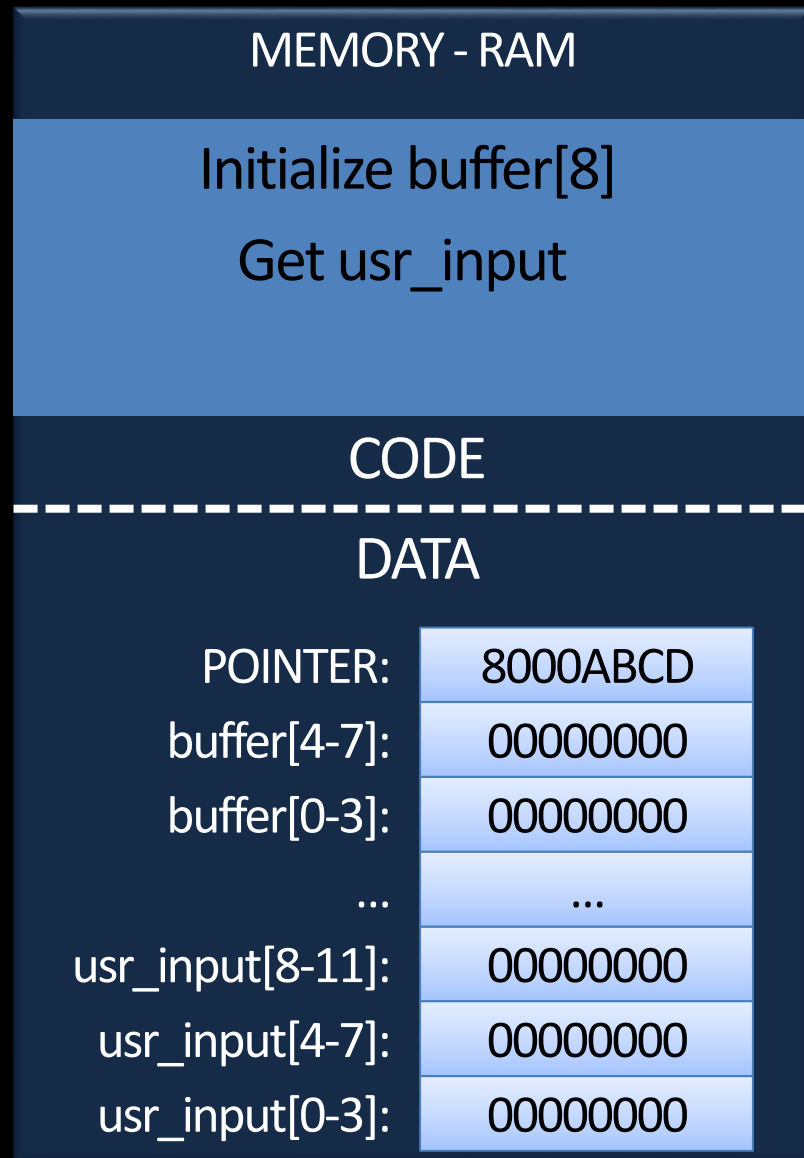
buffer[8]

reg1

usr_input

Big Picture: Program Execution

Executable
binary



Big Picture: Program Execution

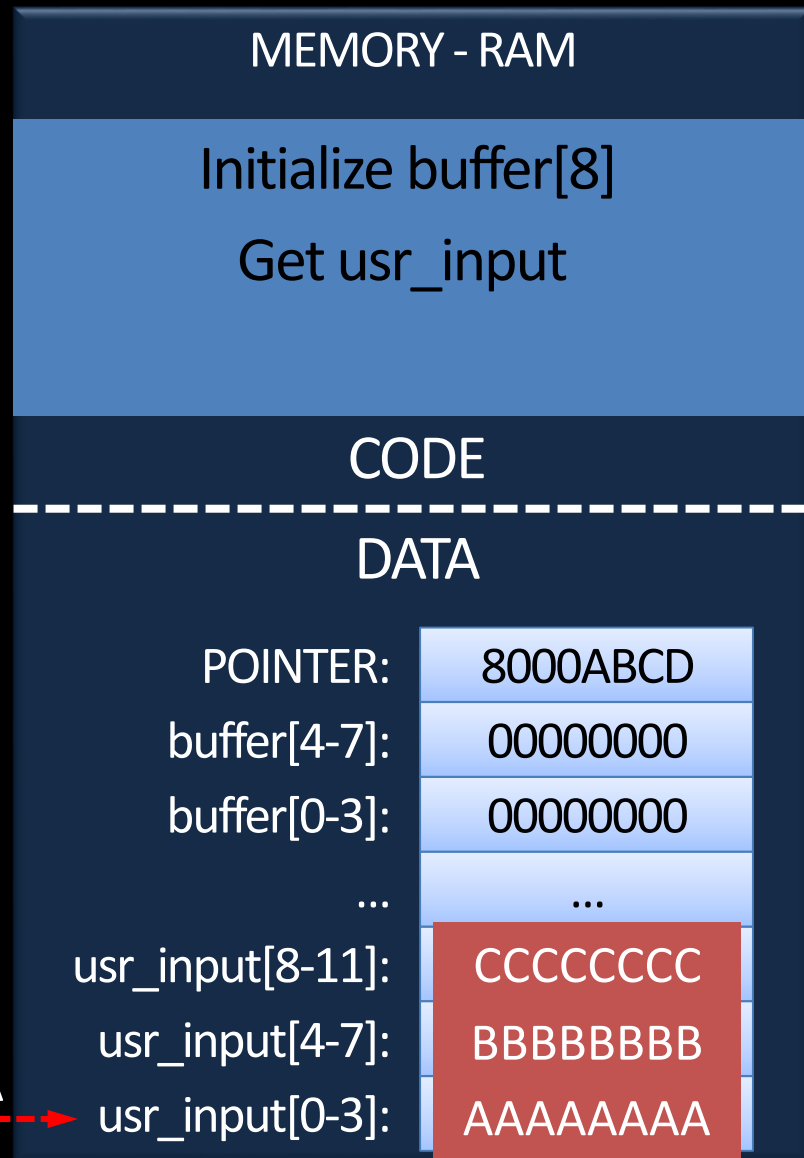
Executable
binary



CCCCCCCC

BBBBBBBB

AAAAAAAA



Big Picture: Program Execution

Executable
binary



MEMORY - RAM

Initialize buffer[8]

Get usr_input

COPY (buffer[8], *usr_input)

CODE

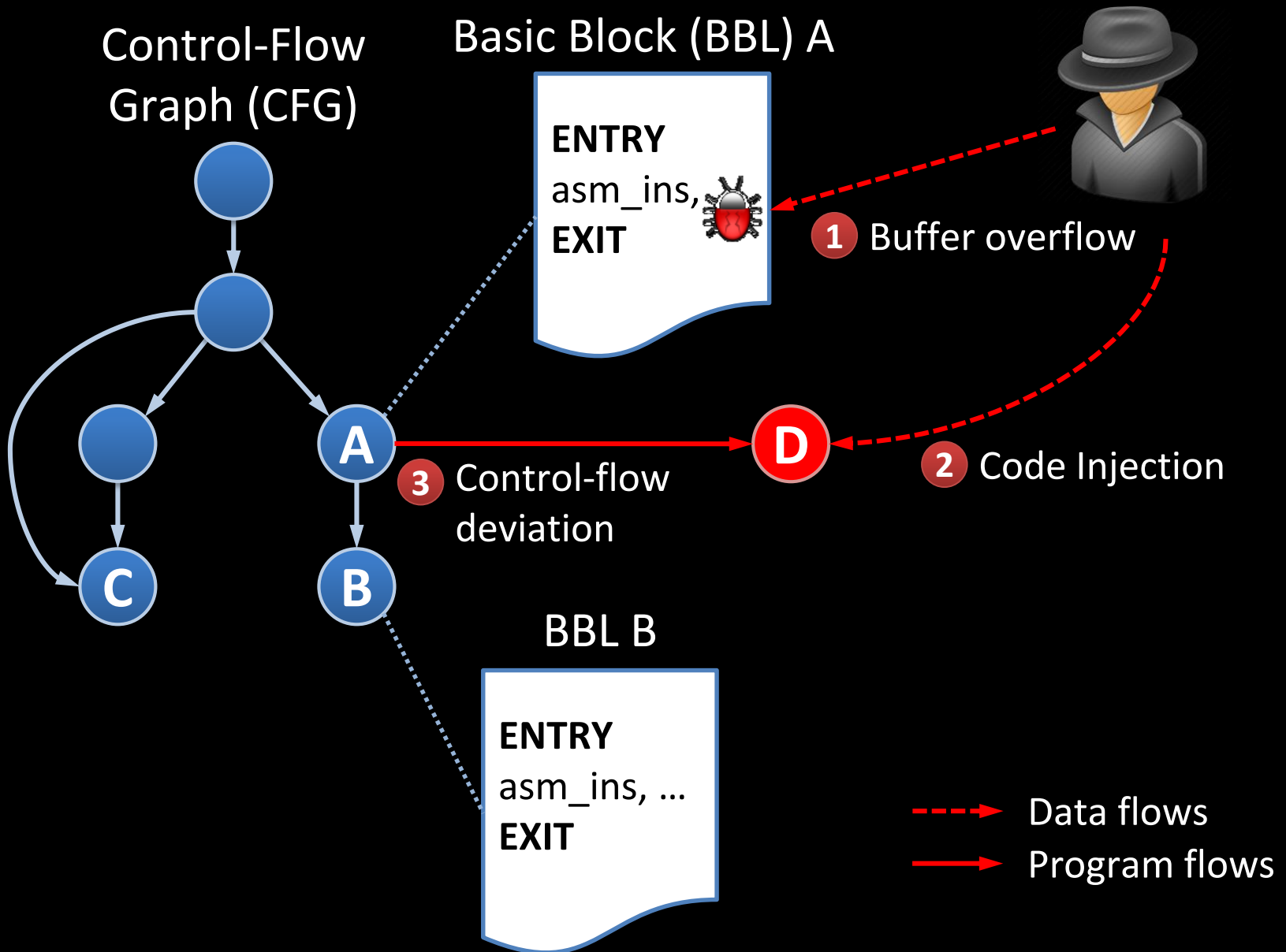
DATA

POINTER:	CCCCCCCC
buffer[4-7]:	BBBBBBBB
buffer[0-3]:	AAAAAAAA
...	...
usr_input[8-11]:	CCCCCCCC
usr_input[4-7]:	BBBBBBBB
usr_input[0-3]:	AAAAAAAA

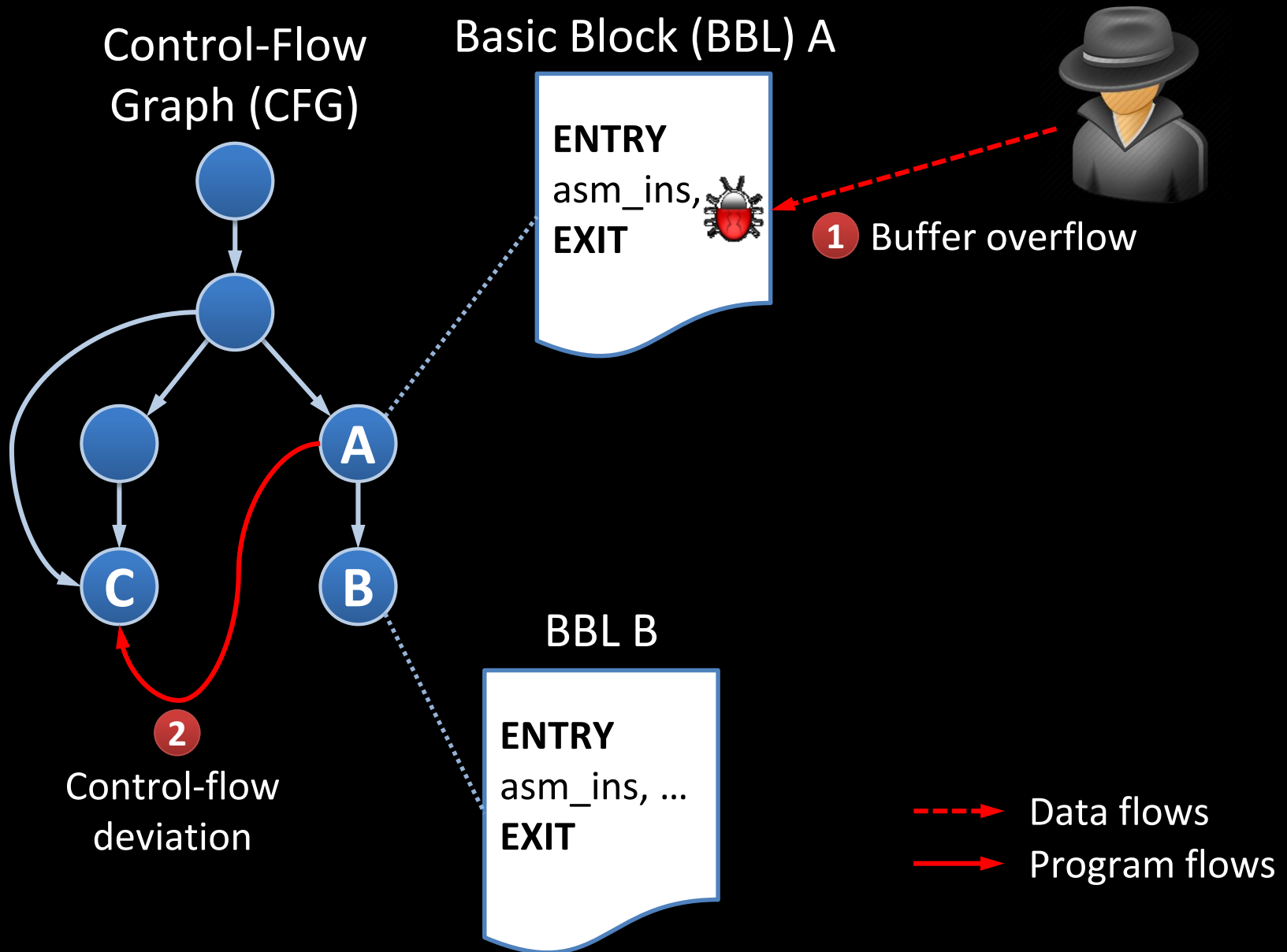
Observations

- ♦ There are several observations
 1. A programming error leads to a program-flow deviation
 2. Missing **bounds checking**
 - ♦ Languages like C, C++, or assembler do not automatically enforce bounds checking on data inputs
 3. An adversary can provide inputs that influence the program flow
- ♦ What are the consequences?

General Principle of Code Injection Attacks



General Principle of Code Reuse Attacks



Code Injection vs. Code Reuse

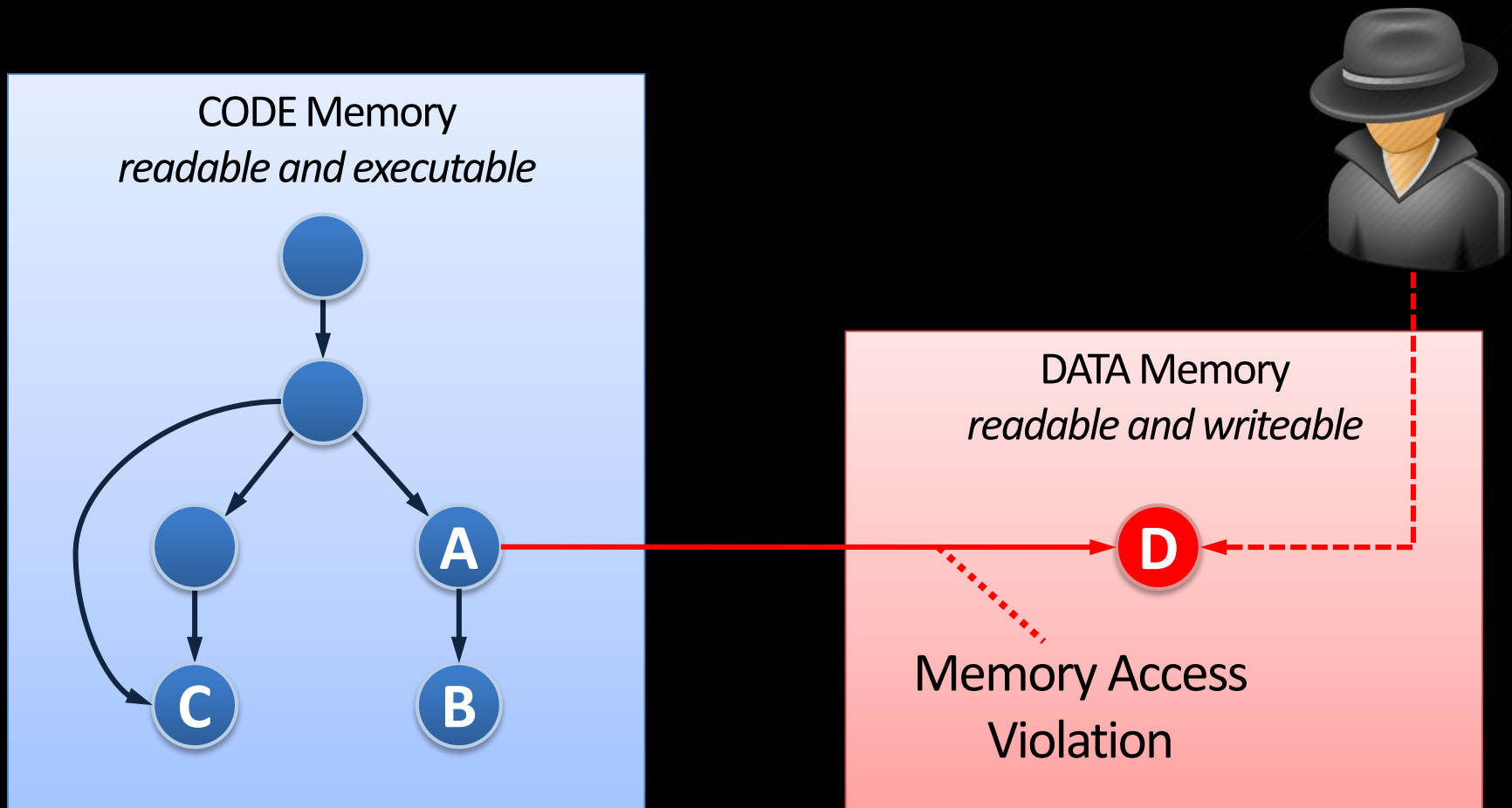
- ◆ Code Injection – *Adding a new **node** to the CFG*
 - ◆ Adversary can execute arbitrary malicious code
 - ◆ open a remote console (classical shellcode)
 - ◆ exploit further vulnerabilities in the OS kernel to install a virus or a backdoor
- ◆ Code Reuse – *Adding a new **path** to the CFG*
 - ◆ Adversary is limited to the code nodes that are available in the CFG
 - ◆ Requires reverse-engineering and static analysis of the code base of a program

BASICS

**Code injection is more powerful;
so why are attacks today
typically using code reuse?**

Data Execution Prevention (DEP)

- ◆ Prevent execution from a writable memory (data) area

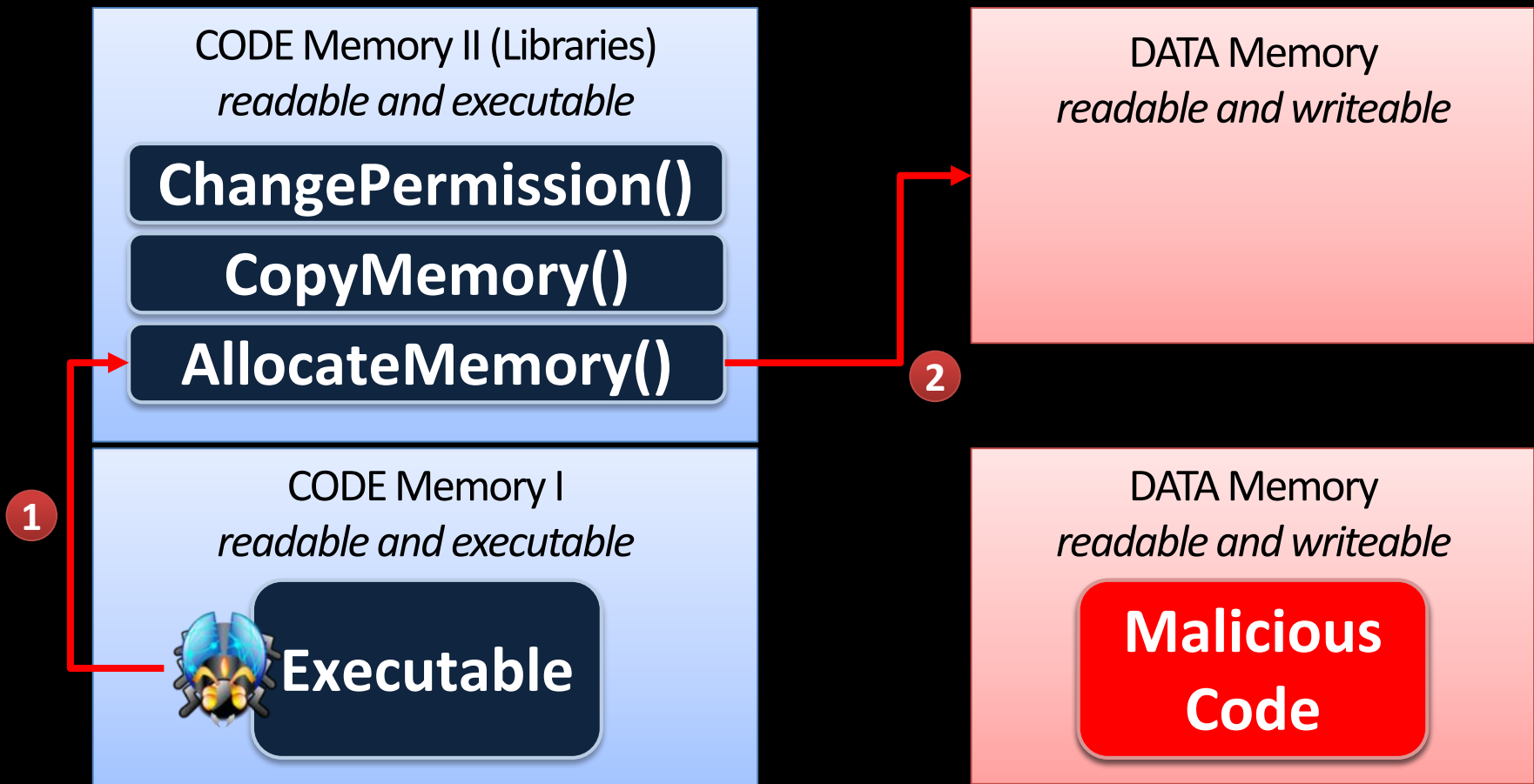


Data Execution Prevention (DEP) cntd.

- ◆ Implementations
 - ◆ Modern OSes enable DEP by default (Windows, Linux, iOS, Android, Mac OSX)
 - ◆ Intel, AMD, and ARM feature a special No-Execute bit to facilitate deployment of DEP
- ◆ Side Note
 - ◆ There are other notions referring to the same principle
 - ◆ $W \oplus X$ – Writeable XOR eXecutable
 - ◆ Non-executable memory

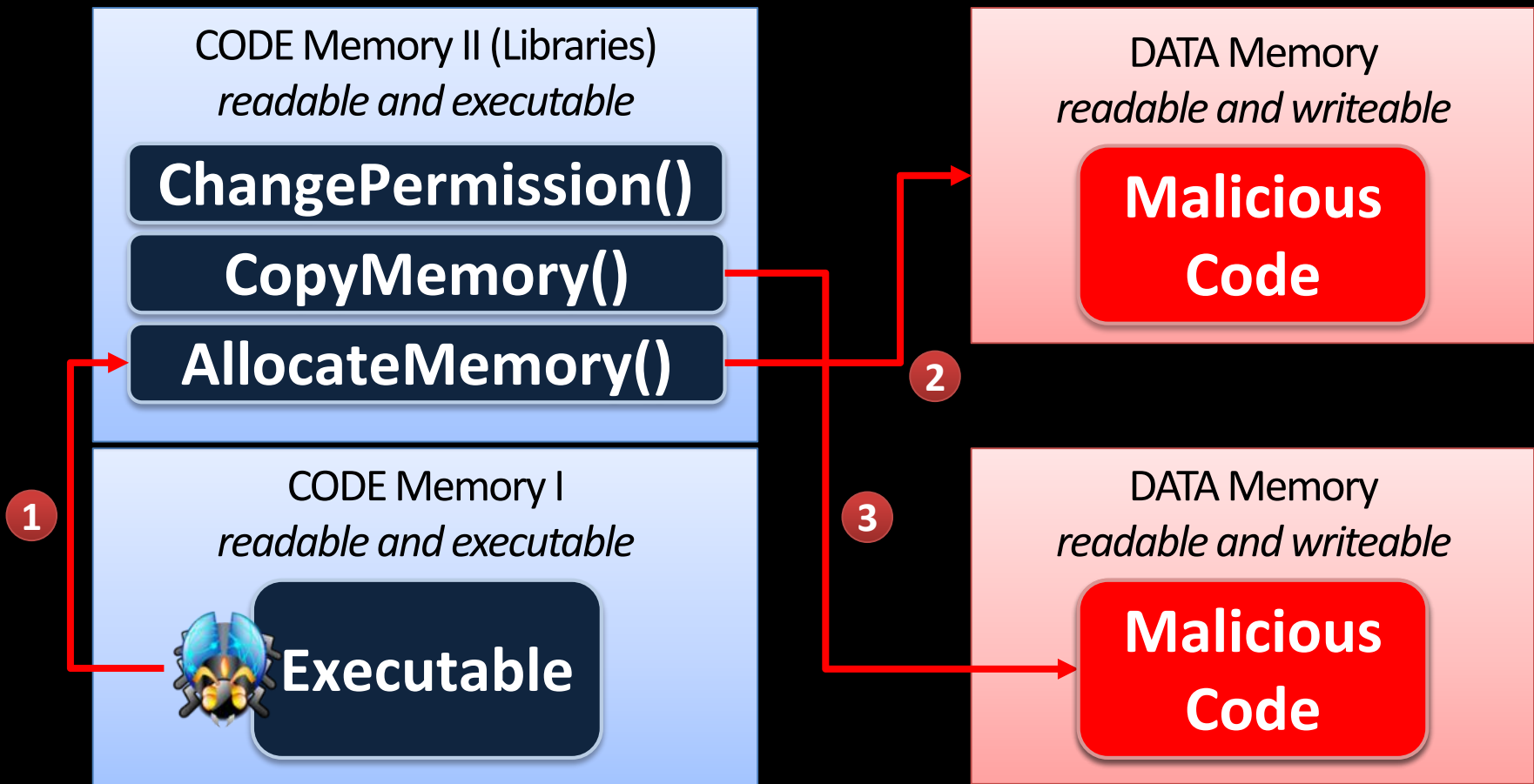
Hybrid Exploits

- Today's attacks combine code reuse with code injection



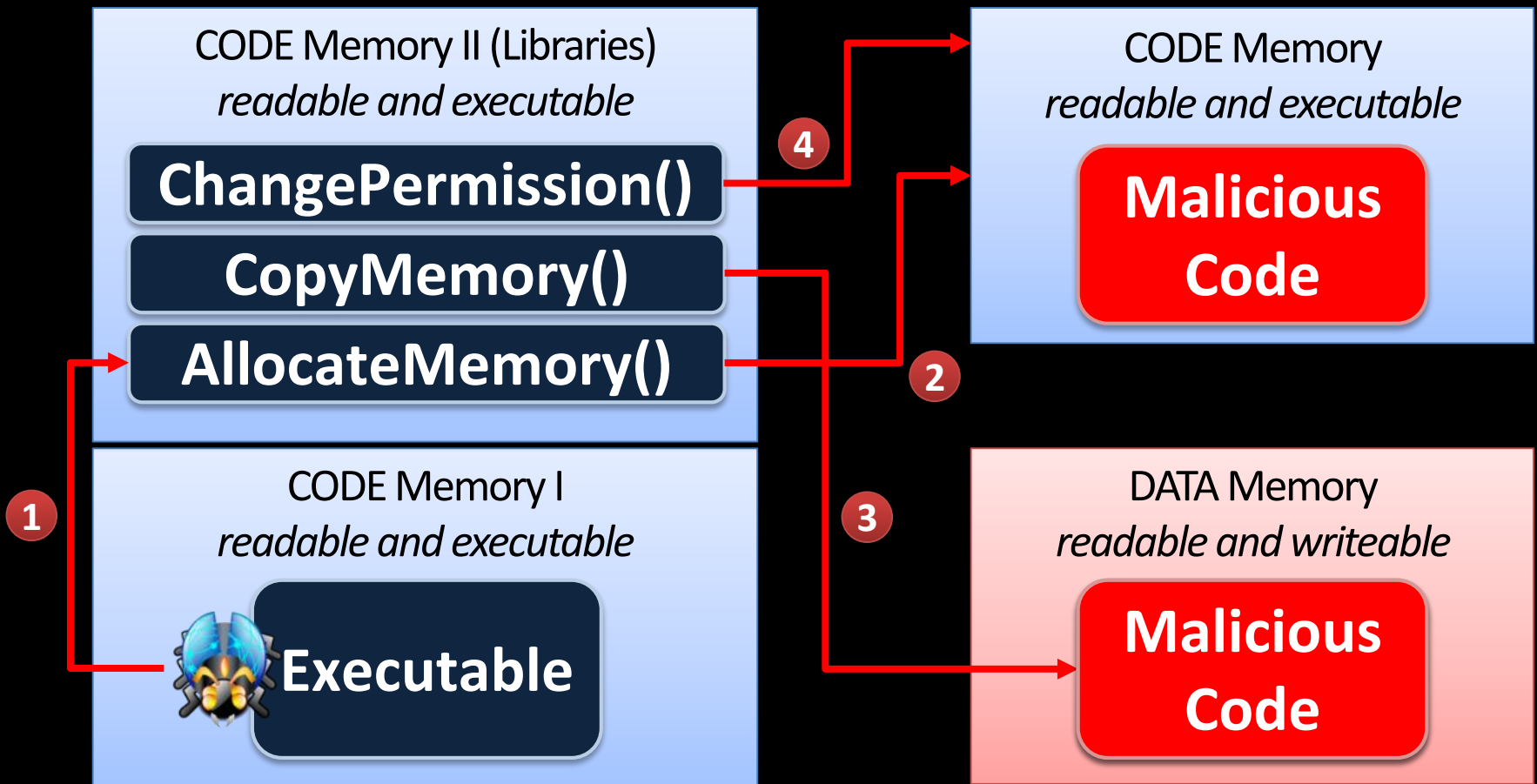
Hybrid Exploits

- Today's attacks combine code reuse with code injection



Hybrid Exploits

- Today's attacks combine code reuse with code injection



**Selected background on ARM registers,
stack layout, and calling convention**

ARM Overview

- ♦ ARM stands for **Advanced RISC Machine**
- ♦ Main application area: Mobile phones, smartphones (Apple iPhone, Google Android), music players, tablets, and some netbooks
- ♦ Advantage: **Low power consumption**
- ♦ Follows **RISC design**
 - ♦ Mostly single-cycle execution
 - ♦ Fixed instruction length
 - ♦ Dedicated load and store instructions
- ♦ ARM features XN (eXecute **N**ever) Bit

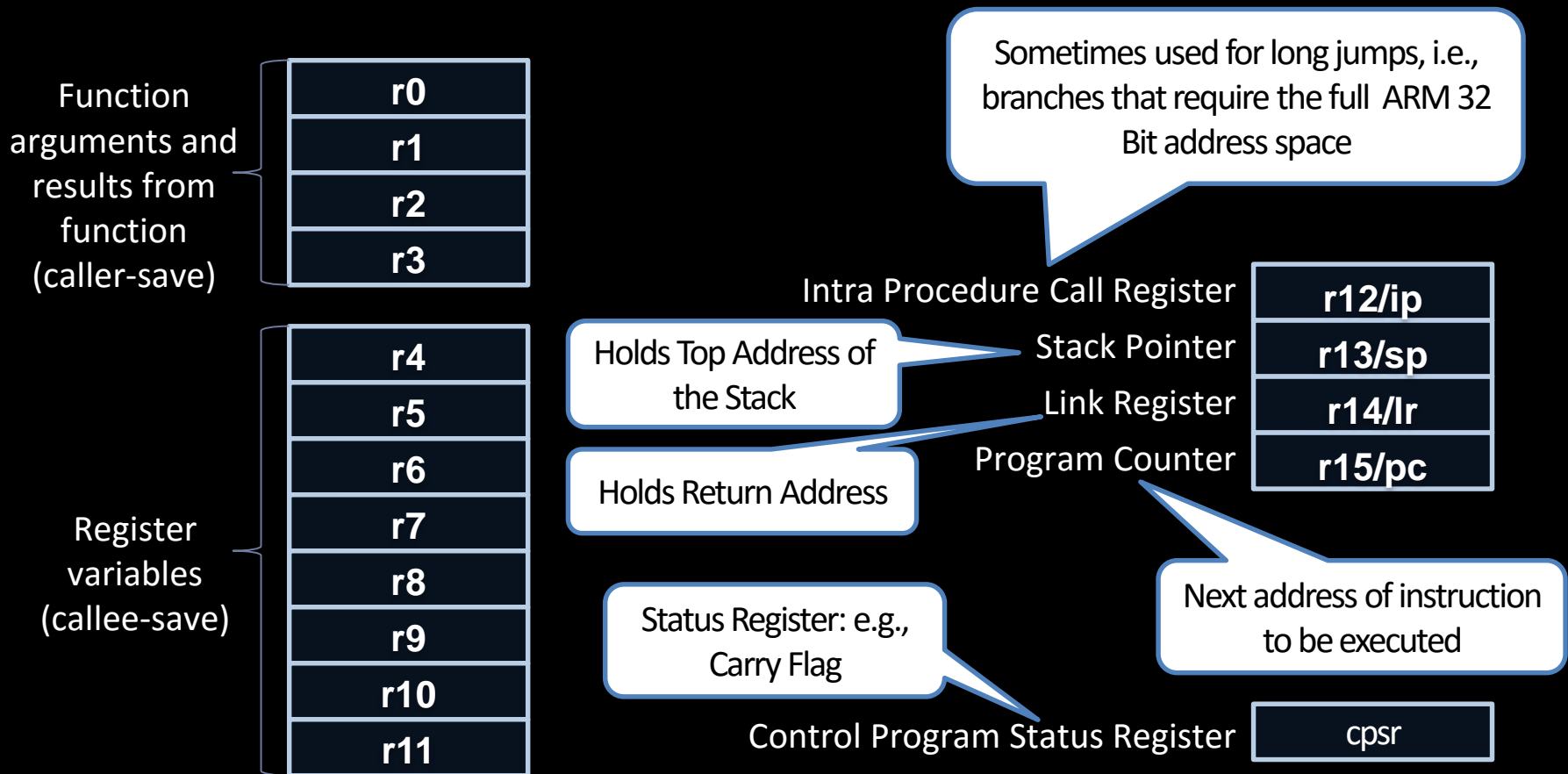
ARM Overview

- ◆ Some features of ARM
 - ◆ Conditional Execution
 - ◆ Two Instruction Sets
 - ◆ **ARM (32-Bit)**
 - ◆ The traditional instruction set
 - ◆ **THUMB (16-Bit)**
 - ◆ Suitable for devices that provide limited memory space
 - ◆ The processor can **exchange** the instruction set on-the-fly
 - ◆ Both instruction sets may occur in a **single** program
 - ◆ 3-Register-Instruction Set
 - ◆ **instruction** *destination, source, source*

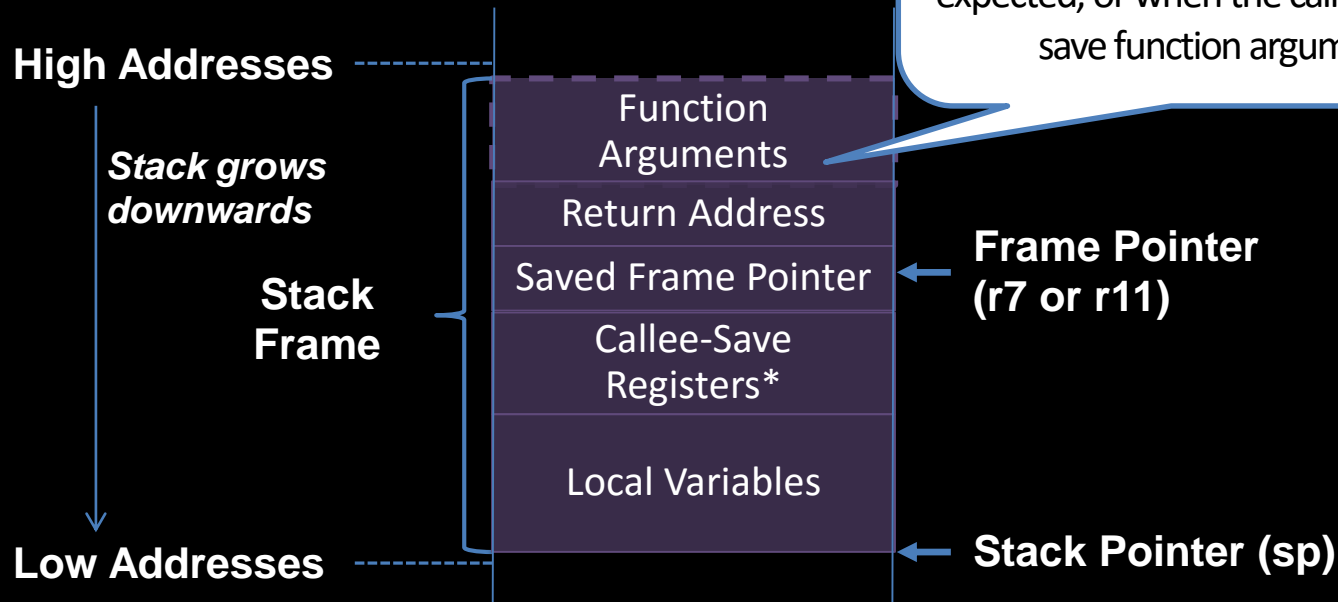


ARM Registers

- ARM's 32 Bit processor features 16 registers
- All registers r0 to r15 are directly accessible



ARM Stack Layout



* Note that a subroutine does not always store all callee-save registers (r4 to r11); instead it stores those registers that it really uses/changes

The Stack and Stack Frame Elements

- ♦ Stack is a last in, first out (LIFO) memory area where the **Stack Pointer** points to the last stored element on the stack
- ♦ The stack can be accessed by two basic operations
 1. **PUSH** elements onto the stack (SP is decremented)
 2. **POP** elements off the stack (SP is incremented)
- ♦ Stack is divided into individual stack frames
 - ♦ Each function call sets up a new stack frame on top of the stack
 1. **Function arguments**
 - ♦ Arguments provided by the caller of the function
 2. **Callee-save Registers**
 - ♦ Registers that a subroutine (callee) needs to reset before returning to the caller of the subroutine
 3. **Return address**
 - ♦ Upon function return control transfers to the code pointed to by the return address (i.e., control transfers back to the caller of the function)
 4. **Saved Frame Pointer/Saved Base Pointer**
 - ♦ Frame pointer/Base pointer of the calling function
 - ♦ Variables and arguments are accessed via an offset to the frame pointer/base pointer
 - ♦ Provided in register **r11** (ARM code), **r7** (THUMB code), or **EBP** (x86 code)
 5. **Local variables**
 - ♦ Variables that the called function uses internally

Function Calls on ARM

Branch with Link

BL addr

- ◆ Branches to **addr**, and stores the return address in link register **lr/r14**
- ◆ The return address is simply the address that follows the **BL** instruction

Branch with Link and eXchange instruction set

BLX addr|reg

- ◆ Branches to **addr|reg**, and stores the return address in **lr/r14**
- ◆ This instruction allows the **exchange** between ARM and THUMB
 - ◆ ARM->THUMB: LSB=1
 - ◆ THUMB->ARM: LSB=0

Function Returns on ARM

*Branch with eXchange
instruction set*

BX lr

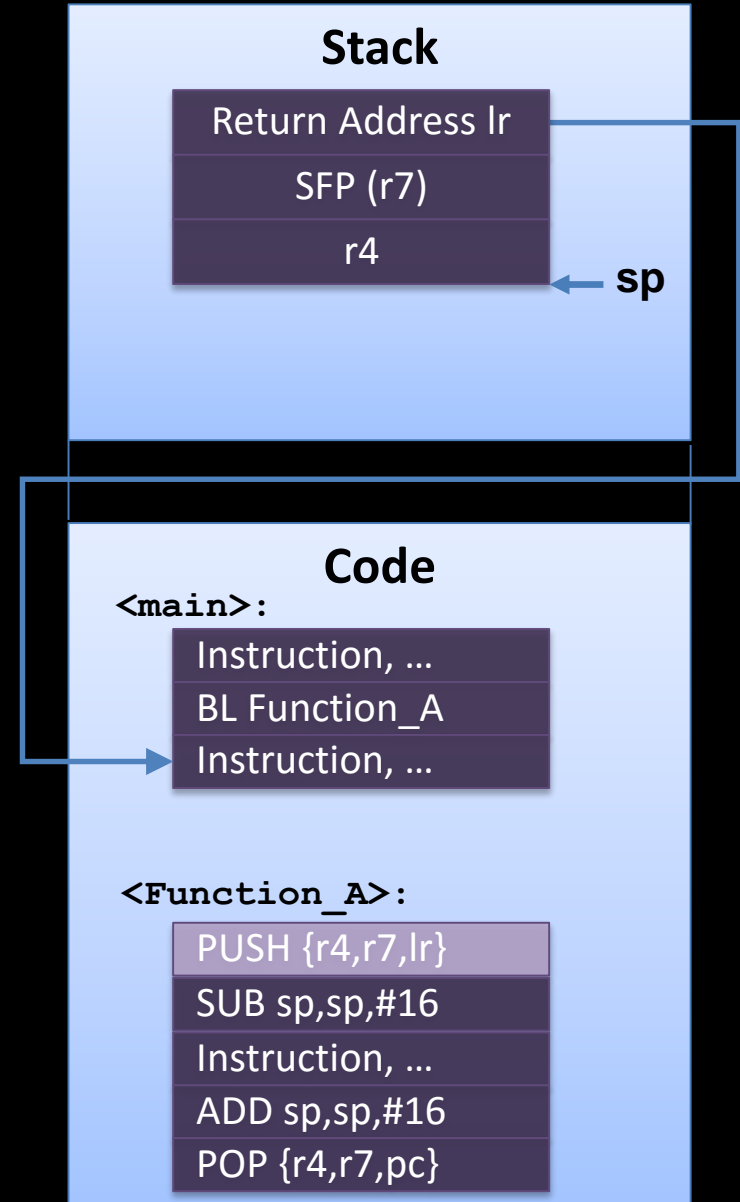
- ◆ Branches to the return address stored in the link register **lr**
- ◆ Register-based return for leaf functions

POP {pc}

- ◆ Pops top of the stack into the program counter **pc/r15**
- ◆ Stack-based return for non-leaf functions

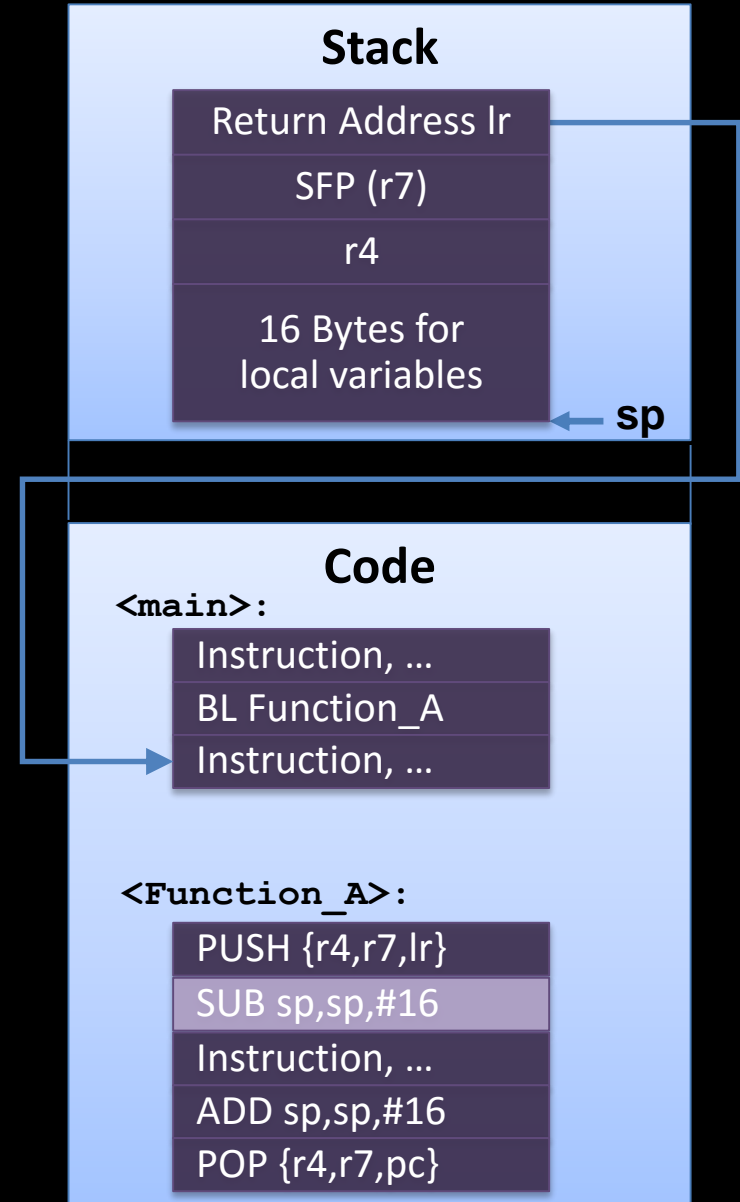
THUMB Example for Calling Convention

- ◆ Function Call: **BL Function_A**
 - ◆ The **BL** instruction automatically loads the return address into the link register **lr**
- ◆ Function Prologue 1: **PUSH {r4,r7,lr}**
 - ◆ Stores callee-save register **r4**, the frame pointer **r7**, and the return address **lr** on the stack
- ◆ Function Prologue 2: **SUB sp,sp,#16**
 - ◆ Allocates **16** Bytes for local variables on the stack
- ◆ Function Body: **Instructions, ...**
- ◆ Function Epilogue 2: **ADD sp,sp,#16**
 - ◆ Reallocates the space for local variables
- ◆ Function Epilogue 2: **POP {r4,r7,pc}**
 - ◆ The **POP** instruction pops the callee-save register **r4**, the saved frame pointer **r7**, and the return address off the stack which is loaded into the program counter **pc**
 - ◆ Hence, the execution will continue in the main function



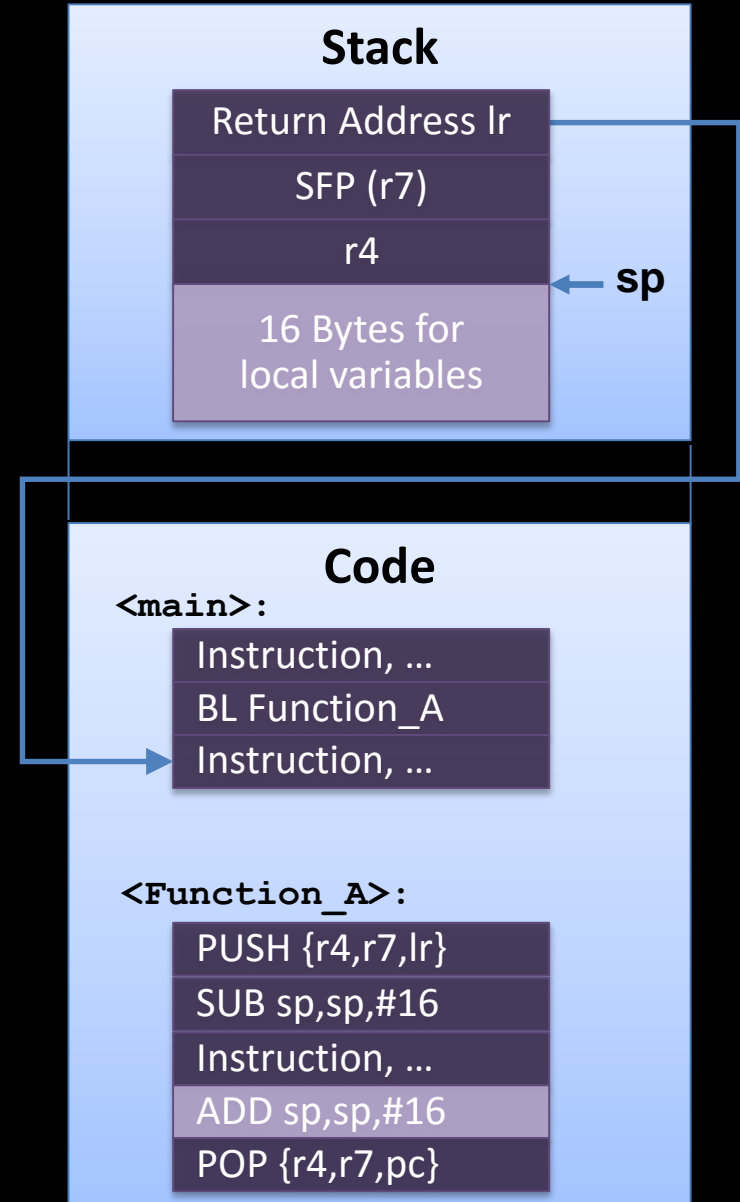
THUMB Example for Calling Convention

- ◆ Function Call: **BL Function_A**
 - ◆ The **BL** instruction automatically loads the return address into the link register **lr**
- ◆ Function Prologue 1: **PUSH {r4,r7,lr}**
 - ◆ Stores callee-save register **r4**, the frame pointer **r7**, and the return address **lr** on the stack
- ◆ Function Prologue 2: **SUB sp,sp,#16**
 - ◆ Allocates **16 Bytes** for local variables on the stack
- ◆ Function Body: **Instructions, ...**
- ◆ Function Epilogue 2: **ADD sp,sp,#16**
 - ◆ Reallocates the space for local variables
- ◆ Function Epilogue 2: **POP {r4,r7,pc}**
 - ◆ The **POP** instruction pops the callee-save register **r4**, the saved frame pointer **r7**, and the return address off the stack which is loaded into the program counter **pc**
 - ◆ Hence, the execution will continue in the main function



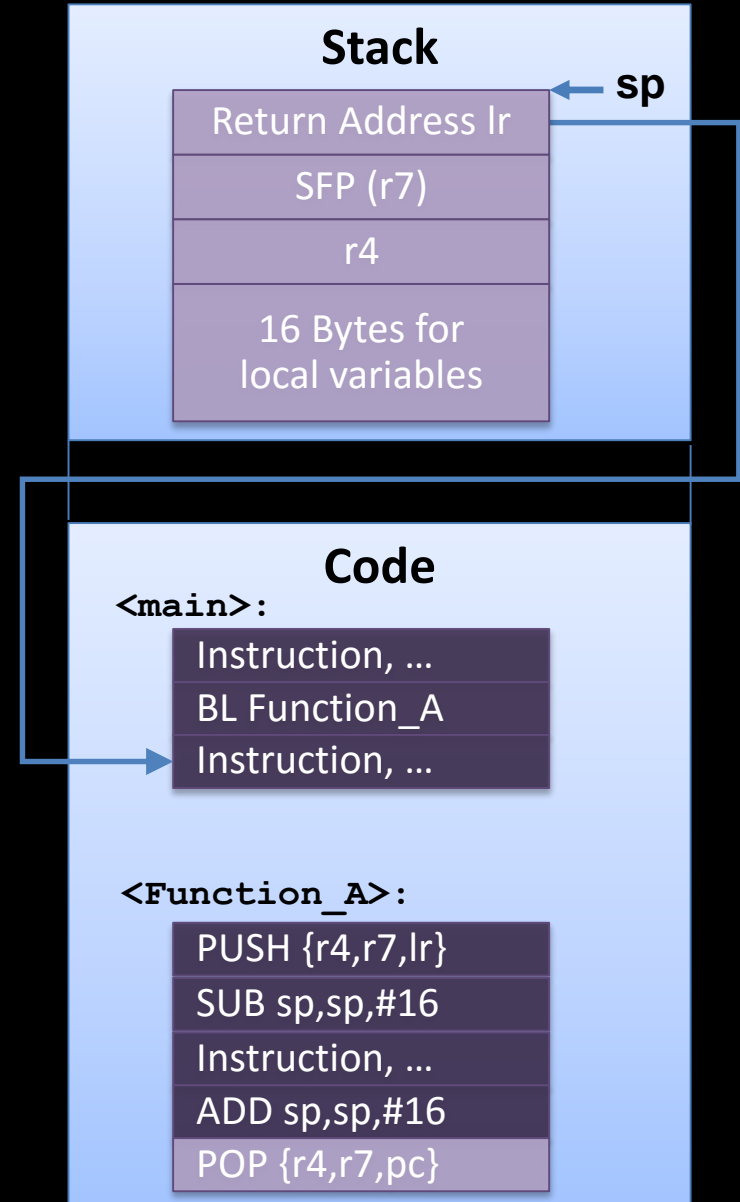
THUMB Example for Calling Convention

- ◆ Function Call: **BL Function_A**
 - ◆ The **BL** instruction automatically loads the return address into the link register **lr**
- ◆ Function Prologue 1: **PUSH {r4,r7,lr}**
 - ◆ Stores callee-save register **r4**, the frame pointer **r7**, and the return address **lr** on the stack
- ◆ Function Prologue 2: **SUB sp,sp,#16**
 - ◆ Allocates **16 Bytes** for local variables on the stack
- ◆ Function Body: **Instructions, ...**
- ◆ Function Epilogue 2: **ADD sp,sp,#16**
 - ◆ Reallocates the space for local variables
- ◆ Function Epilogue 2: **POP {r4,r7,pc}**
 - ◆ The **POP** instruction pops the callee-save register **r4**, the saved frame pointer **r7**, and the return address off the stack which is loaded into the program counter **pc**
 - ◆ Hence, the execution will continue in the main function



THUMB Example for Calling Convention

- ◆ Function Call: **BL Function_A**
 - ◆ The **BL** instruction automatically loads the return address into the link register **lr**
- ◆ Function Prologue 1: **PUSH {r4,r7,lr}**
 - ◆ Stores callee-save register **r4**, the frame pointer **r7**, and the return address **lr** on the stack
- ◆ Function Prologue 2: **SUB sp,sp,#16**
 - ◆ Allocates **16 Bytes** for local variables on the stack
- ◆ Function Body: **Instructions, ...**
- ◆ Function Epilogue 2: **ADD sp,sp,#16**
 - ◆ Reallocates the space for local variables
- ◆ Function Epilogue 2: **POP {r4,r7,pc}**
 - ◆ The **POP** instruction pops the callee-save register **r4**, the saved frame pointer **r7**, and the return address off the stack which is loaded into the program counter **pc**
 - ◆ Hence, the execution will continue in the main function



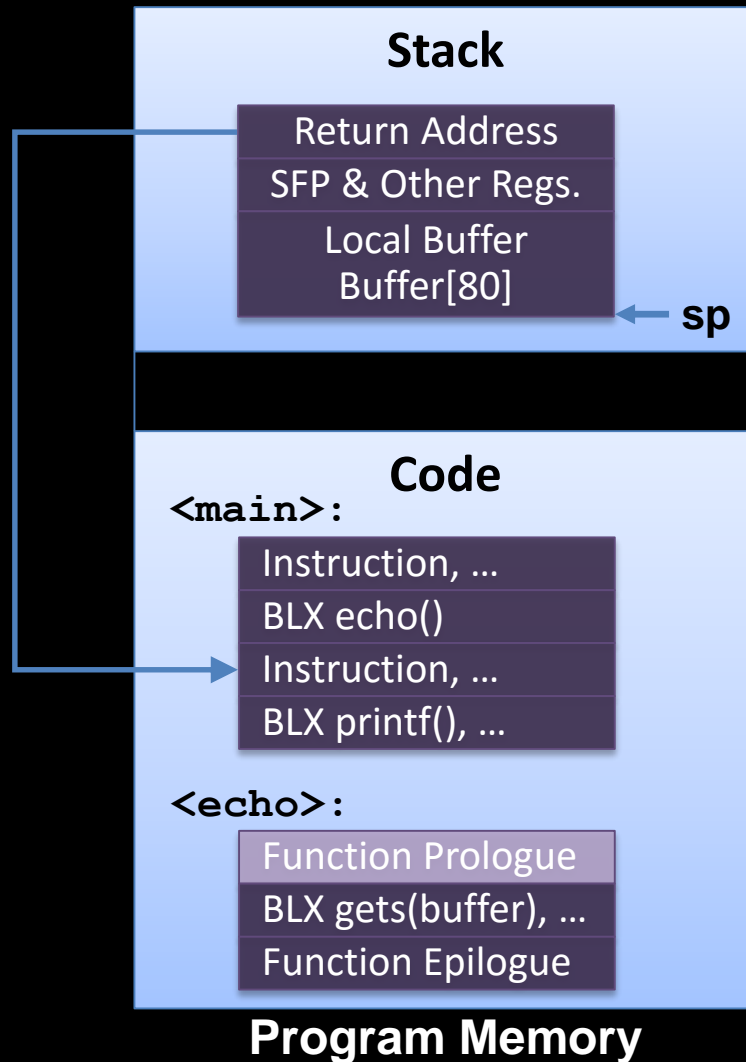
Let's go back to runtime attacks

Running Example

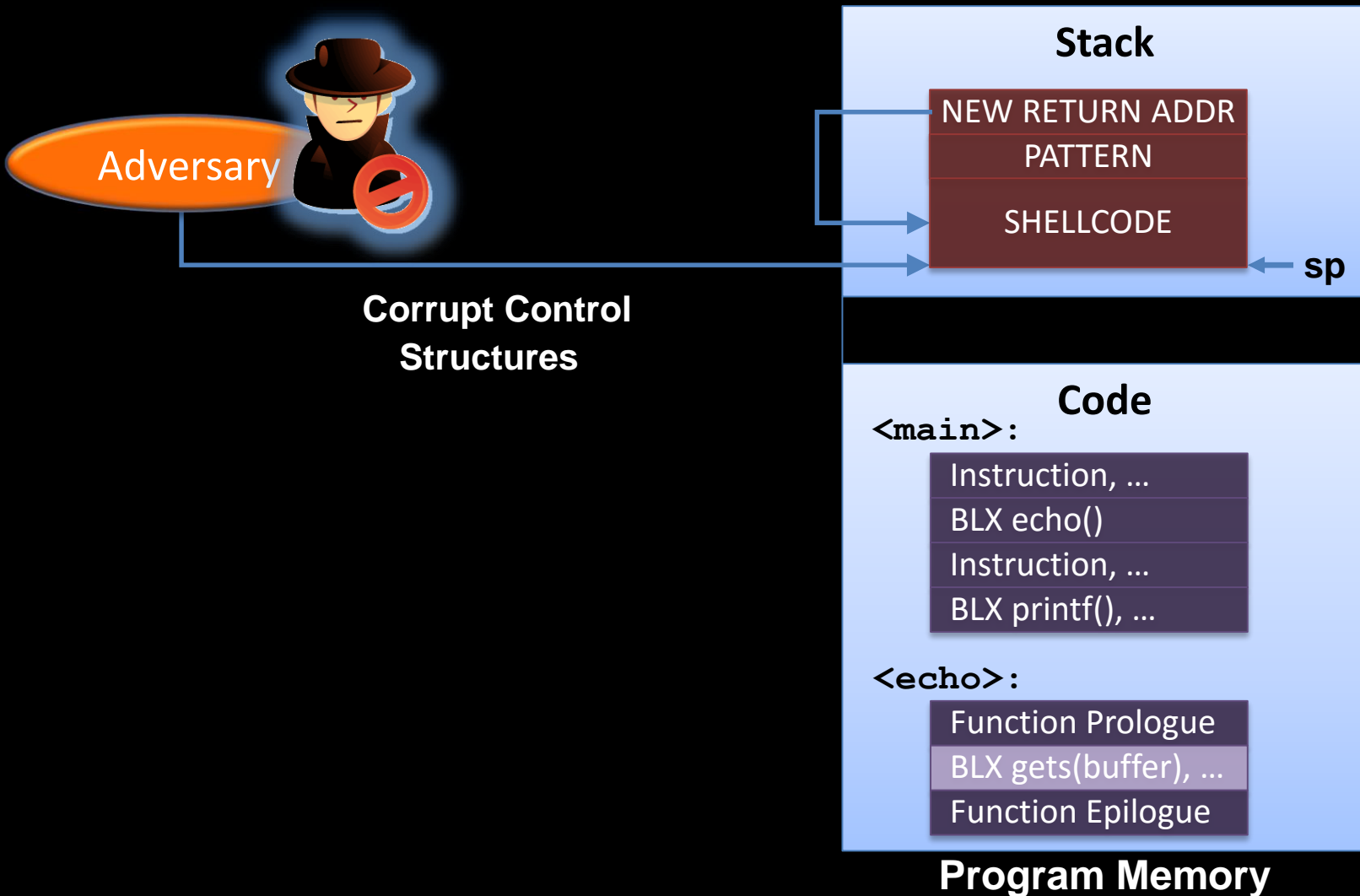
```
#include <stdio.h>
void echo ()
{
    char buffer [80];
    gets (buffer);
    puts (buffer);
}
int main ()
{
    echo ();
    printf (" Done" );
    return 0;
}
```


**Launching a code injection attack
against the vulnerable program**

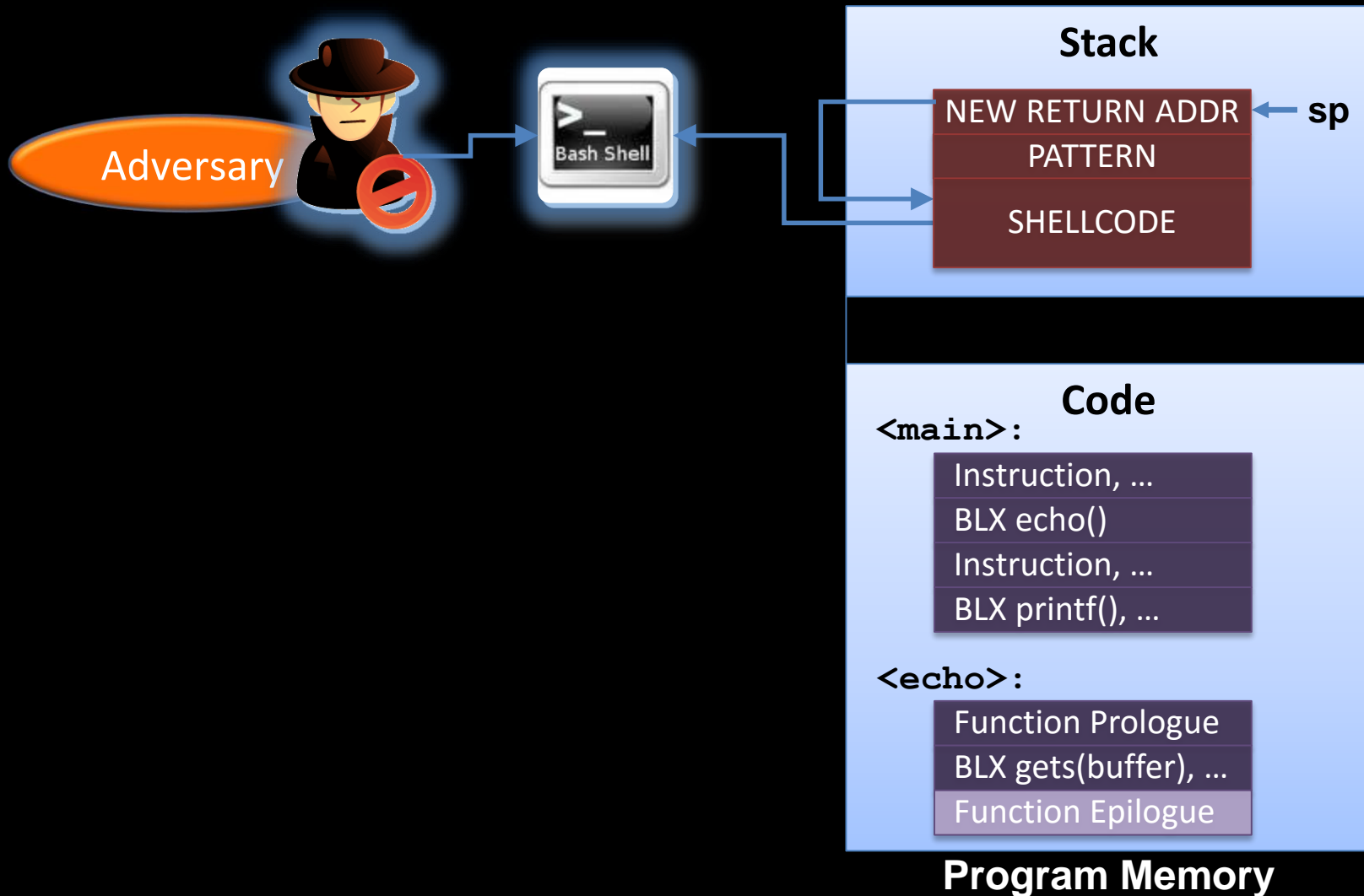
Code Injection Attack on ARM



Code Injection Attack on ARM



Code Injection Attack on ARM



Code-Reuse Attacks

It started with return-into-libc

[Solar Designer, <http://insecure.org/sploits/linux.libc.return.lpr.sploit.html> 1997]

- ◆ Basic idea of return-into-libc
 - ◆ Redirect execution to functions in shared libraries
 - ◆ Main target is UNIX C library libc
 - ◆ Libc is linked to nearly every Unix program
 - ◆ Defines system calls and other basic facilities such as `open()`, `malloc()`, `printf()`, `system()`, `execve()`, etc.
 - ◆ Attack example: `system (“/bin/sh”), exit()`

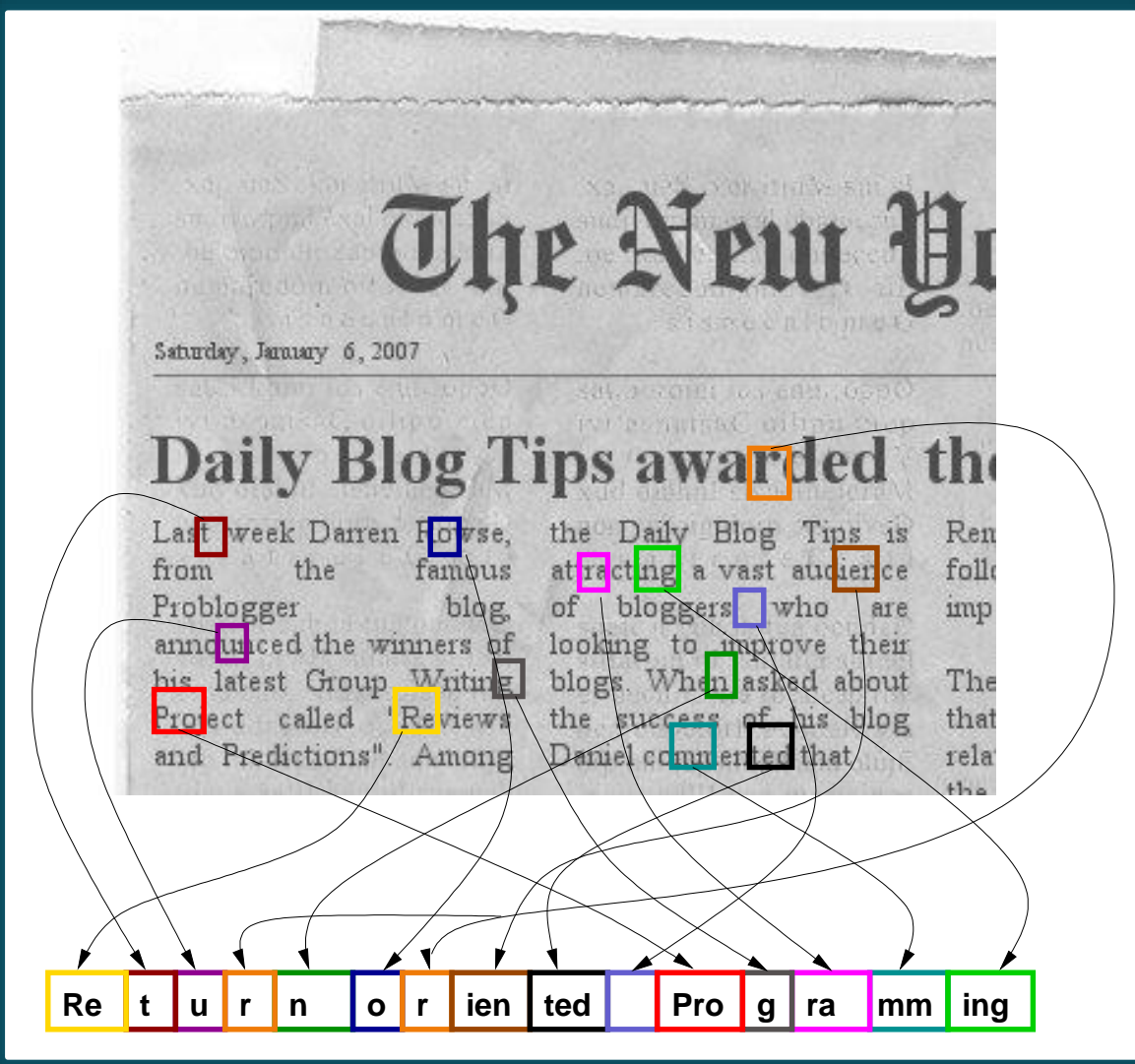
Limitations

- ◆ No branching, i.e., no arbitrary code execution
- ◆ Critical functions can be eliminated or wrapped

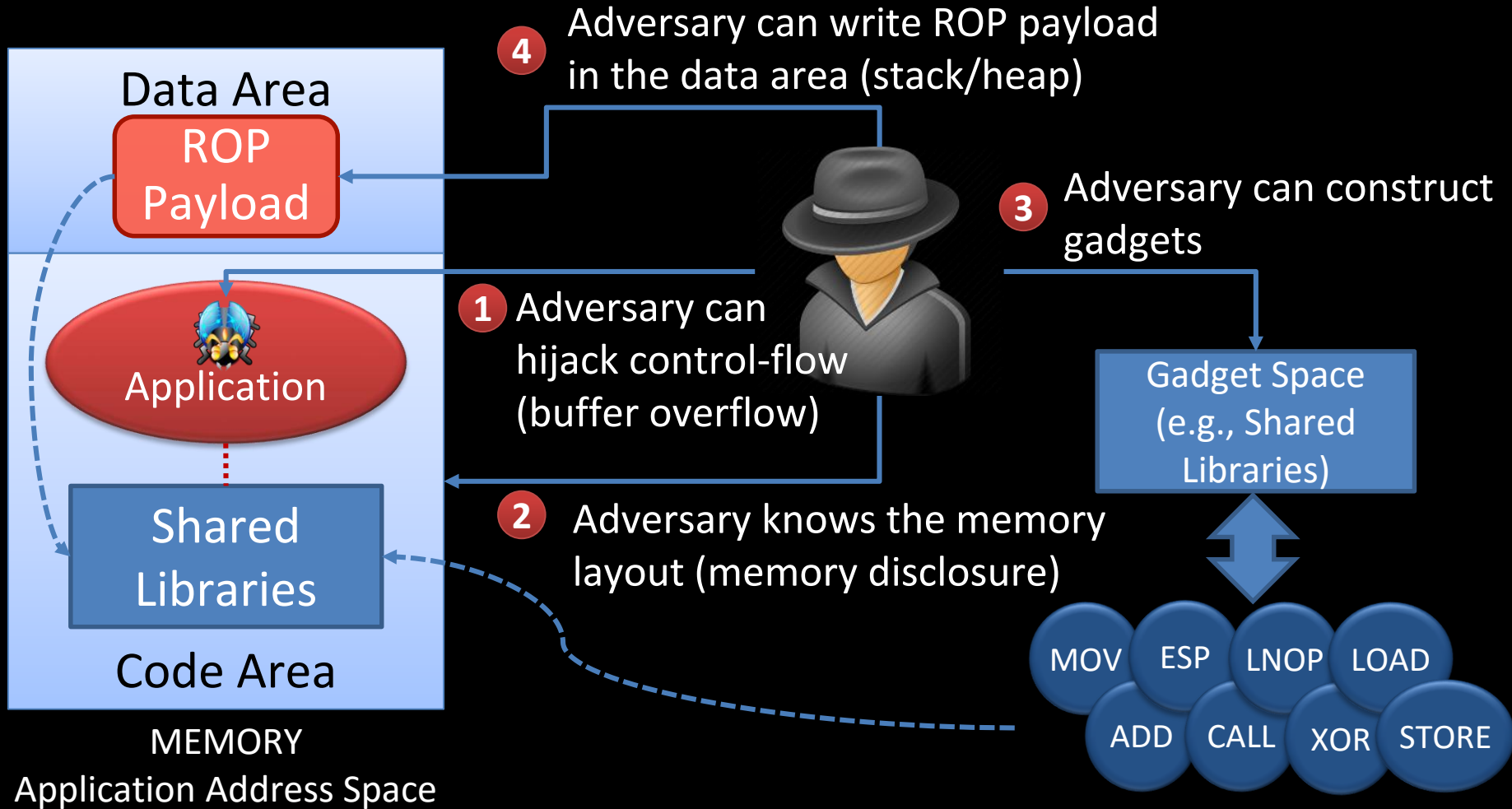
**Generalization of return-into-libc
attacks:**

**return-oriented programming (ROP)
[Shacham, ACM CCS 2007]**

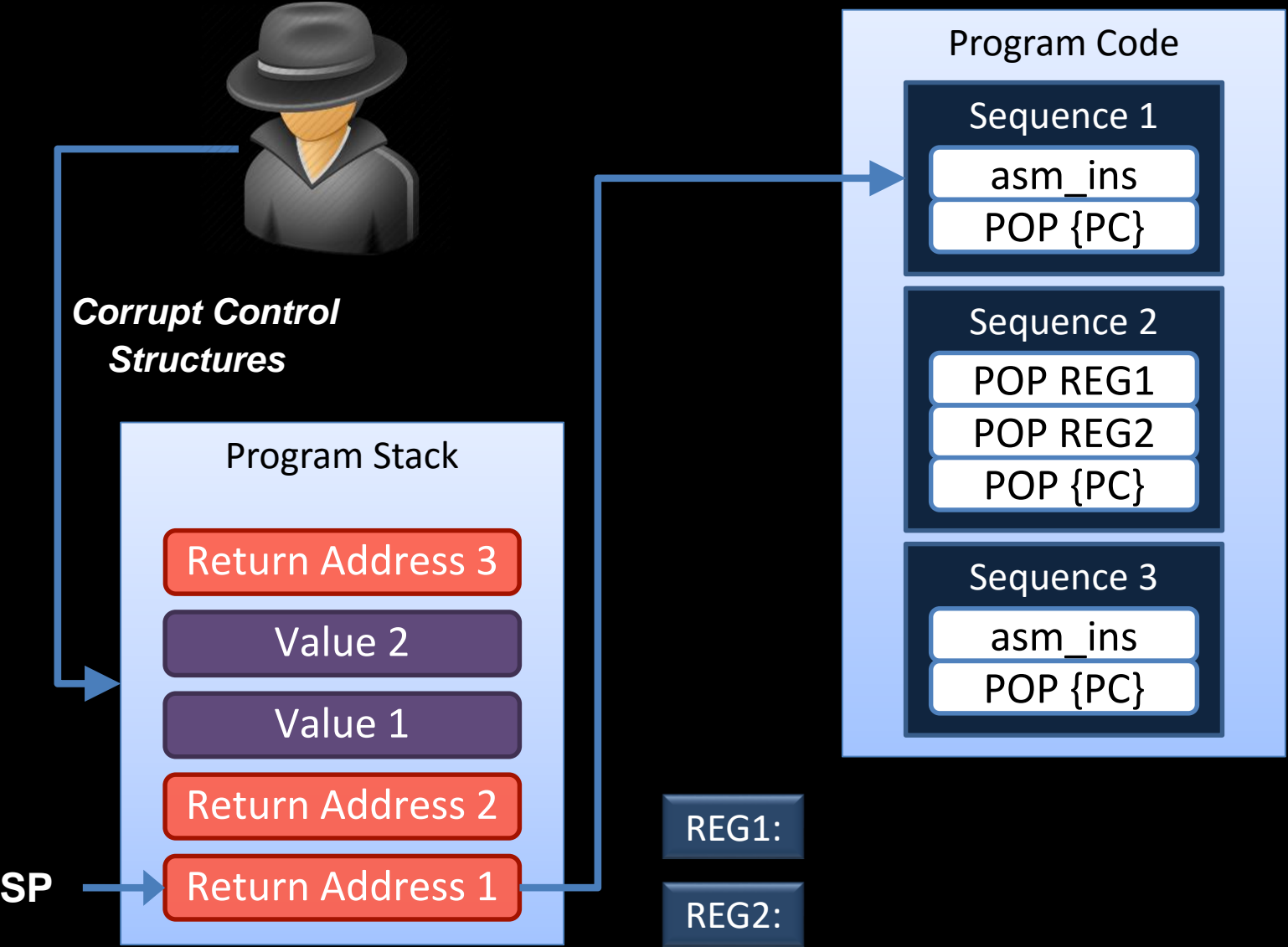
The Big Picture



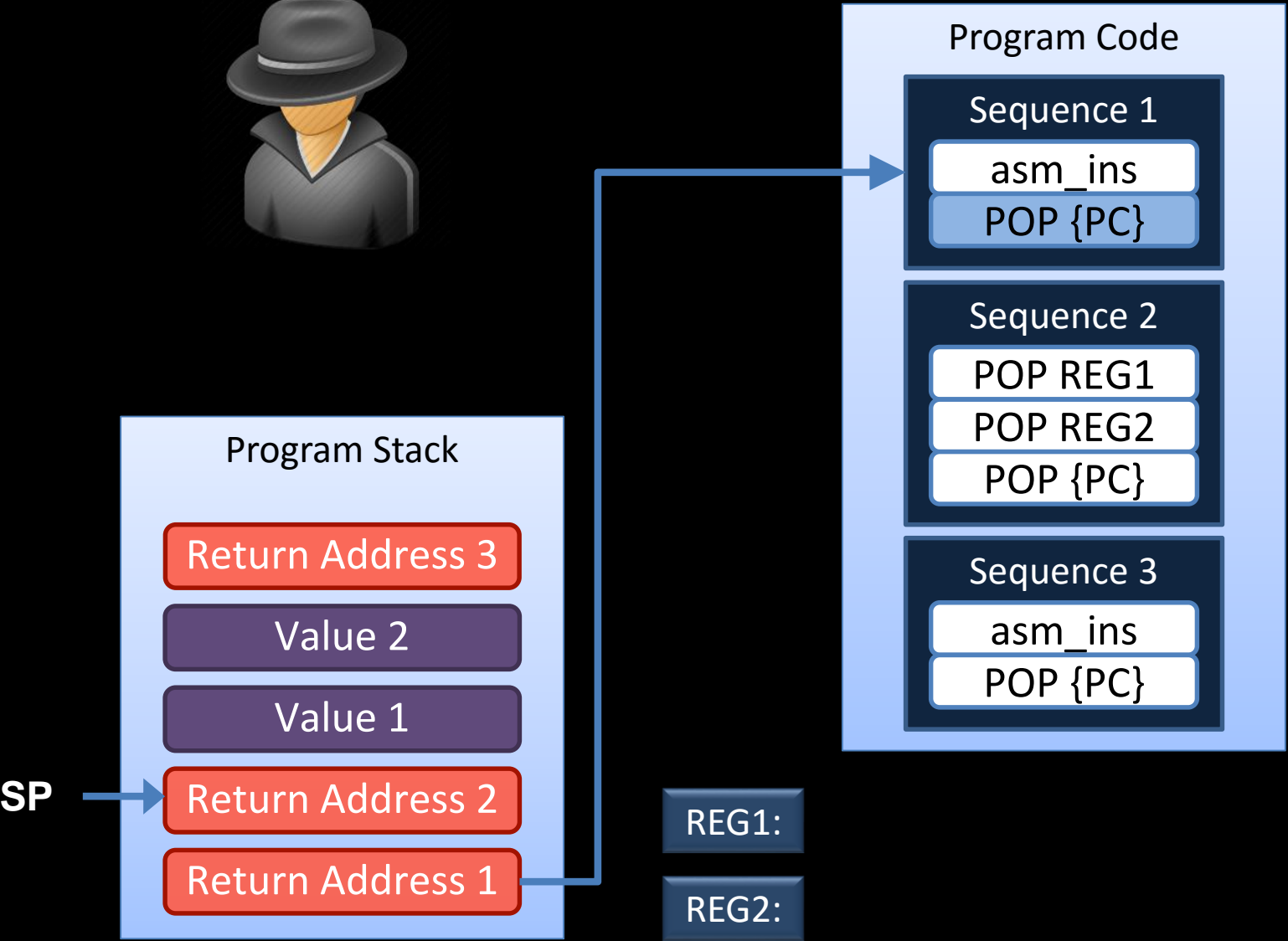
ROP Adversary Model/Assumption



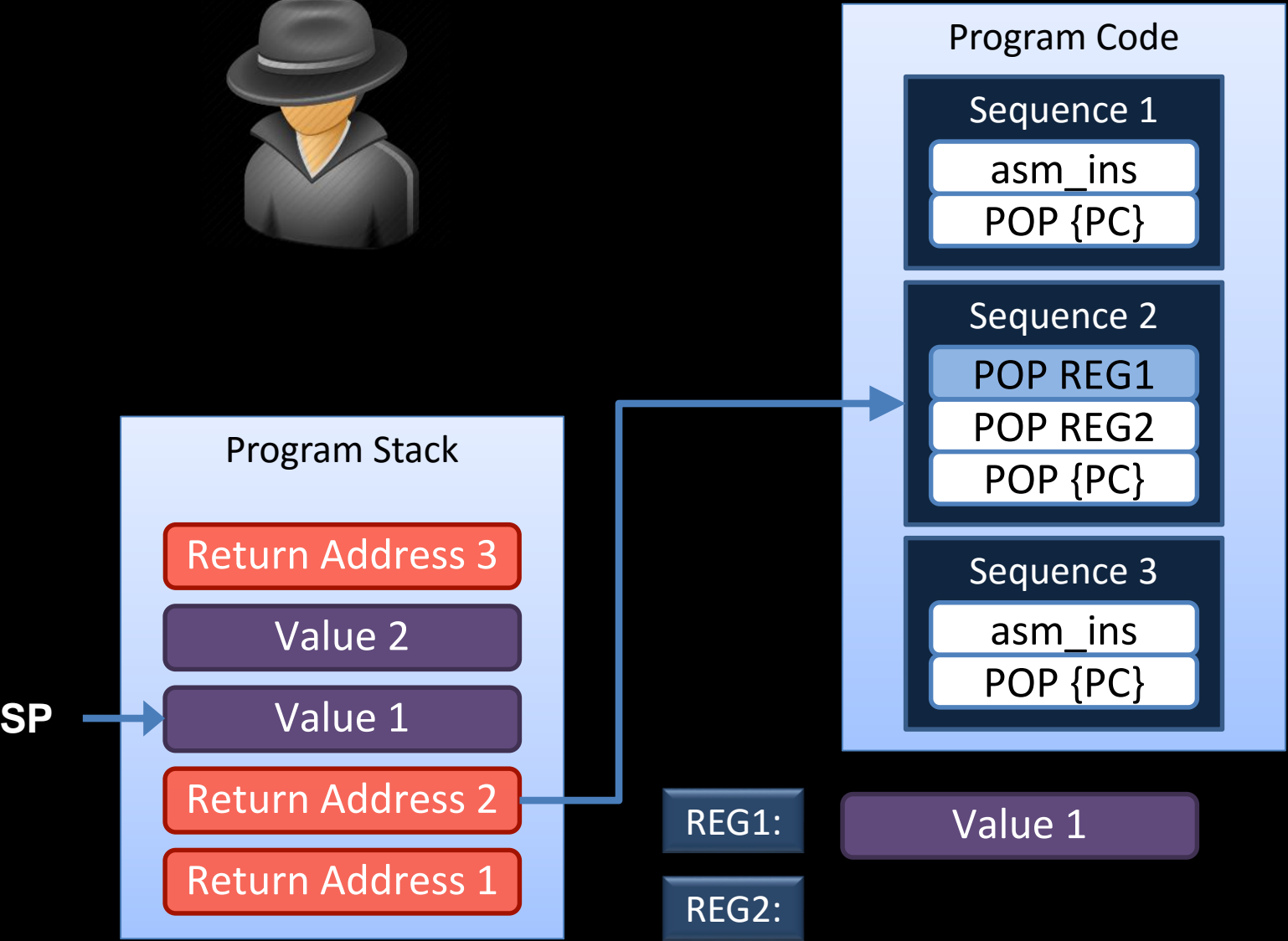
ROP Attack Technique: Overview



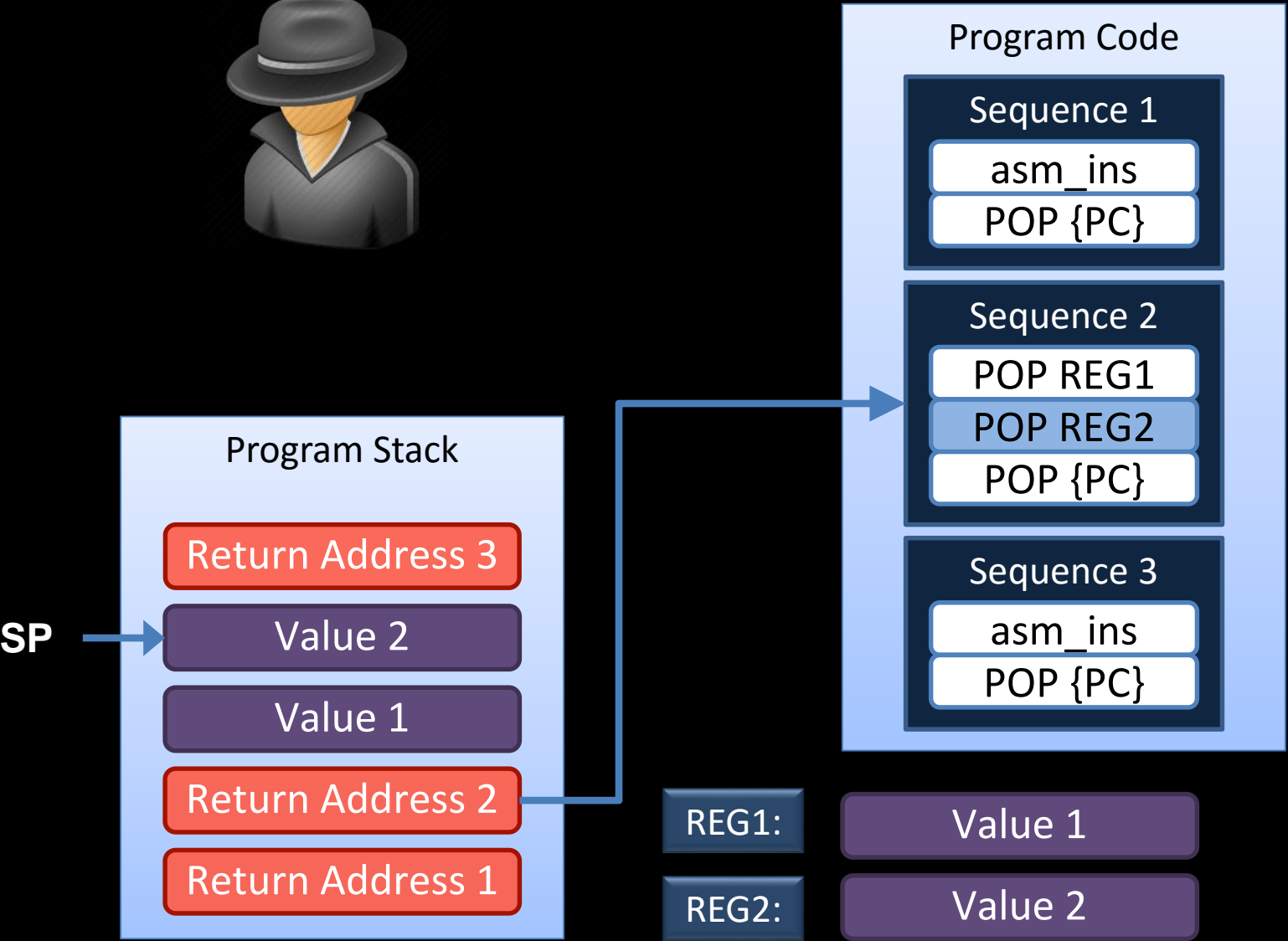
ROP Attack Technique: Overview



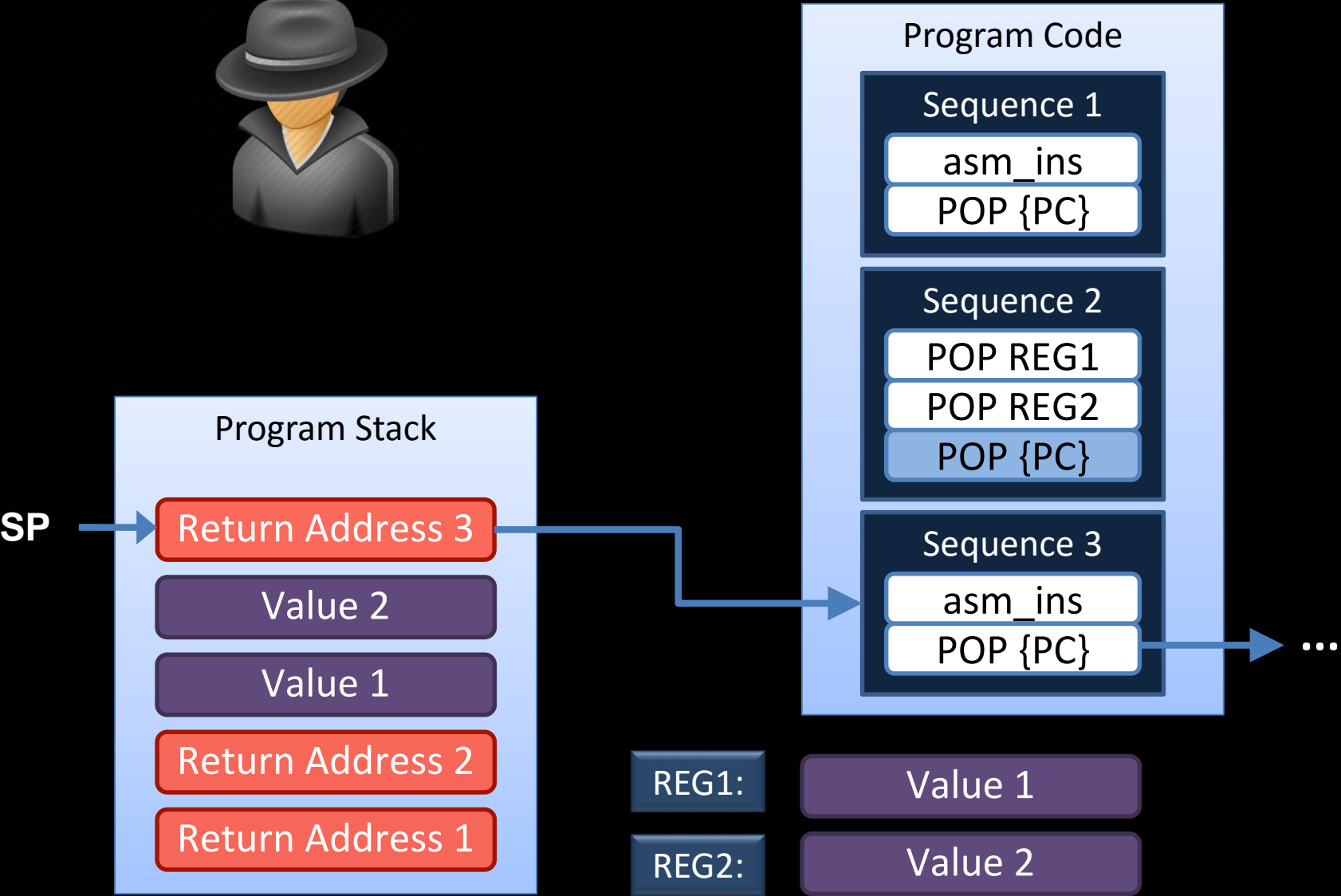
ROP Attack Technique: Overview



ROP Attack Technique: Overview



ROP Attack Technique: Overview

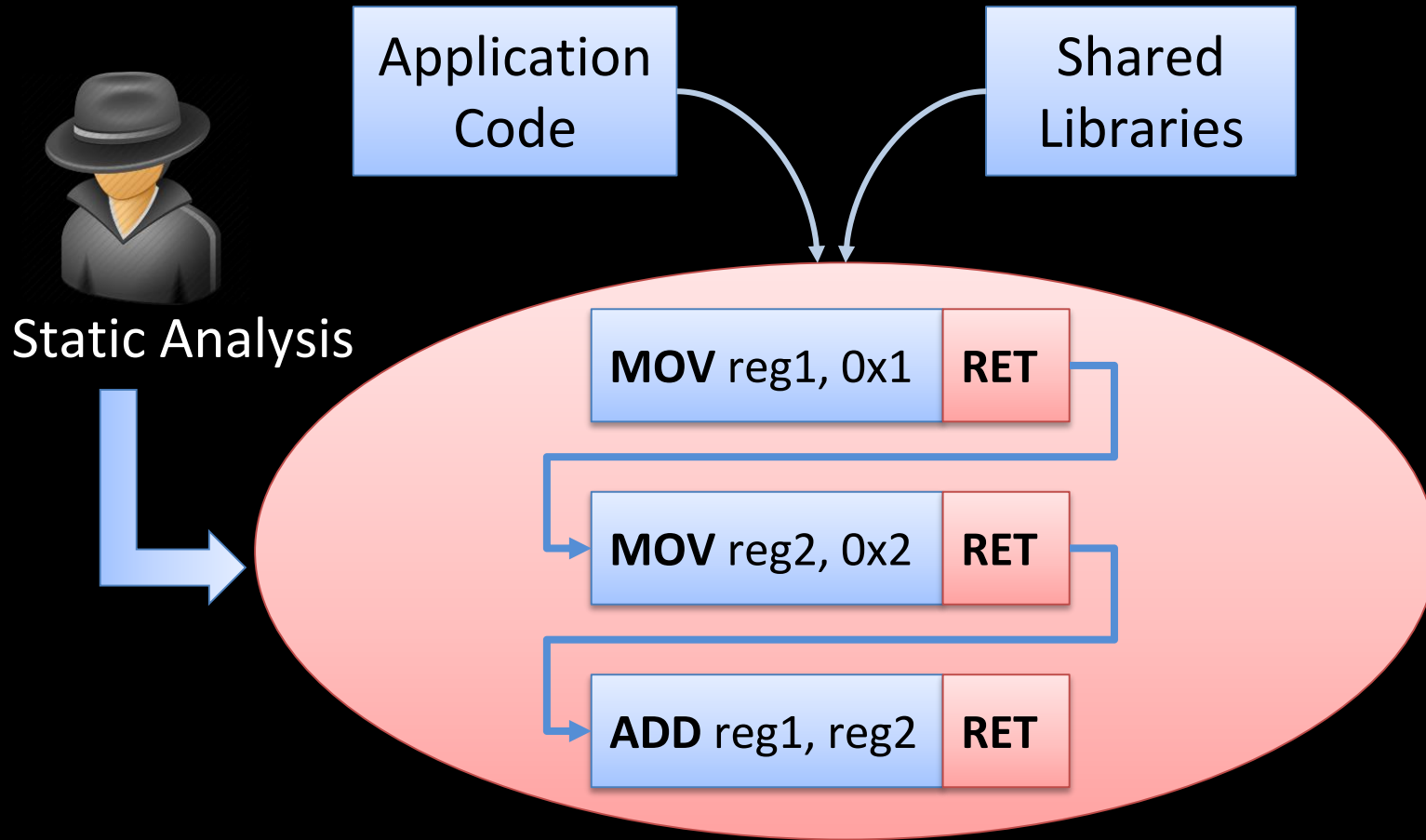


Summary of Basic Idea

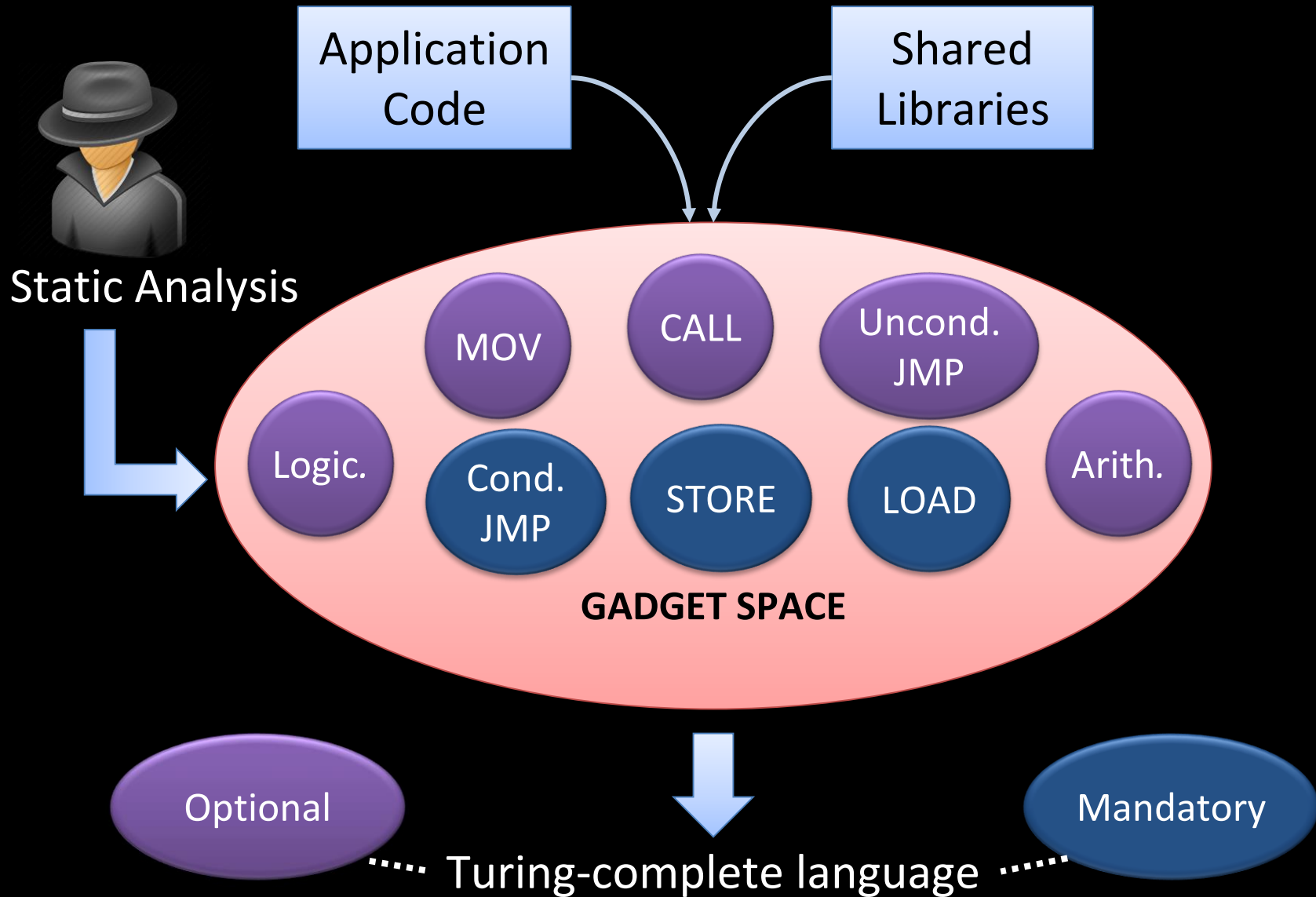
- ◆ Perform arbitrary computation with return-into-libc techniques
- ◆ Approach
 - ◆ Use **small instruction sequences** (e.g., of libc) instead of using whole functions
 - ◆ Instruction sequences range from 2 to 5 instructions
 - ◆ All sequences end with a **return (POP{PC})** instruction
 - ◆ Instruction sequences are chained together to a **gadget**
 - ◆ A gadget performs a particular task (e.g., load, store, xor, or branch)
 - ◆ Afterwards, the adversary enforces his desired actions by combining the gadgets

Special Aspects of ROP

Code Base and Turing-Completeness



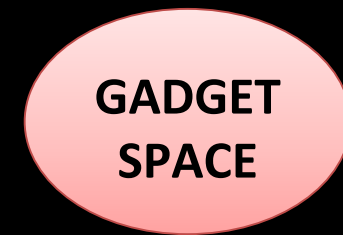
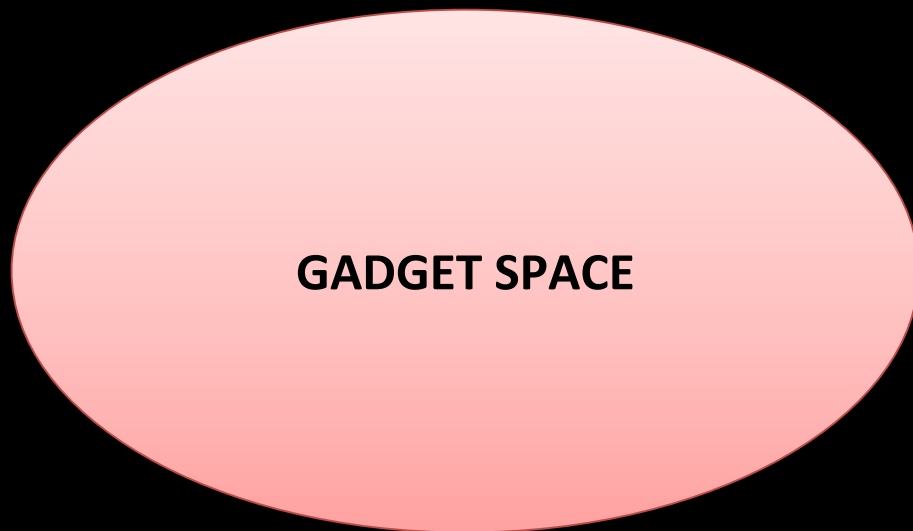
Code Base and Turing-Completeness



Gadget Space on Different Architectures

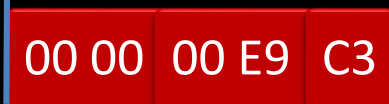
Architectures with no memory alignment, e.g., Intel x86

Architectures with memory alignment, e.g., SPARC, ARM



Intended Code

```
mov $0x13,%eax  
jmp 3aae9
```



Unintended Code

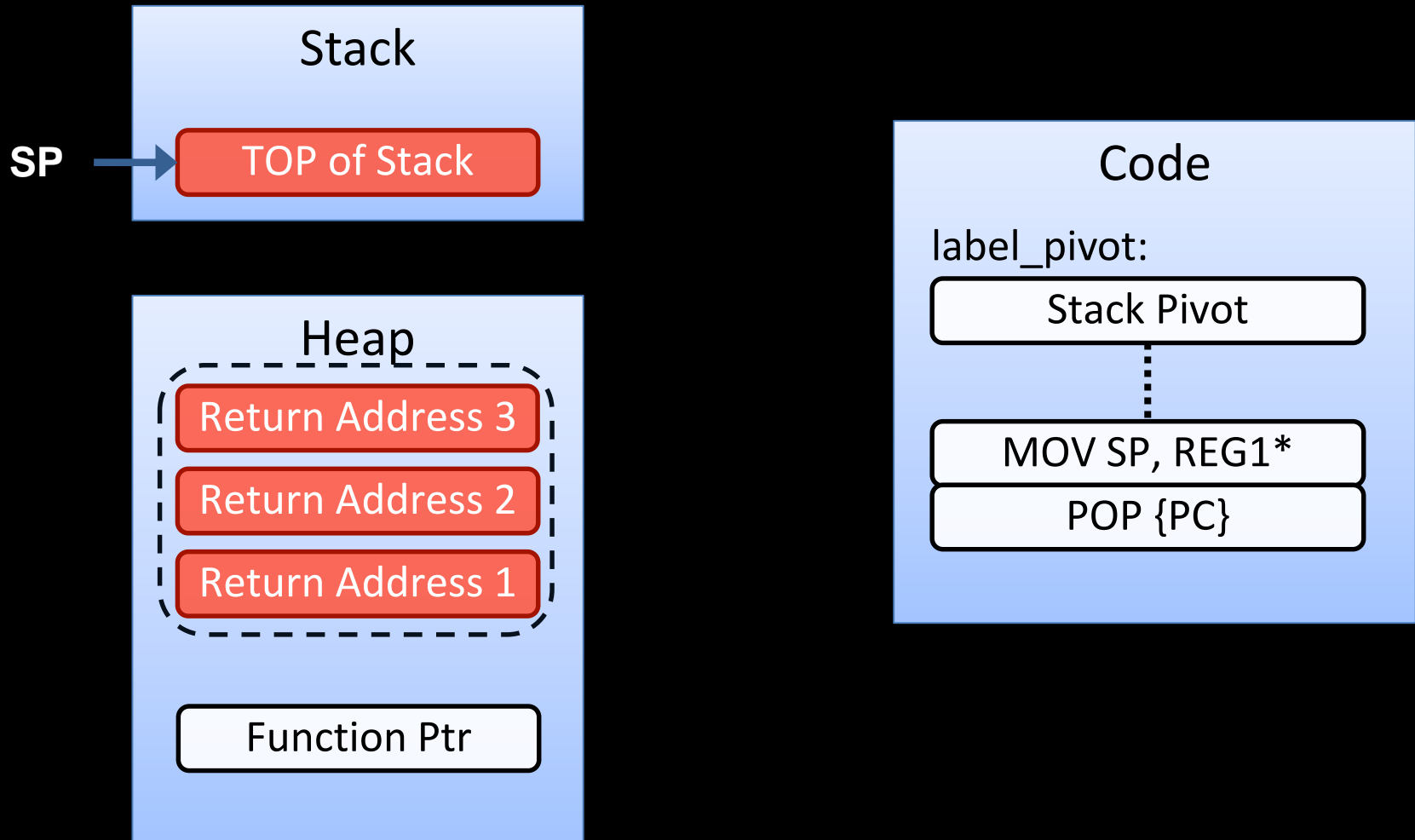
```
add %a1, (%eax)  
add %ch,%c1  
ret
```

Stack Pivot

[Zovi, RSA Conference 2010]

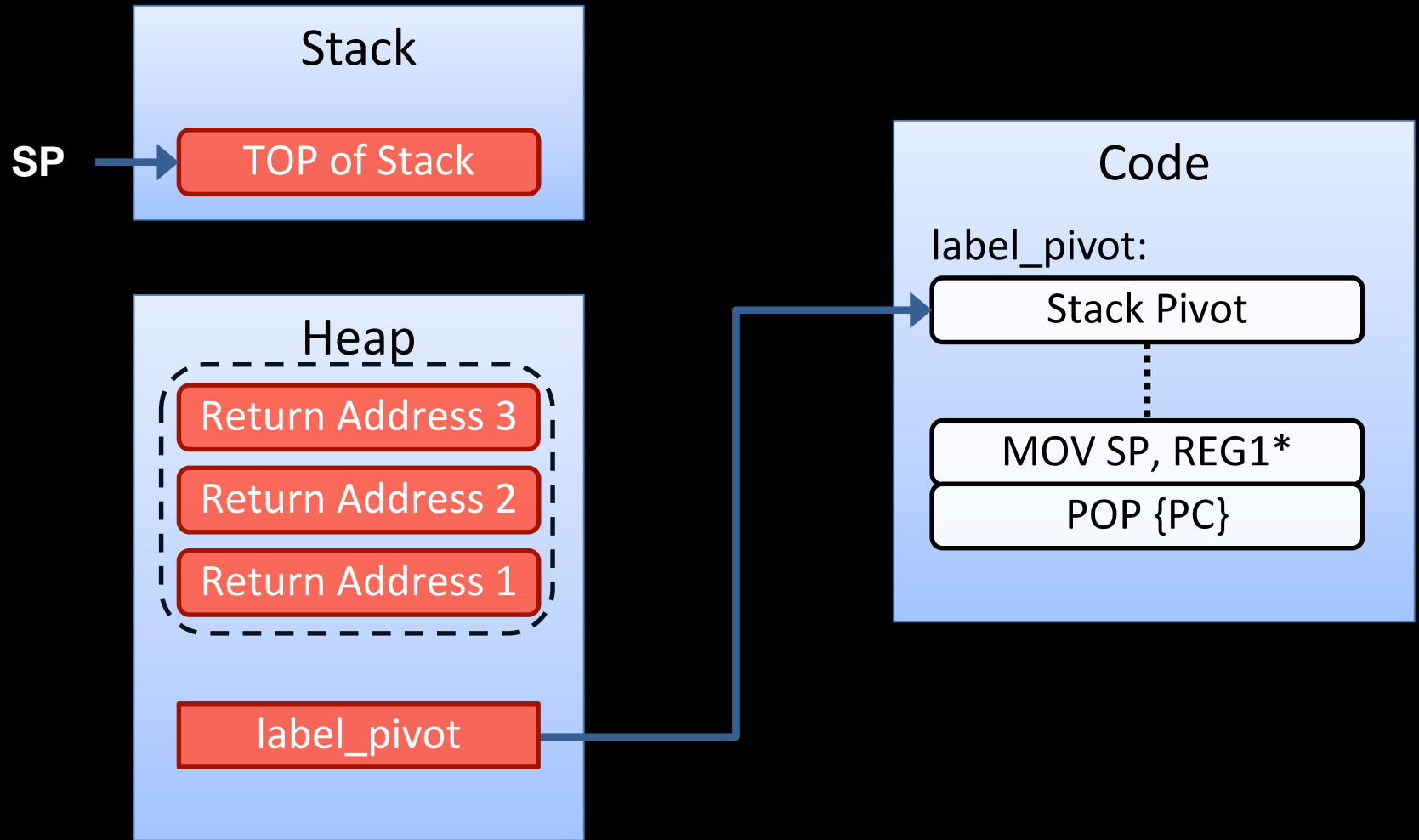
- ♦ Stack pointer plays an important role
 - ♦ It operates as an instruction pointer in ROP attacks
- ♦ Challenge
 - ♦ In order to launch a ROP exploit based on a heap overflow, we need to set the stack pointer to point to the heap
 - ♦ This is achieved by a **stack pivot**

Stack Pivot in Detail



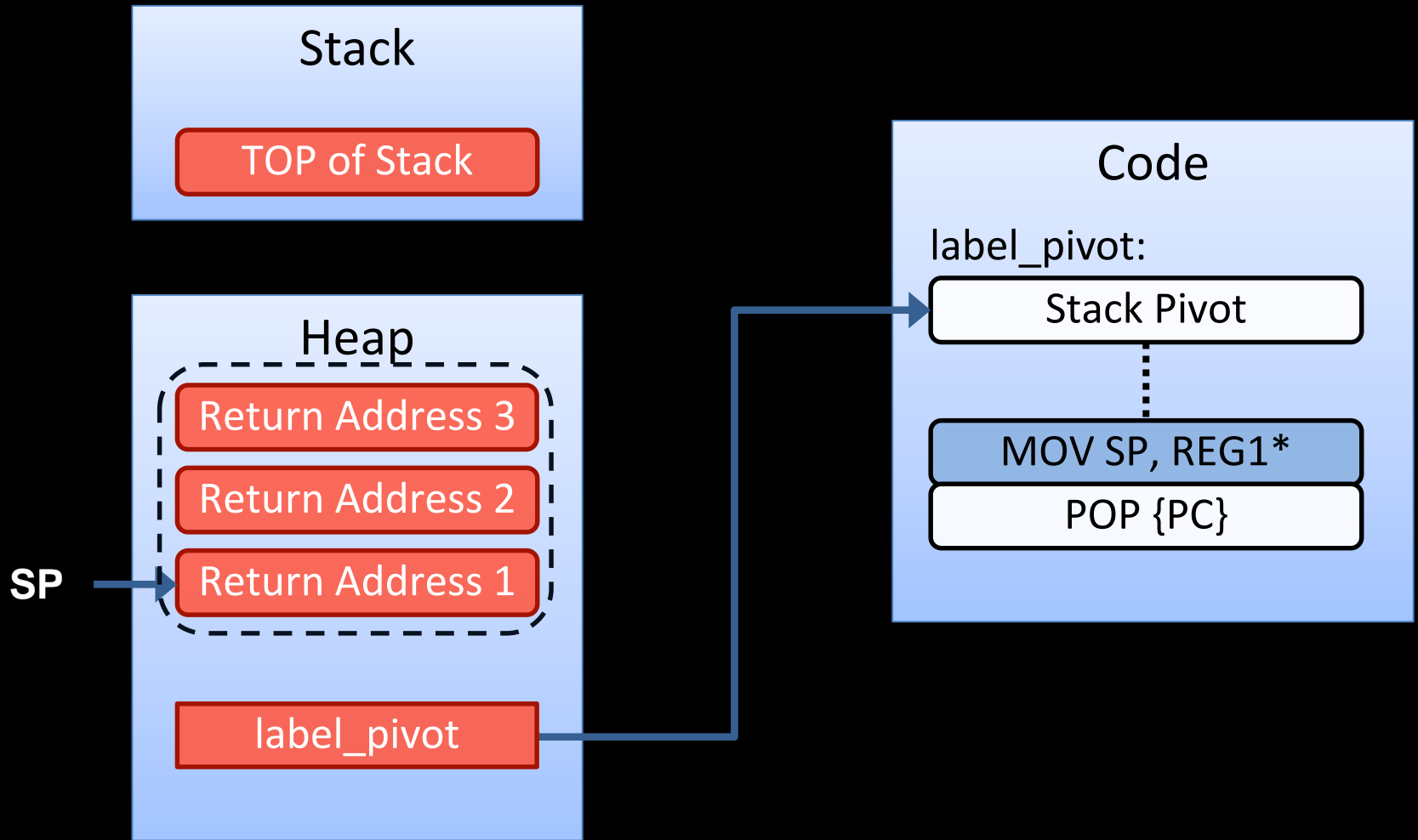
*REG1 is controlled by the adversary and holds beginning of ROP payload

Stack Pivot in Detail



*REG1 is controlled by the adversary and holds beginning of ROP payload

Stack Pivot in Detail



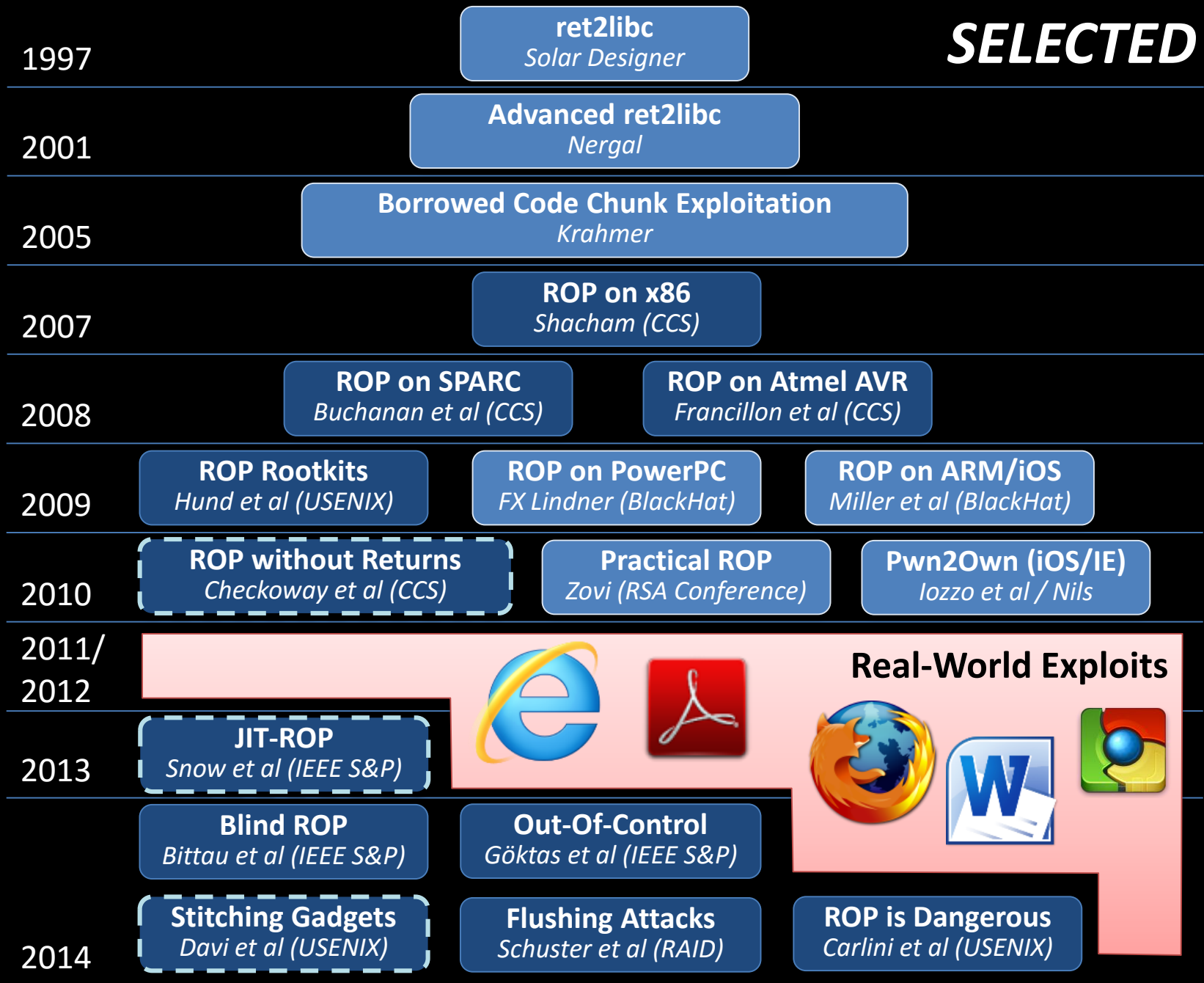
*REG1 is controlled by the adversary and holds beginning of ROP payload

ROP Variants

- ♦ Motivation: return address protection (shadow stack)
 - ♦ Validate every return (intended and unintended) against valid copies of return addresses
[Davi et al., AsiaCCS 2011]
- ♦ Exploit indirect jumps and calls
 - ♦ ROP without returns
[Checkoway et al., ACM CCS 2010]

CURRENT RESEARCH

SELECTED



Our Work & Involvement

♦ Attacks

- ♦ Return-Oriented Programming without Returns [CCS 2010]
- ♦ Privilege Escalation Attacks on Android [ISC 2010]
- ♦ Just-In-Time Return-oriented Programming (JIT-ROP) [IEEE S&P 2013, Best Student Paper] & [BlackHat USA 2013]
- ♦ Stitching the Gadgets [USENIX Security 2014] & [BlackHat USA 2014]
- ♦ COOP [IEEE Security & Privacy 2015]
- ♦ Losing Control [CCS 2015]

♦ Detection & Prevention

- ♦ ROPdefender [AsiaCCS 2011]
- ♦ Mobile Control-Flow Integrity (MoCFI) [NDSS 2012]
- ♦ XIFER: Fine-Grained ASLR [AsiaCCS 2013]
- ♦ Filtering ROP Payloads [RAID 2013]
- ♦ Isomeron [NDSS 2015]
- ♦ Readactor [IEEE Security & Privacy 2015, CCS 2015]
- ♦ HAFIX: Fine-Grained CFI in Hardware [DAC 2014, DAC 2015, DAC 2016]
- ♦ Readactor++ [CCS 2015]

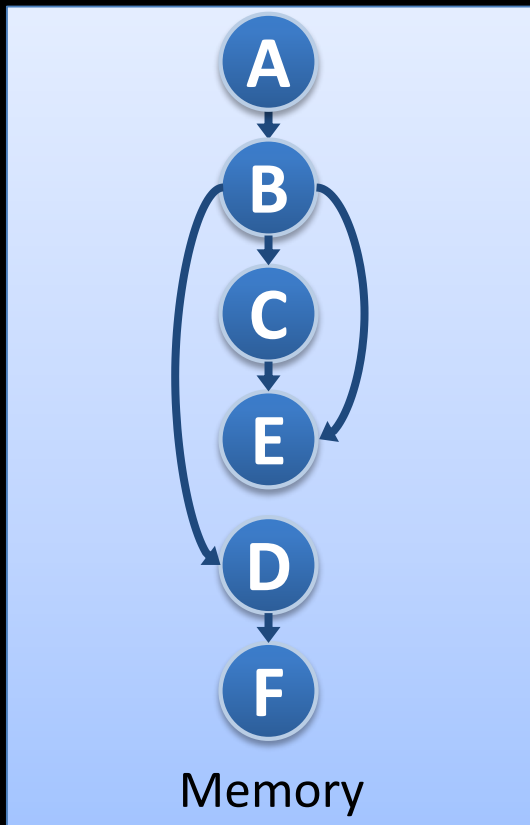


In this tutorial

Main Defense Techniques

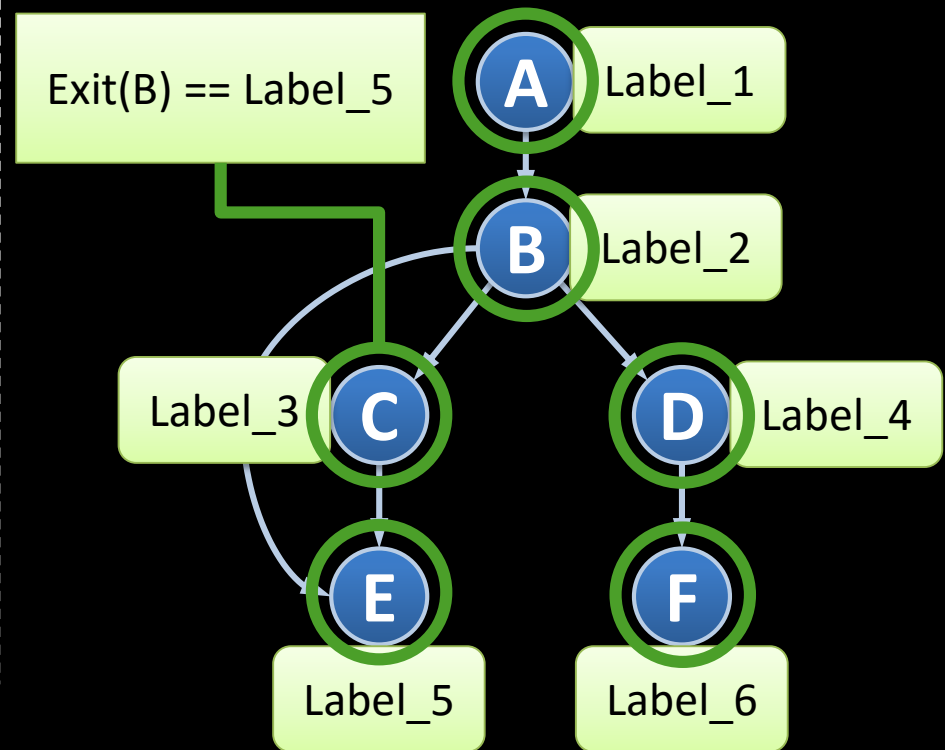
(Fine-grained) Code Randomization

[Cohen 1993 & Larsen et al., SoK IEEE S&P 2014]



Control-Flow Integrity (CFI)

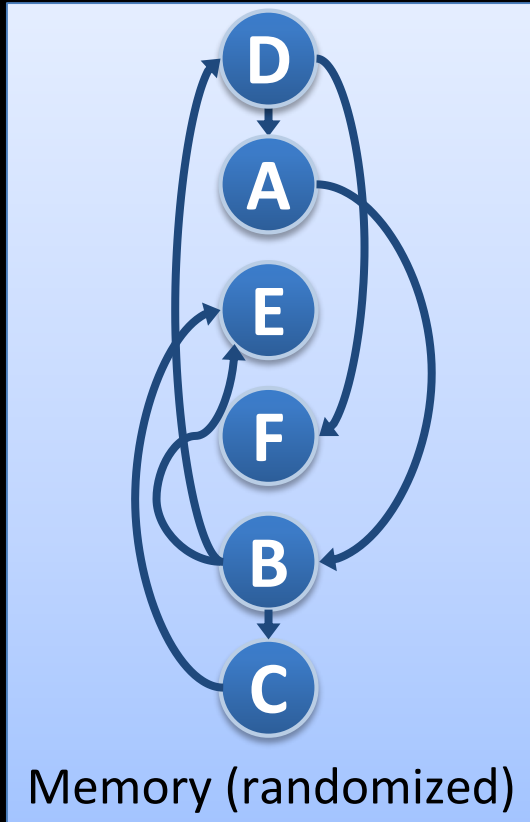
[Abadi et al., CCS 2005 & TISSEC 2009]



Main Defense Techniques

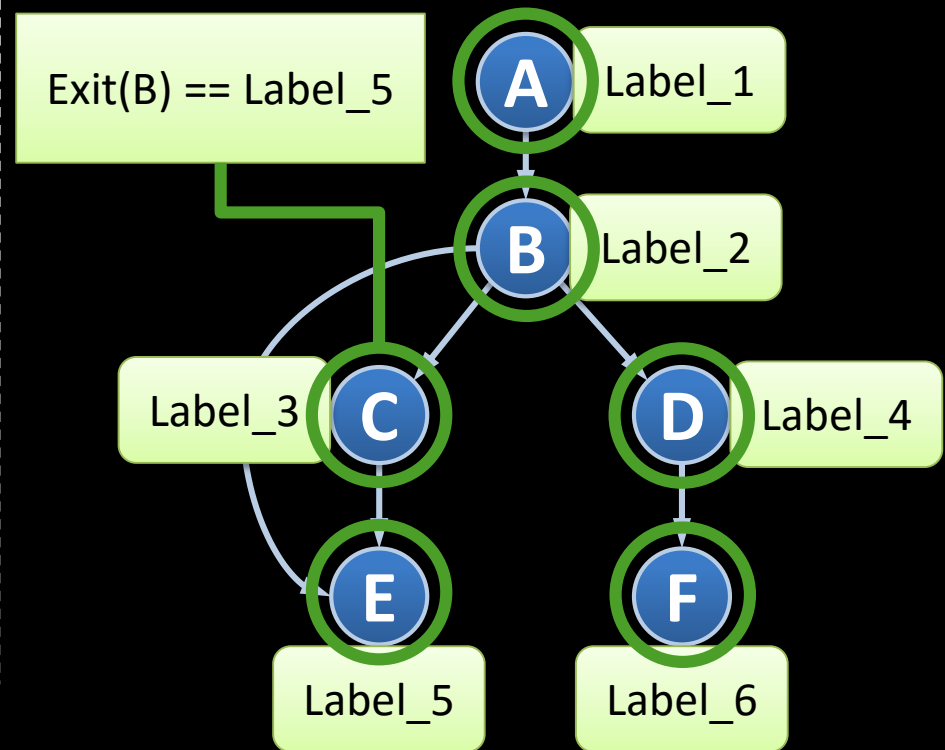
(Fine-grained) Code Randomization

[Cohen 1993 & Larsen et al., SoK IEEE S&P 2014]



Control-Flow Integrity (CFI)

[Abadi et al., CCS 2005 & TISSEC 2009]

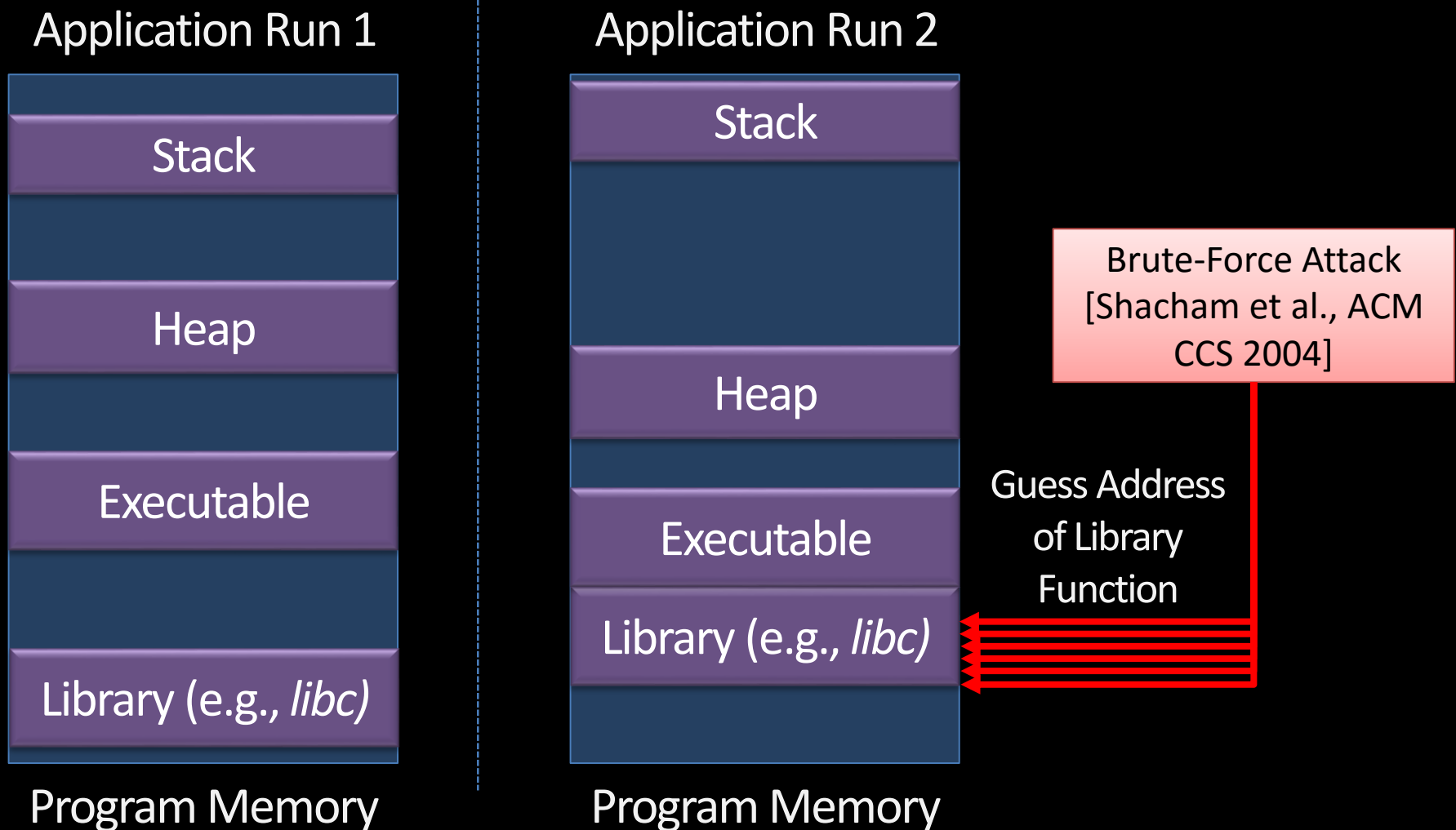


ASLR – Address Space Layout Randomization



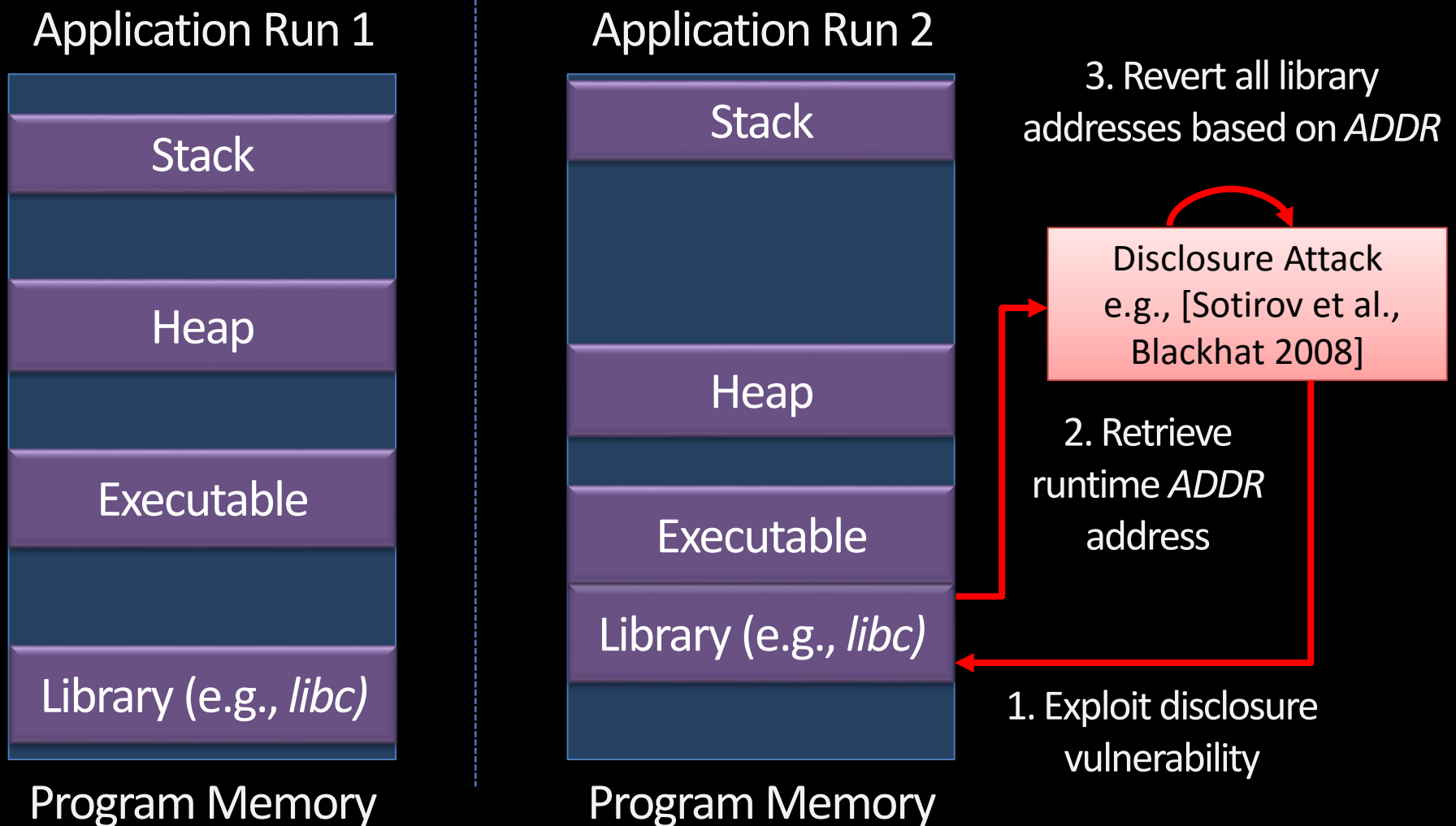
Basics of Memory Randomization

- ASLR randomizes the base address of code/data segments

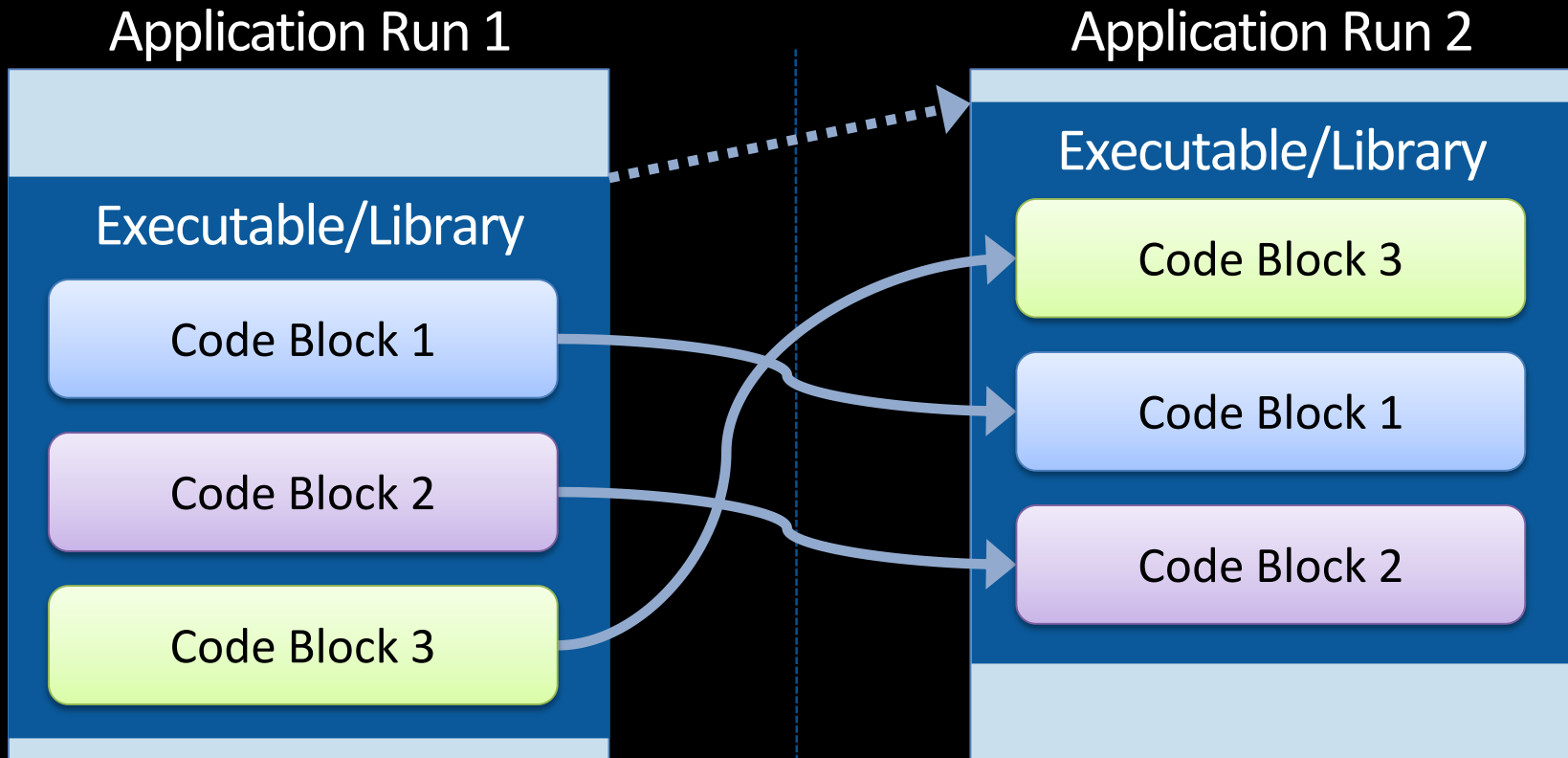


Basics of Memory Randomization

- ASLR randomizes the base address of code/data segments



Fine-Grained ASLR



- ♦ **ORP** [Pappas et al., IEEE S&P 2012]: Instruction reordering/substitution within a BBL
- ♦ **ILR** [Hiser et al., IEEE S&P 2012]: Randomizing each instruction's location
- ♦ **STIR** [Wartell et al., ACM CCS 2012] & **XIFER** [Davi et al., AsiaCCS 2013]: Permutation of BBLs

Does Fine-Grained ASLR Provide a Viable Defense in the Long Run?



Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained
Address Space Layout Randomization

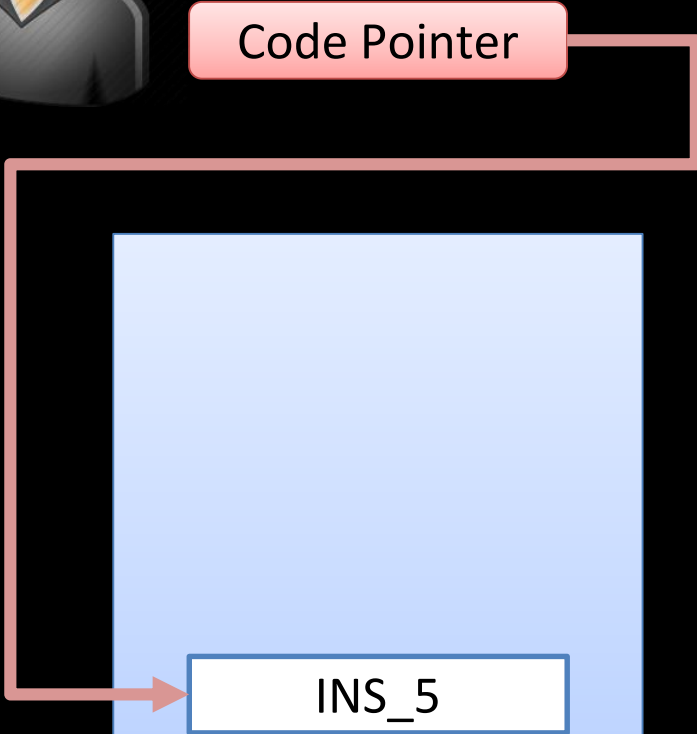
IEEE Security and Privacy Best Student Paper 2013

Kevin Z. Snow (UNC Chapel Hill), Lucas Davi, Alexandra
Dmitrienko, Christopher Liebchen, Fabian Monrose (UNC
Chapel Hill), Ahmad-Reza Sadeghi

High-Level Idea



Code Pointer



INS_5

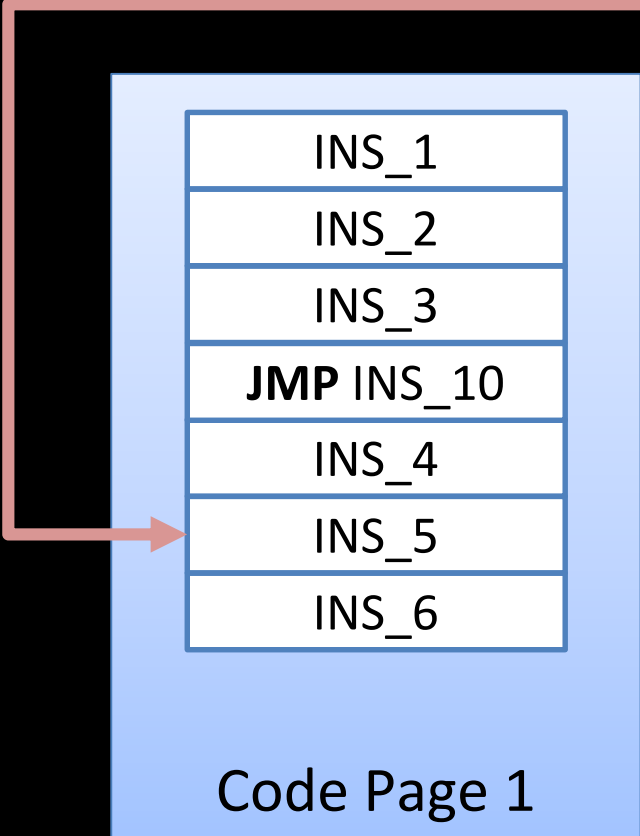
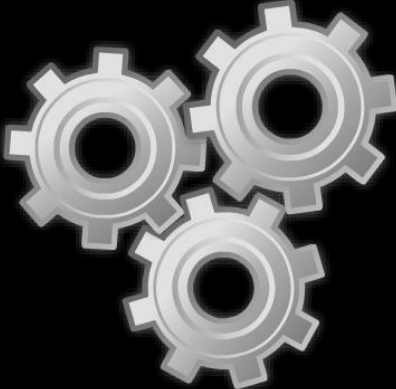
Code Page 1

High-Level Idea

Scripting Engine



Code Pointer

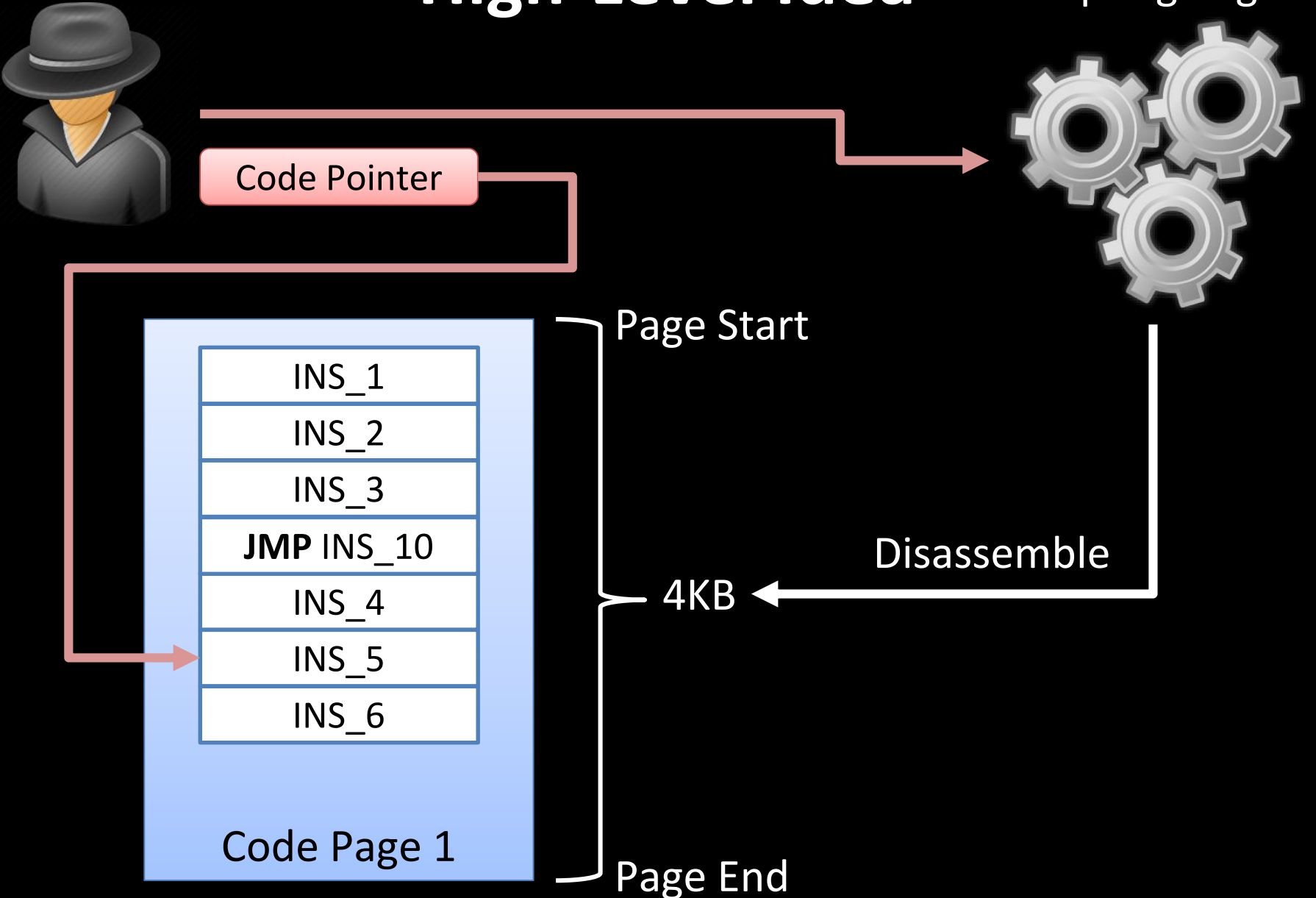


Page Start

4KB

Page End

Disassemble

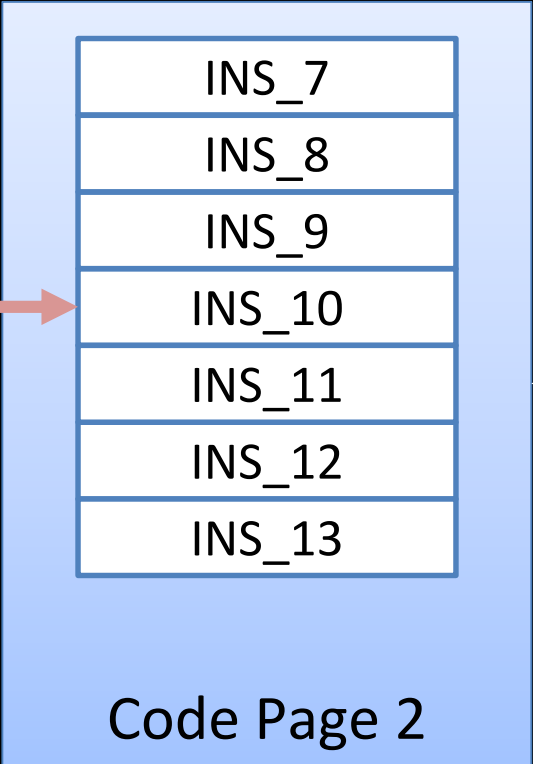
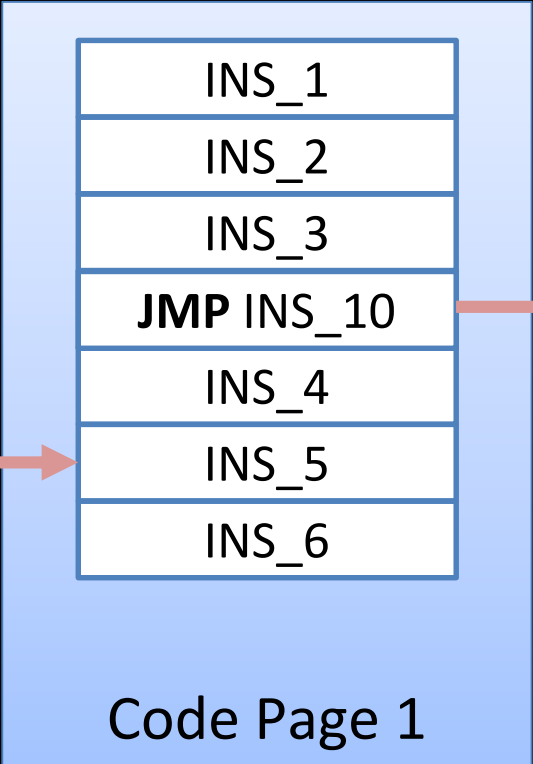
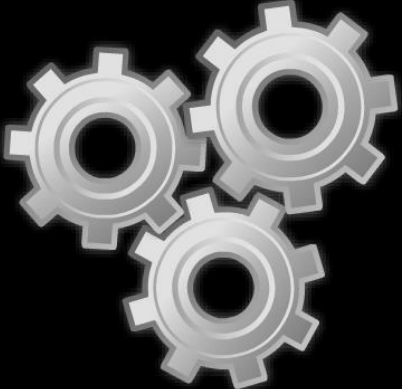


High-Level Idea

Scripting Engine



Code Pointer



Code Randomization: Lessons Learned

1. Memory disclosure attacks are far more damaging than previously believed

→ A single address-instruction mapping leads to many leaks of code pages

2. Fine-grained ASLR can be bypassed with JIT-ROP

→ Enforce execute-only memory

Software-based [Backes et al., CCS 2014]

Hardware-based: Readactor(++) [with Crane et al., IEEE S&P 2015 & CCS 2015]

→ Combine code- and execution randomization

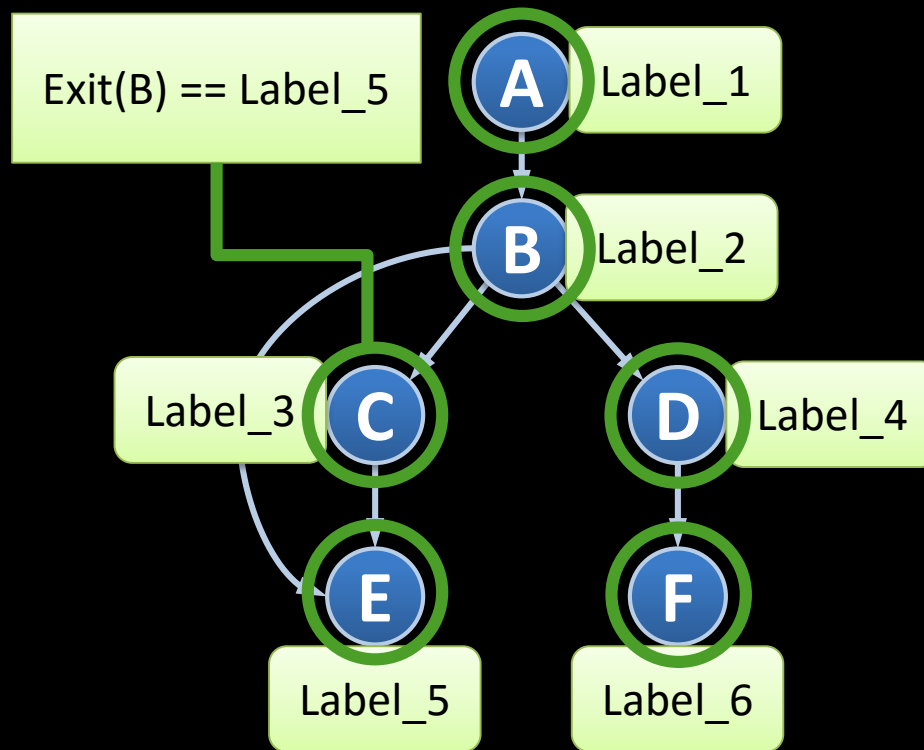
Isomeron [with Liebchen et al., NDSS 2015]

→ Mitigating memory disclosure

Control-Flow Integrity (CFI)

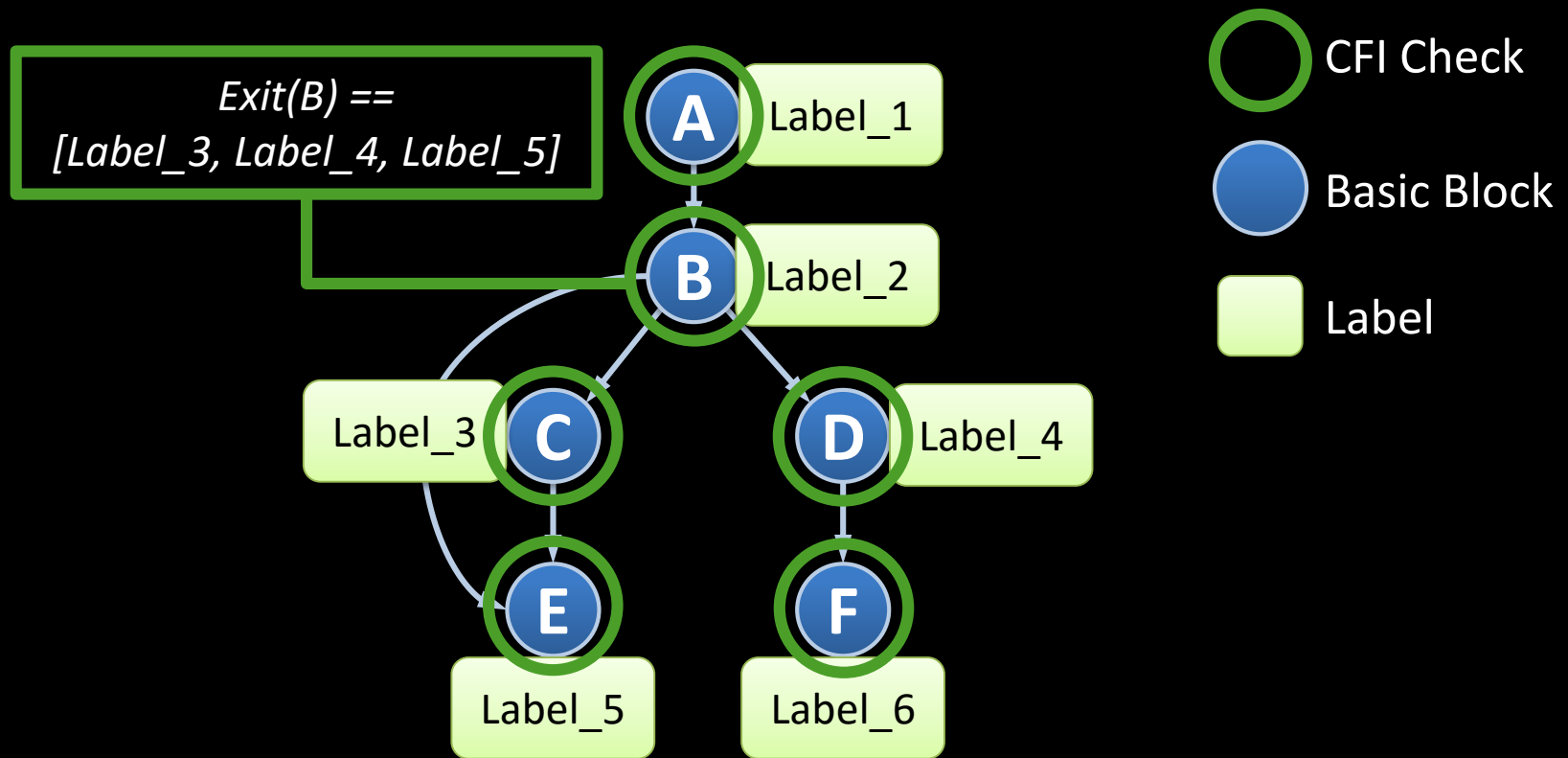
[Abadi et al., CCS 2005 & TISSEC 2009]

A general defense against code-reuse attacks



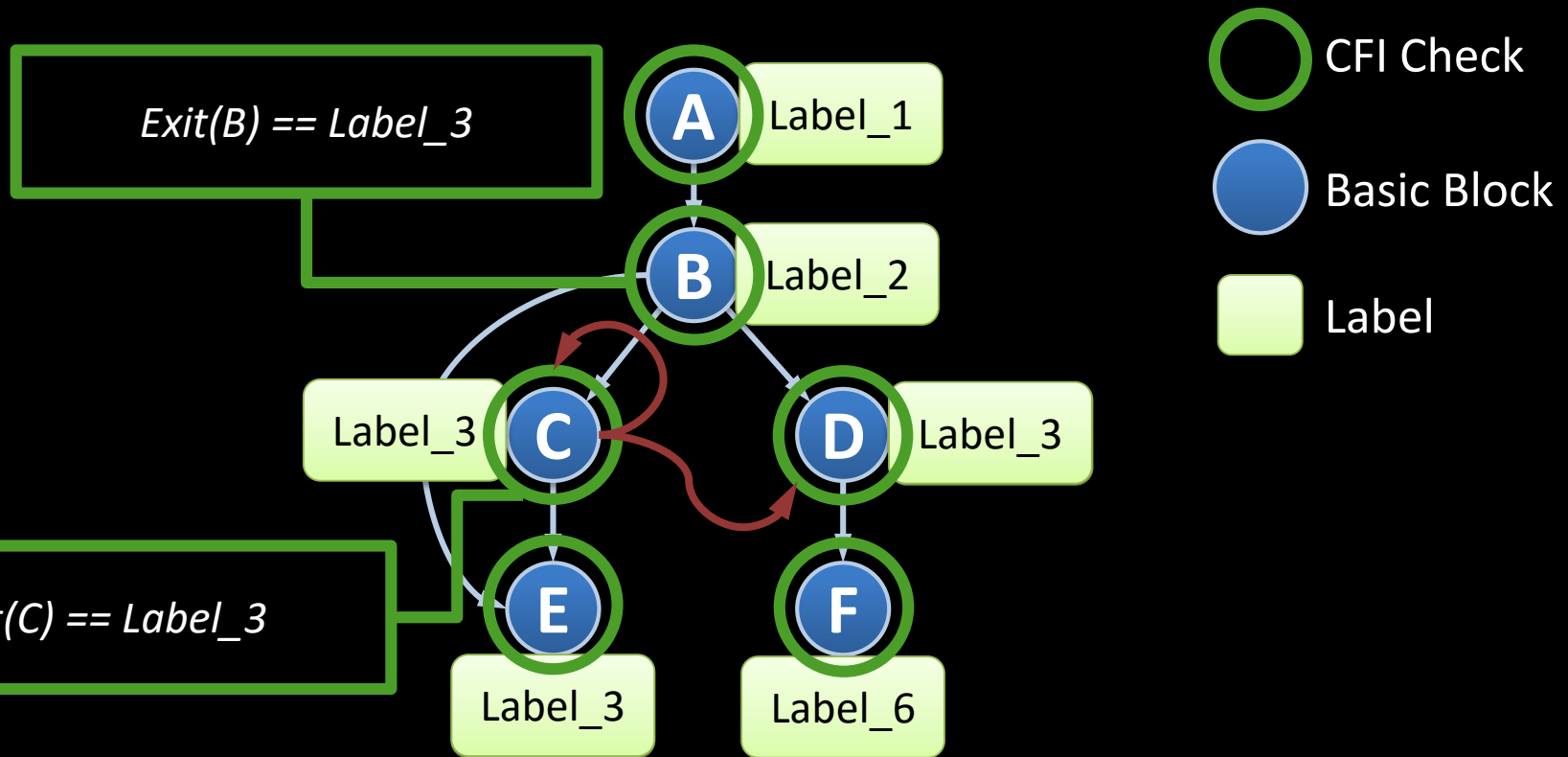
Label Granularity: Trade-Offs (1/2)

- Many CFI checks are required if unique labels are assigned per node



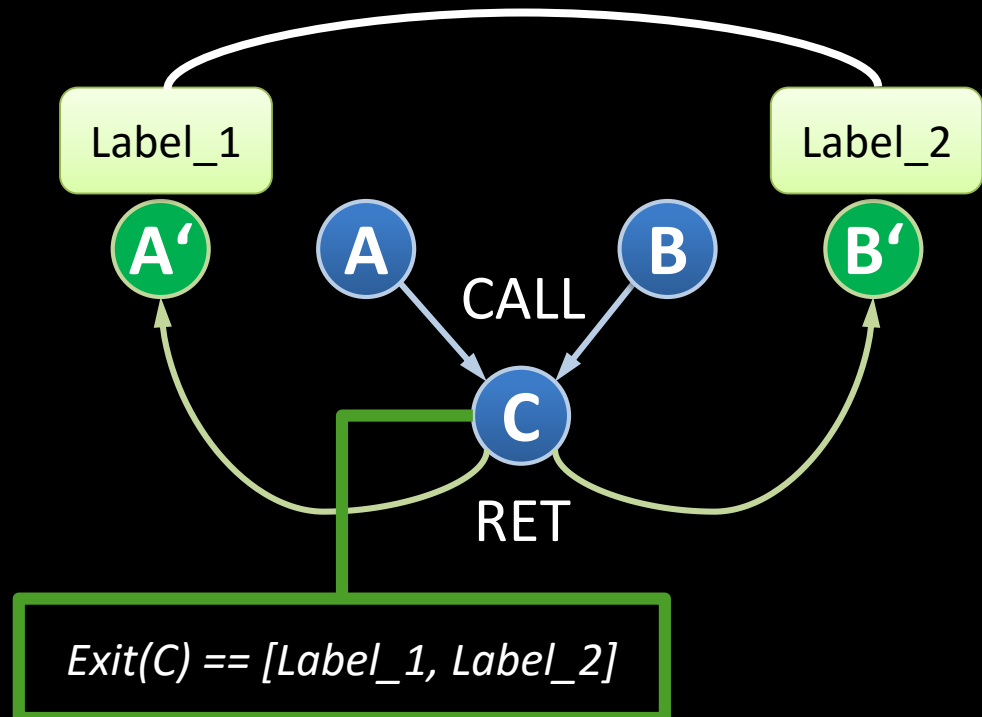
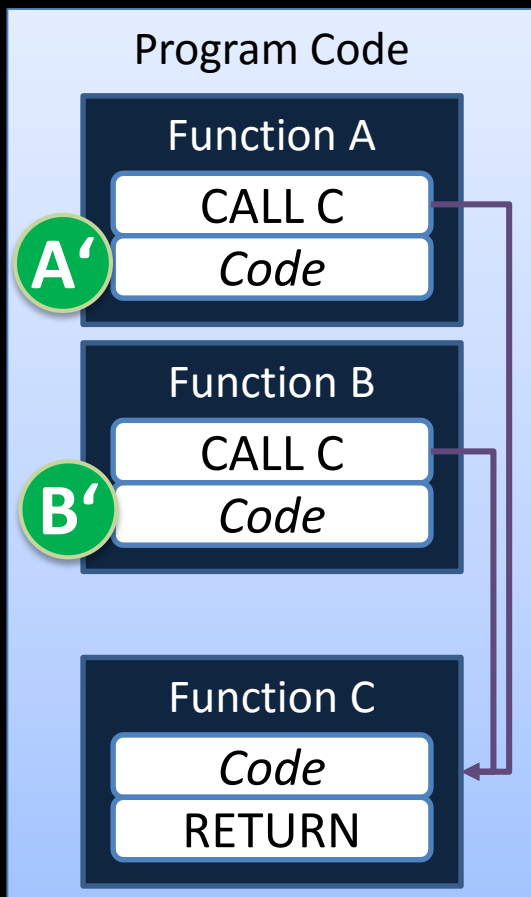
Label Granularity: Trade-Offs (2/2)

- ♦ Optimization step: Merge labels to allow single CFI check
- ♦ However, this allows for unintended control-flow paths



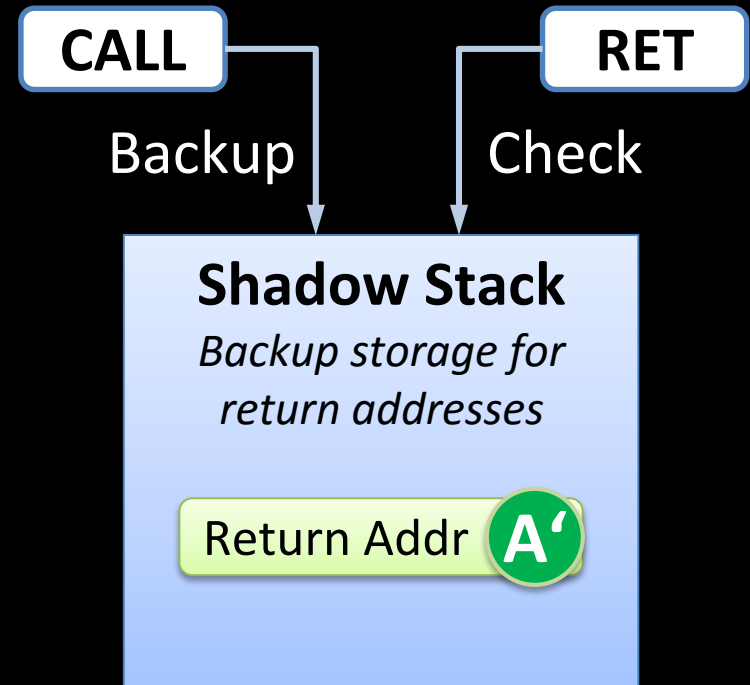
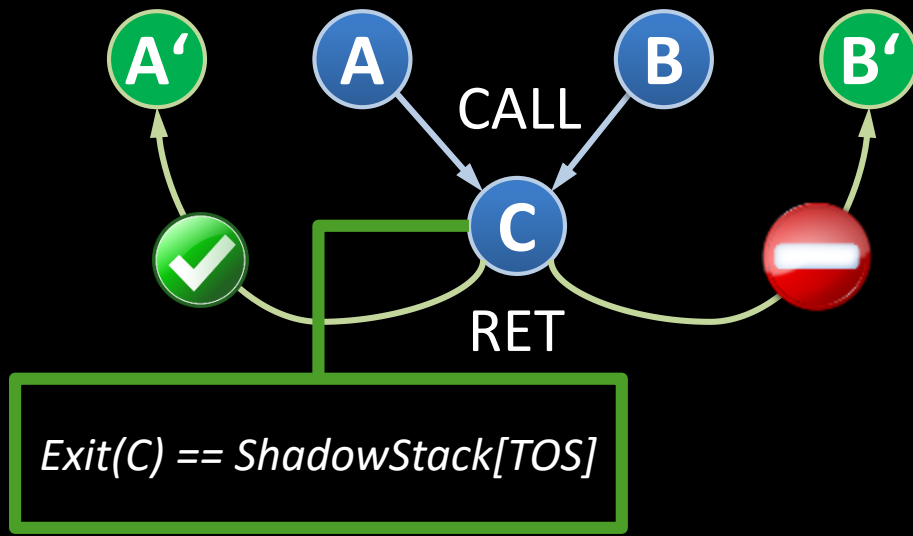
Label Problem for Returns

- **Static CFI label checking** leads to coarse-grained protection for returns



Shadow Stack / Return Address Stack

- ◆ **Shadow stack** allows for fine-grained return address protection but incurs higher overhead



CFI: Benefits and Limitations



Fine-grained protection

Blackbox Vulcan
(unpublished)



Require side info
(debug symbols,
compiler support)



Performance overhead



Hot Research Topic: “Practical” (coarse-grained) Control Flow Integrity (CFI)

Recently, many solutions proposed

CCFIR
[IEEE S&P'13]



kBouncer
[USENIX Sec'13]

ROPecker
[NDSS'14]



ROPGuard
[Microsoft EMET]

CFI for COTS
Binaries
[USENIX Sec'13]



EMET

<http://technet.microsoft.com/en-us/security/jj653751>

Open Question:

Practical and secure mitigation of code reuse attacks

Turing-completeness of return-oriented programming

Negative Result:

All current (published)
coarse-grained CFI solutions can be
bypassed

Big Picture

Systematic Security
Analysis of Coarse-
Grained CFI

Gadget
Analysis

Exploit
Development

CFI Policies

Frequency of CFI Checks

Deriving a CFI policy that
combines all schemes

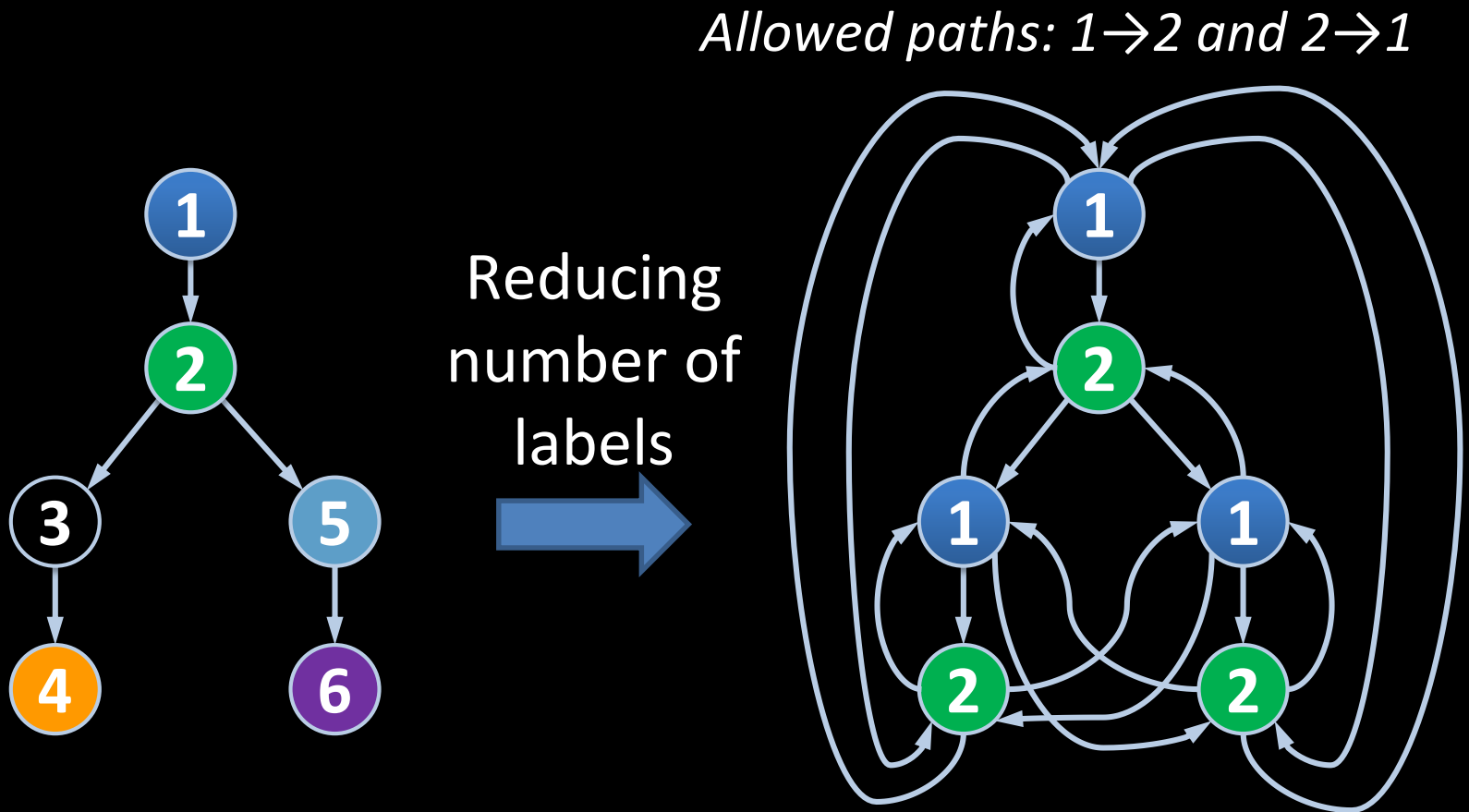
Turing-complete
gadget set

Gadgets to
bypass heuristics



1. Systematic Security Analysis of Coarse-Grained CFI

Coarse-grained CFI leads to CFG imprecision



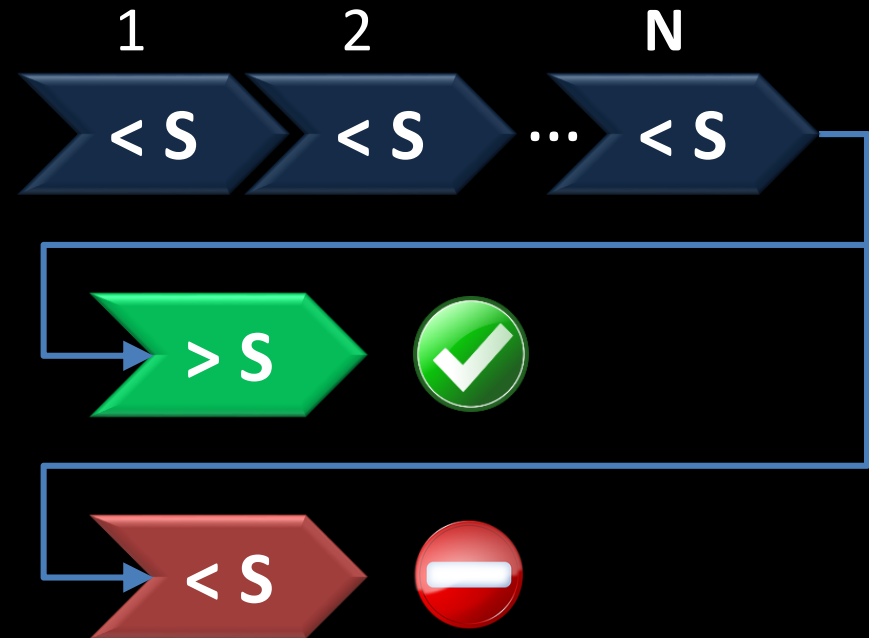
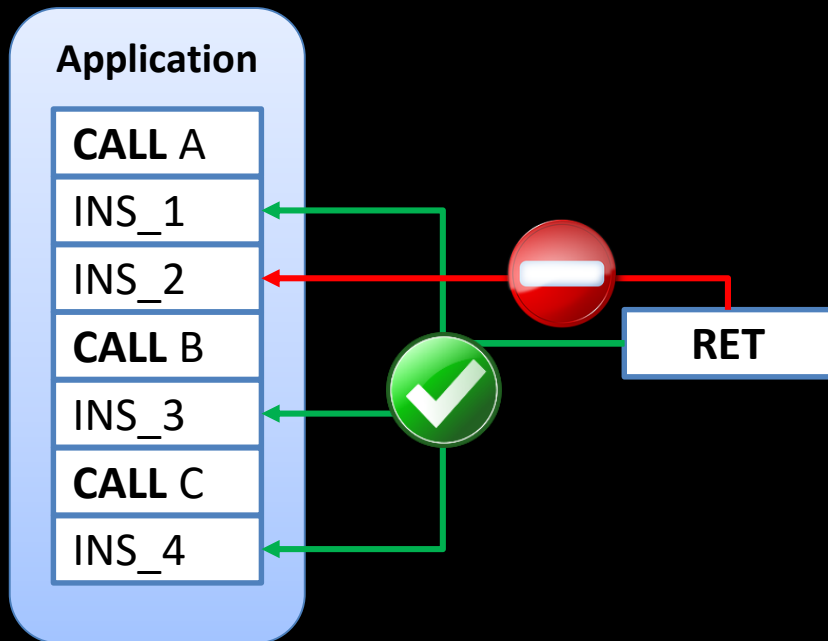
Main Coarse-Grained CFI Policies

- ◆ **CFI Policy 1: Call-Preceded Sequences**

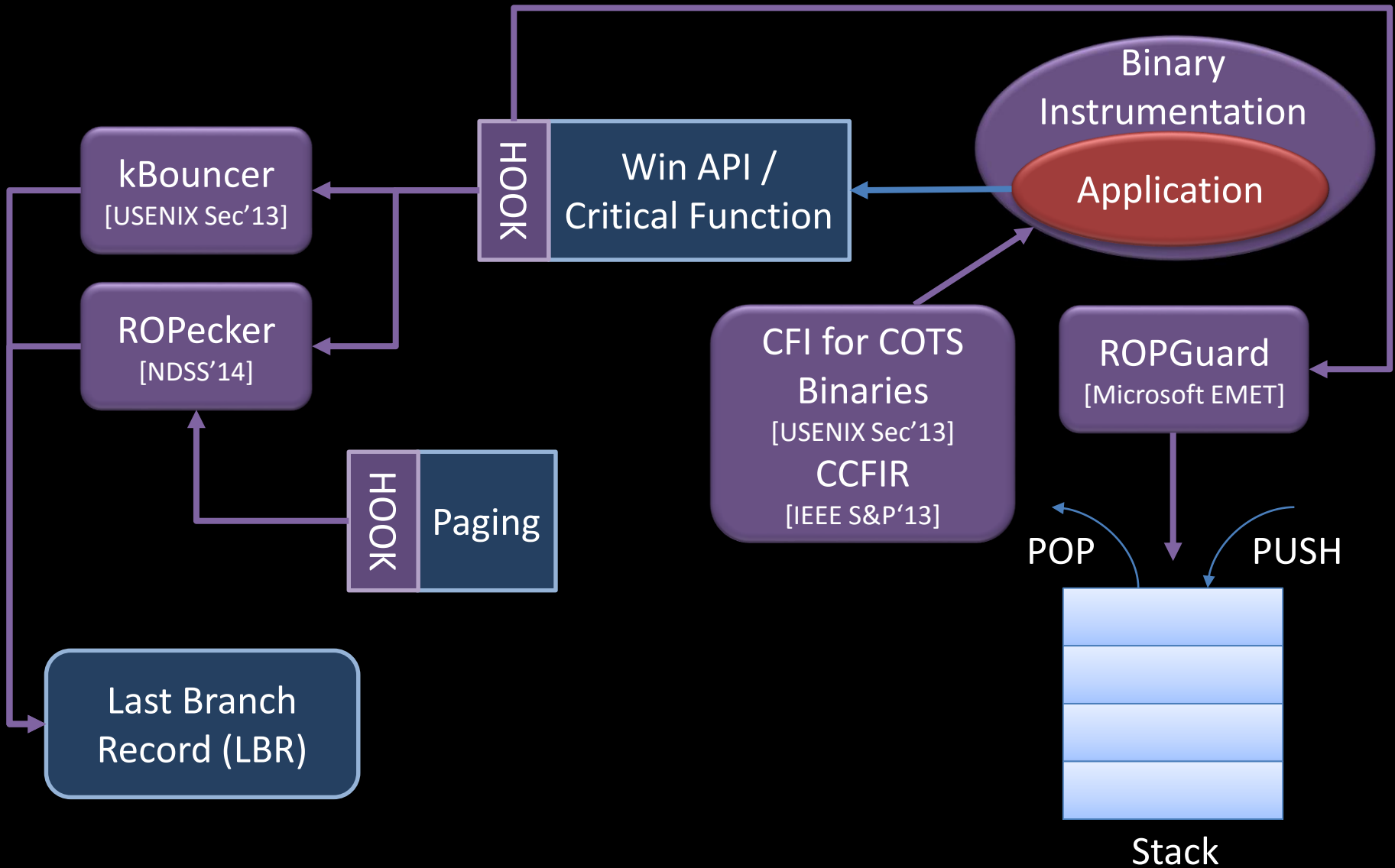
- ◆ Returns need to target a call-preceded instruction
- ◆ **No shadow stack required**

- ◆ **CFI Policy 2: Behavioral-Based Heuristics**

Threshold Setting
kBouncer: (N=8; S≤20)
ROPecker: (N=11; S≤6)



Coarse-Grained CFI Proposals



Deriving a Combined CFI Policy

CFI Policy	kBouncer [USENIX Sec. 2013]	ROPecker [NDSS 2014]	ROPGuard [Microsoft EMET]	CFI for COTS Binaries [USENIX Sec. 2013]	<i>Combined CFI Policy</i>
CFI Policy 1 <i>Call-Preceded Sequences</i>					
CFI Policy 2 <i>Behavioral-Based Heuristics</i>					
Time of CFI Check	WinAPI	2 Page Sliding Window/ Critical Functions	WinAPI/ Critical Functions	Indirect Branch	Any Time



No Restriction

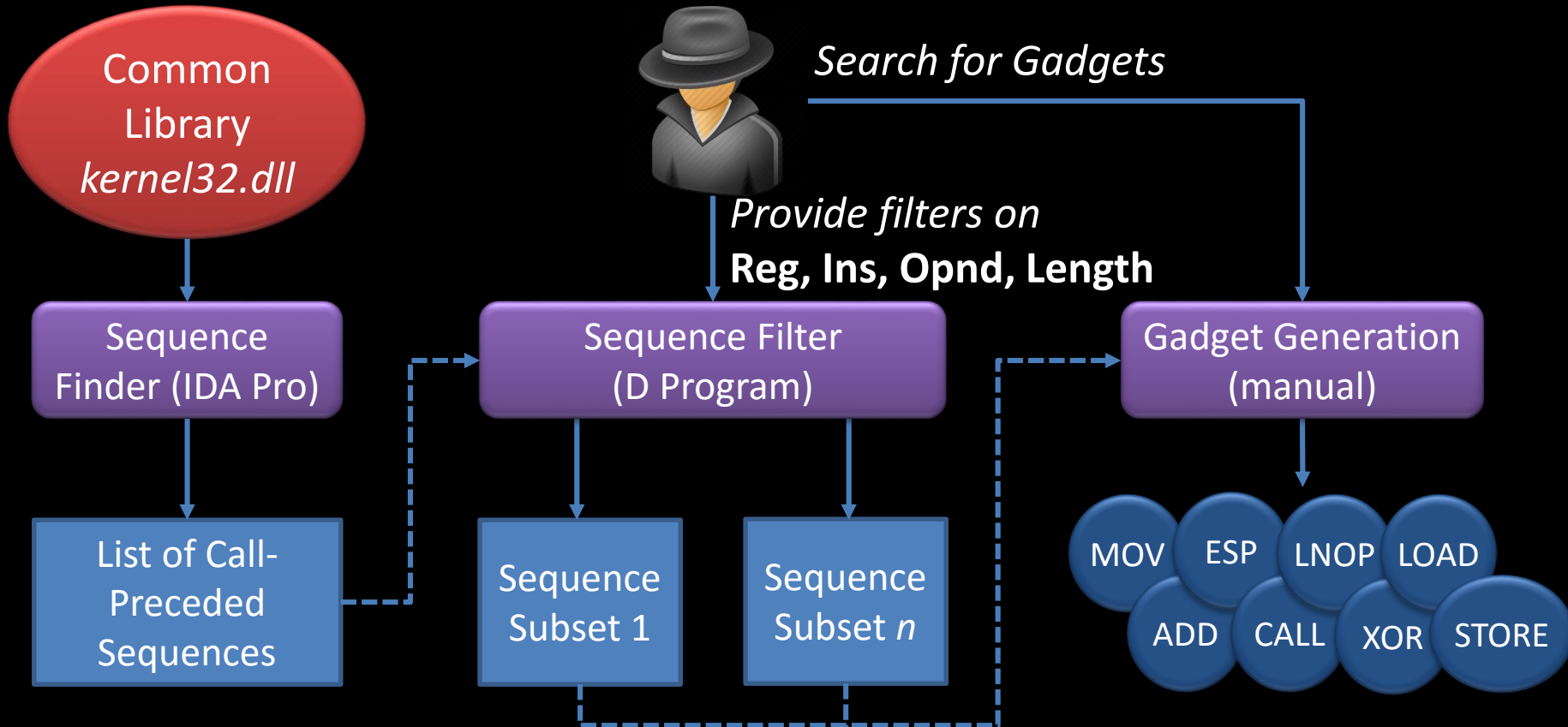


CFI Policy

Here only the core policies shown. However, we consider all other deployed policies in our analysis.

2. Gadget Analysis

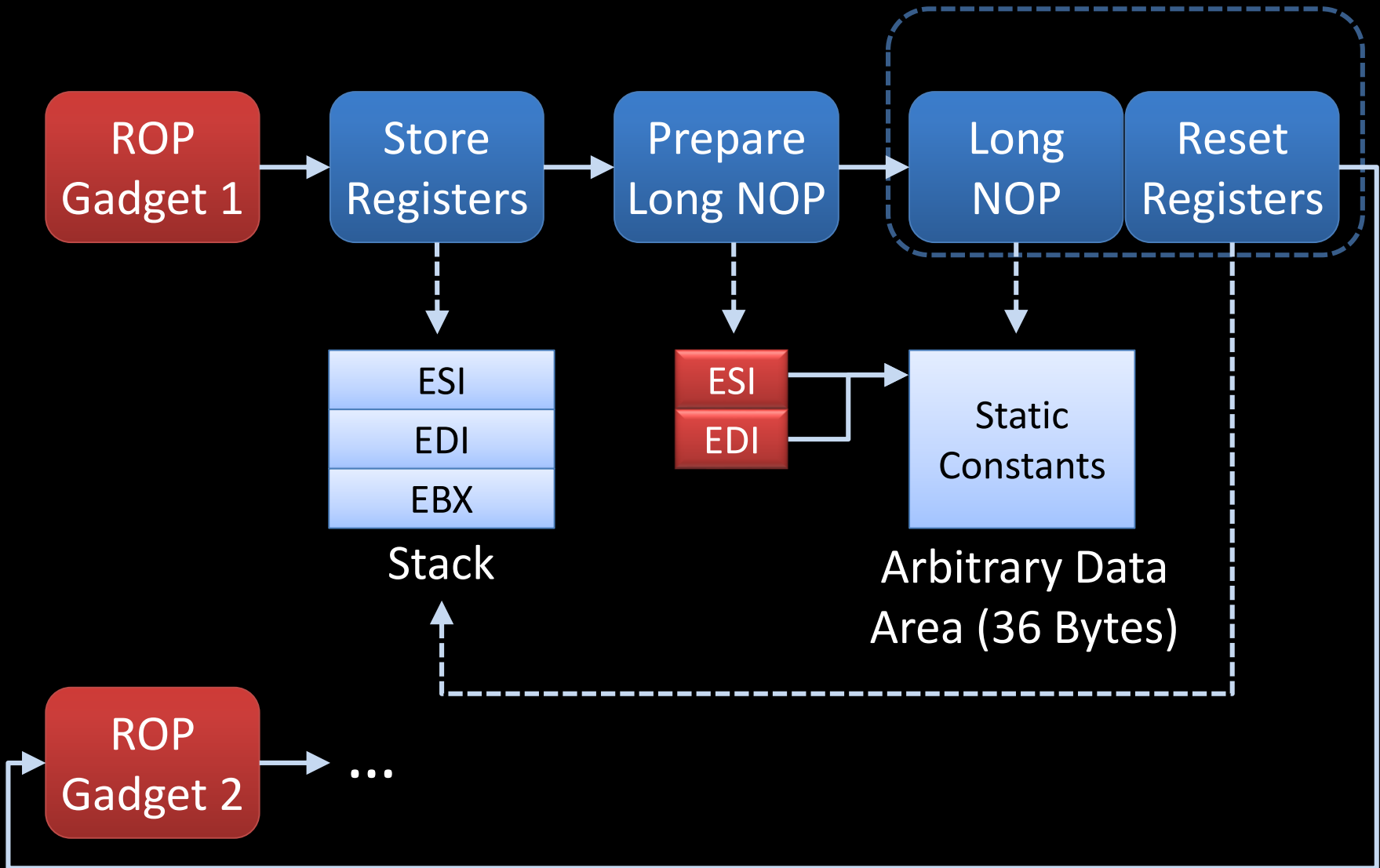
Methodology



(Excerpt of) Turing-Complete Gadget Set in CFI-Protected *kernel32.dll*

Gadget Type	CALL-Preceded Sequence <i>ending in a RET instruction</i>
LOAD Register	<pre> EBP := pop ebp ESI := pop esi; pop ebp EDI := pop edi; leave ECX := pop ecx; leave EBX := pop edi; pop esi; pop ebx; pop ebp EAX := mov eax,edi; pop edi; leave EDX := mov eax,[ebp-8]; mov edx,[ebp-4]; pop edi; leave </pre>
LOAD/STORE Memory	<pre> LD(EAX) := mov eax,[ebp+8]; pop ebp ST(EAX) := mov [esi],eax; xor eax,eax; pop esi; pop ebp ST(ESI) := mov [ebp-20h],esi ST(EDI) := mov [ebp-20h],edi </pre>
Arithmetic/ Logical	<pre> ADD/SUB := sub eax,esi; pop esi; pop ebp XOR := xor eax,edi; pop edi; pop esi; pop ebp </pre>
Branches	<pre> unconditional branch 1 := leave unconditional branch 2 := add esp,0Ch; pop ebp conditional LD(EAX) := neg eax; sbb eax,eax; and eax,[ebp-4]; leave </pre>

Long-NOP Gadget



3. Exploit Development

Adobe Reader 9.1
CVE-2010-0188



MPlayer Lite r33064 m3u
Buffer Overflow Exploit



Original exploits
detected by coarse-
grained CFI



Our instrumented
exploits bypass coarse-
grained CFI



Coarse-Grained CFI: Lessons Learned

1. Too many call sites available

→ Restrict returns to their actual caller (shadow stack)

2. Heuristics are ad-hoc and ineffective

→ Adjusted sequence length leads to high false positive

3. Too many indirect jump and call targets

♦ Resolving indirect jumps and calls is non-trivial

→ Compromise: Compiler support

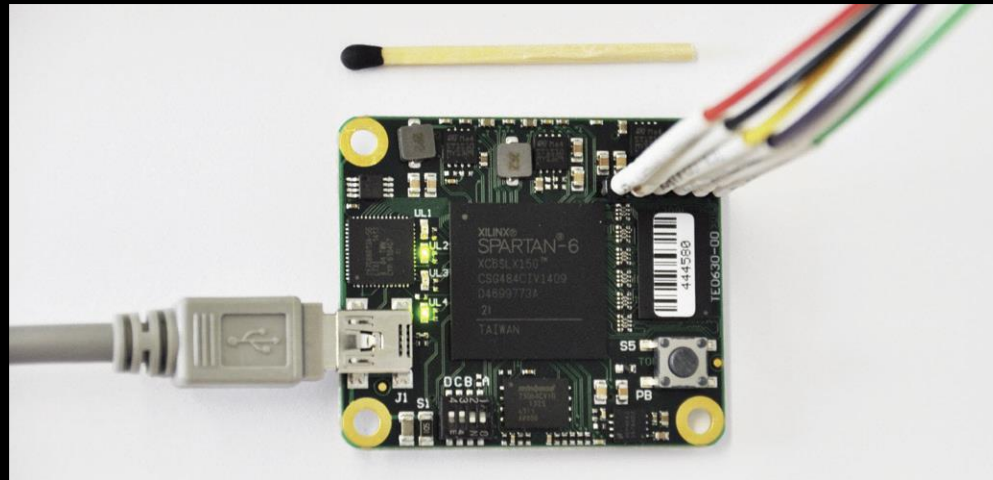
CURRENT RESEARCH

What's next?

Hardware-Assisted CFI

HAFIX: Hardware Flow Integrity Extensions

[O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul,
A.-R. Sadeghi, D. Sullivan, DAC 2015, Best Paper]



Design Decisions: Why CFI Processor Support?

CFI Processor Support based on Instruction set architecture (ISA) extensions

Dedicated CFI instructions

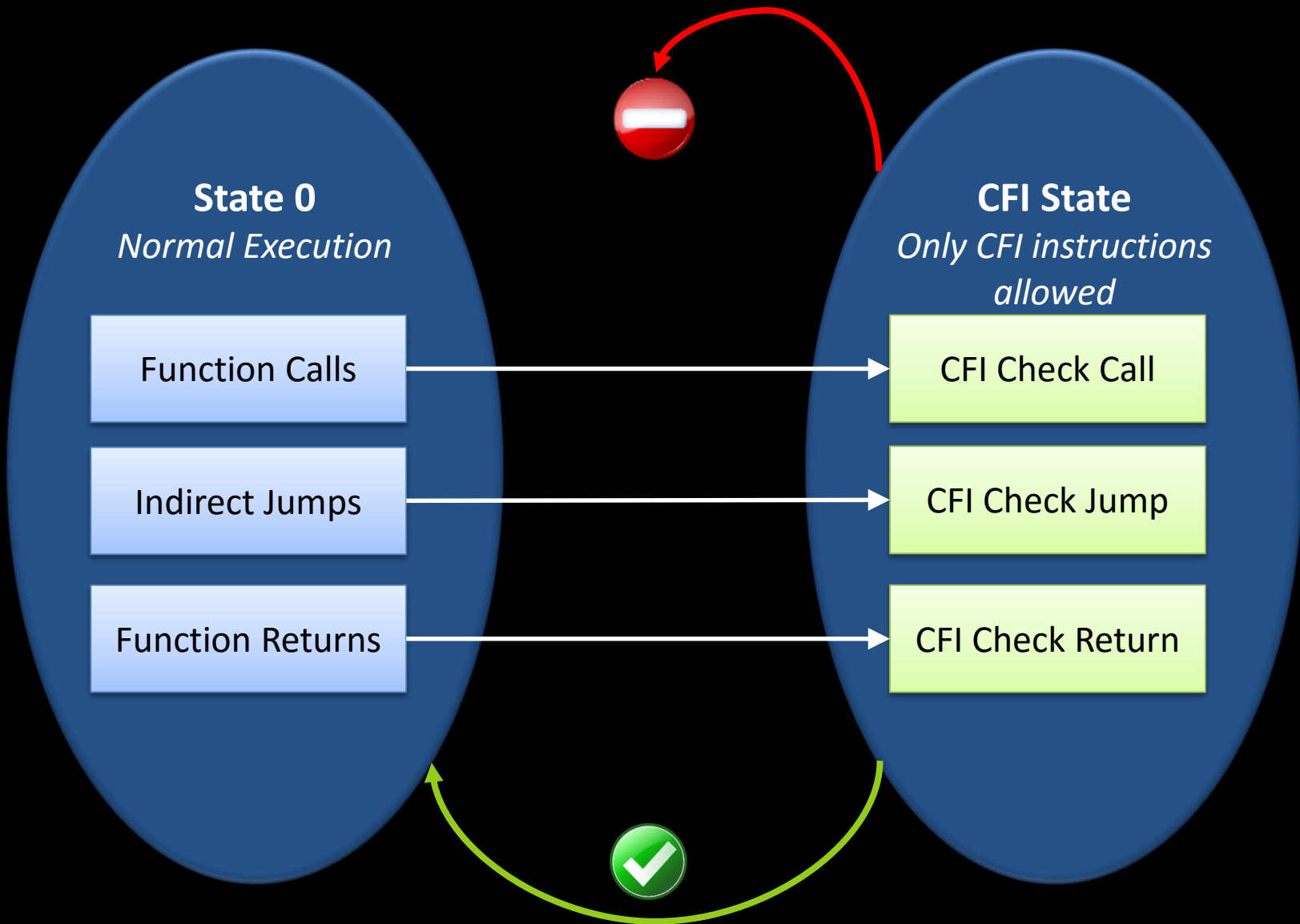
No offline training phase

Instant attack detection

CFI control state

Binding of CFI data to CFI state and instructions

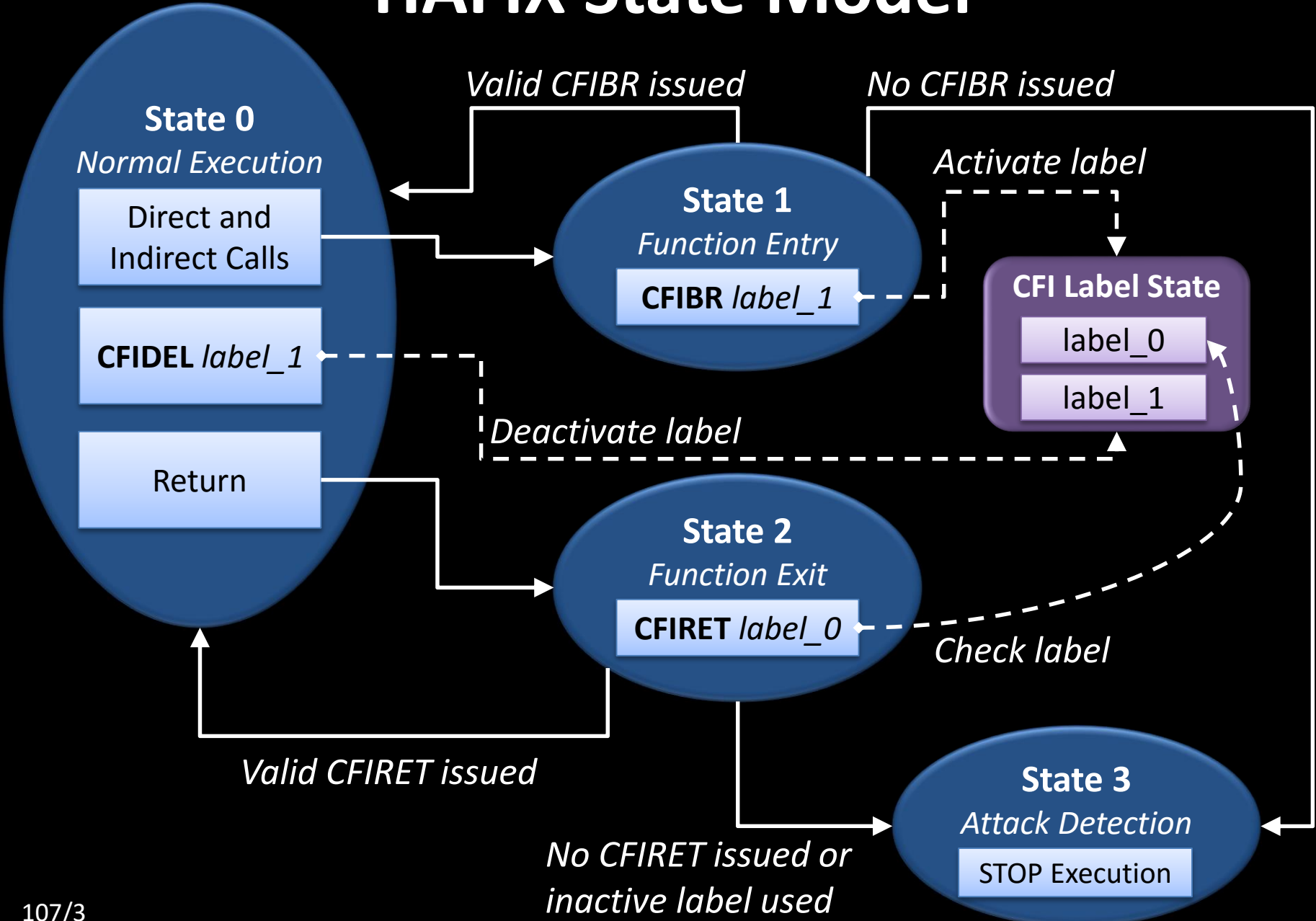
Big Picture



Example Policy

Returns can only target call sites of functions that are currently executing

HAFIX State Model



Remarks

- ◆ Implementation on Intel Siskiyou Peak and SPARC-LEON3
- ◆ High efficiency 1-2%
- ◆ Current prototype supports different levels of CFG precision [visit our DAC'16 talk on Thursday, June 09, 3:30pm - 5:30pm | 19AB]

Conclusion

- ◆ Code-reuse attacks are prevalent
 - ◆ Google and Microsoft take these attacks seriously
 - ◆ Many real-world exploits
 - ◆ Existing solutions can be bypassed
- ◆ Good News
 - ◆ Many innovative defense techniques have been proposed
- ◆ Promising new directions
 - ◆ Memory safety based on code-pointer integrity [Kuznetsov et al., OSDI 2014]

References

References (1/5)

- ♦ [Abadi et al., ACM CCS 2005 & ACM TISSEC 2009]
M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti.
Control-flow integrity: Principles, implementations, and applications.
- ♦ [Buchanan et al., ACM CCS 2008]
E. Buchanan, R. Roemer, H. Shacham, and S. Savage.
When good instructions go bad: Generalizing return-oriented programming to RISC.
- ♦ [Checkoway et al., EVT/WOTE 2009]
S. Checkoway, A.J. Feldman, B. Kantor, J.A. Halderman, E.W. Felten, and H. Shacham.
Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage.
- ♦ [Checkoway et al., ACM CCS 2010]
S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy.
Return-oriented programming without returns.
- ♦ [Cheng et al., NDSS 2014]
Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng.
ROPecker: A generic and practical approach for defending against ROP attacks.
- ♦ [Cohen, Computer & Security 1993]
F. B. Cohen.
Operating system protection through program evolution.

References (2/5)

- ♦ [Cowan et al., USENIX Security 1998]
C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang.
StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks.
- ♦ [Davi et al., ASIACCS 2013]
L. Davi, A. Dmitrienko, S. Nürnberger, A.-R. Sadeghi.
Gadge me if you can - Secure and efficient ad-hoc instruction-level randomization for x86 and ARM.
- ♦ [Davi et al., ASIACCS 2011]
L. Davi, A.-R. Sadeghi, and M. Winandy.
ROPdefender: A detection tool to defend against return-oriented programming attacks.
- ♦ [Davi et al., USENIX Security 2014]
L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose.
Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection.
- ♦ [Davi et al., DAC 2014]
L. Davi, P. Koeberl, and A.-R. Sadeghi.
Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation.

References (3/5)

- ♦ [Erlingsson, Technical Report 2007]
Ú. Erlingsson.
Low-level software security: Attacks and defenses.
- ♦ [Forrest et al., Hot Topics in Operating Systems 1997]
S. Forrest, A. Somayaji, and D. Ackley.
Building diverse computer systems.
- ♦ [Fratric, Technical Report 2012]
I. Fratric.
ROPGuard: Runtime prevention of return-oriented programming attacks.
- ♦ [Francillon et al., ACM CCS 2008]
A. Francillon and C. Castelluccia.
Code injection attacks on Harvard-architecture devices.
- ♦ [Hiser et al., IEEE Security & Privacy 2012]
J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson.
ILR: Where'd my gadgets go?.
- ♦ [Iozzo et al., Pwn2Own 2010]
Ralf-Philipp Weinmann and Vincenzo Iozzo.

References (4/5)

- ♦ [Pappas et al., IEEE Security & Privacy 2012]
V. Pappas, M. Polychronakis, and A. D. Keromytis.
Smashing the gadgets: Hindering return-oriented programming using in-place code randomization.
- ♦ [Pappas et al., USENIX Security 2013]
V. Pappas, M. Polychronakis, and A. D. Keromytis.
Transparent ROP exploit mitigation using indirect branch tracing.
- ♦ [Shacham, ACM CCS 2004]
H. Shacham.
The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).
- ♦ [Shacham, ACM CCS 2007]
H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh.
On the effectiveness of address-space randomization.
- ♦ [Snow et al., IEEE Security & Privacy 2013]
K. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monroe, A.-R. Sadeghi.
Just-in-time code reuse: On the effectiveness of fine-grained ASLR.

References (5/5)

- ♦ [Sotirov et al., BlackHat USA 2013]
A. Sotirov and M. Dowd.
Bypassing browser memory protections in Windows Vista.
- ♦ [Wartell et al., ACM CCS 2012]
R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin.
Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code.
- ♦ [Zhang et al., USENIX Security 2013]
M. Zhang and R. Sekar.
Control flow integrity for COTS binaries.
- ♦ [Zhang et al., IEEE Security & Privacy 2013]
C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou.
Practical control flow integrity & randomization for binary executables.
- ♦ [Zovi, RSA Conference 2010]
D. D. Zovi.
Practical return-oriented programming.