

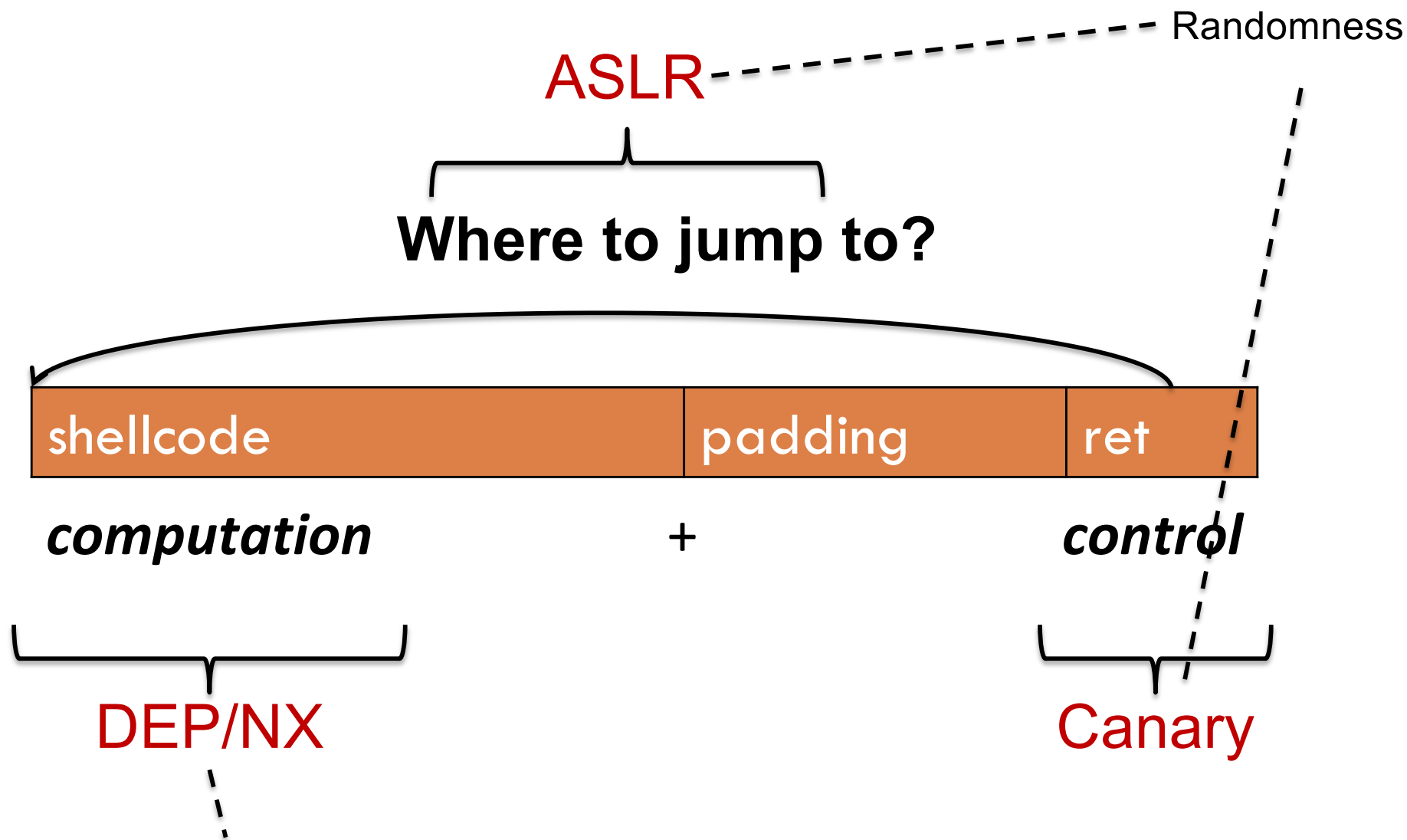
CS260 – Advanced Systems Security

Hardening

April 21, 2025

Current Memory Defenses

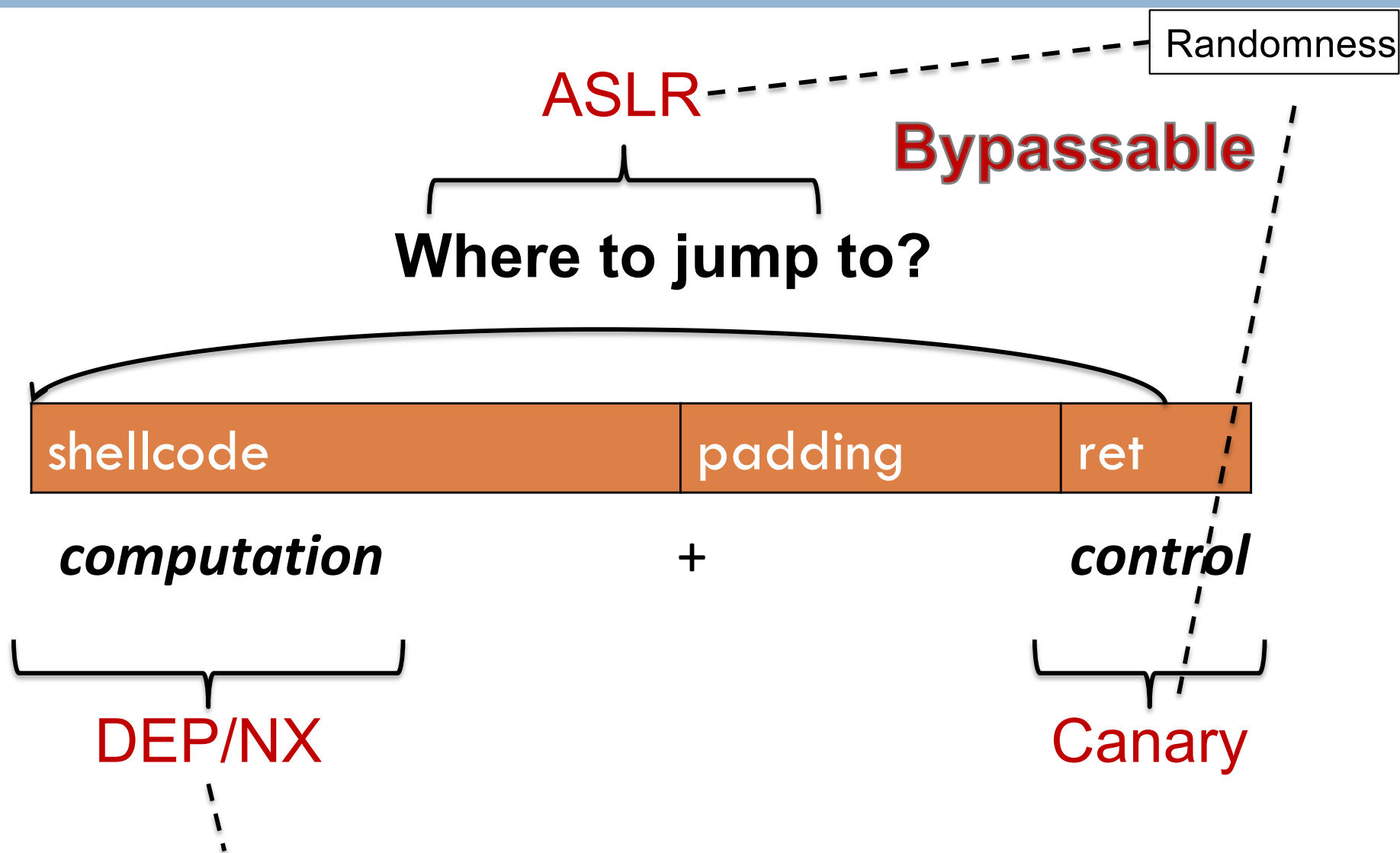
20



Eliminating "Mixing of data and code"

Current Memory Defenses

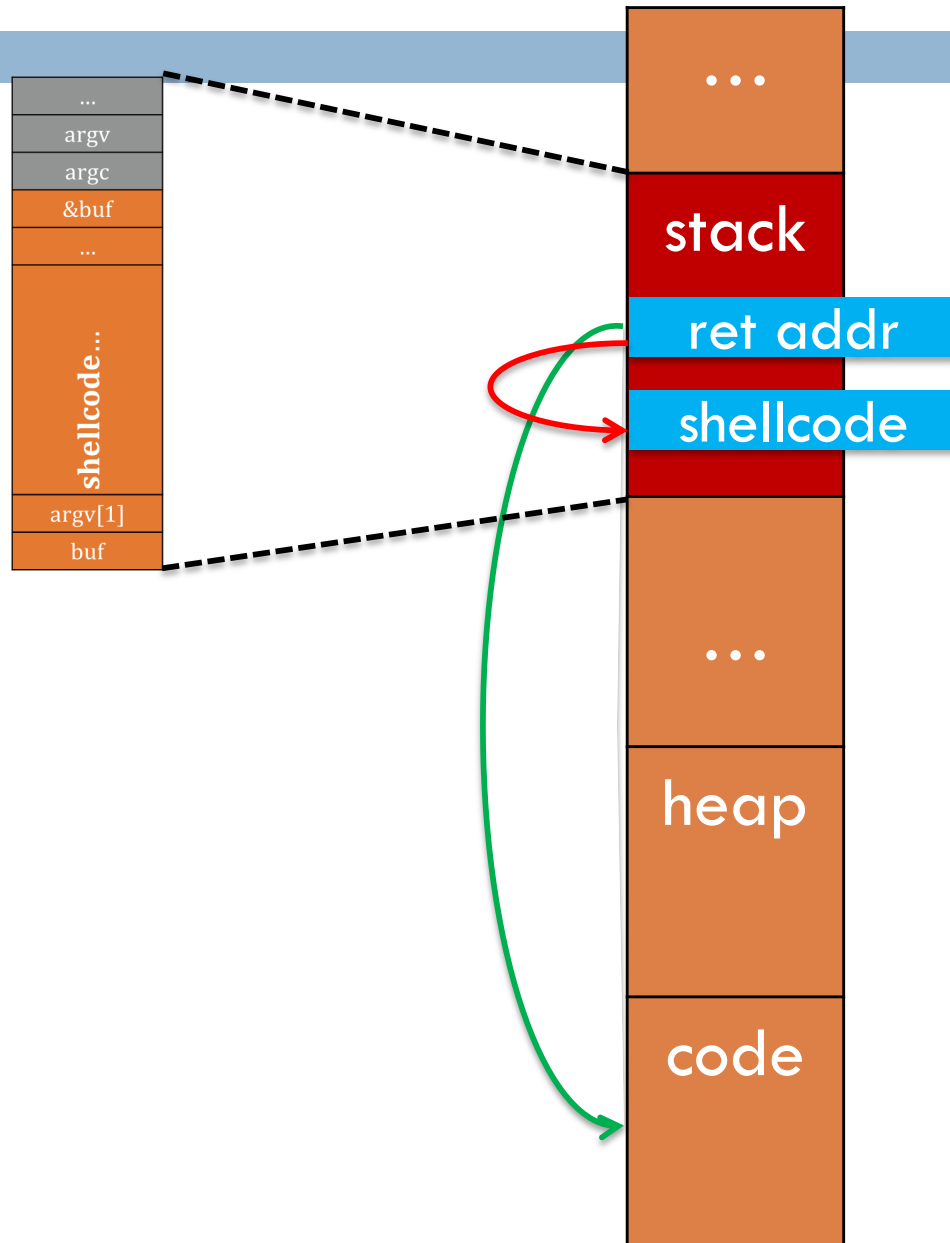
20



Eliminating "Mixing of data and code"

Thwarts Finding Shellcode

5



Motivation: Return-to-libc Attack

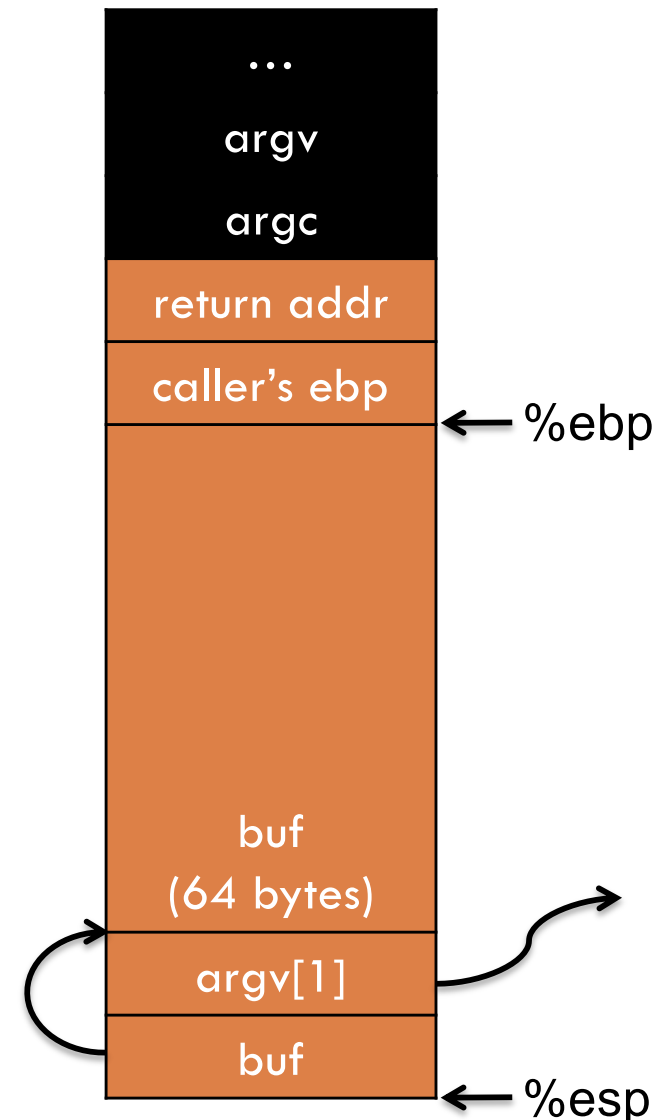
4

Bypassing DEP!

Overwrite return address with address of **libc** function

- setup fake return address and argument(s)
- ret will “call” libc function

No injected code!



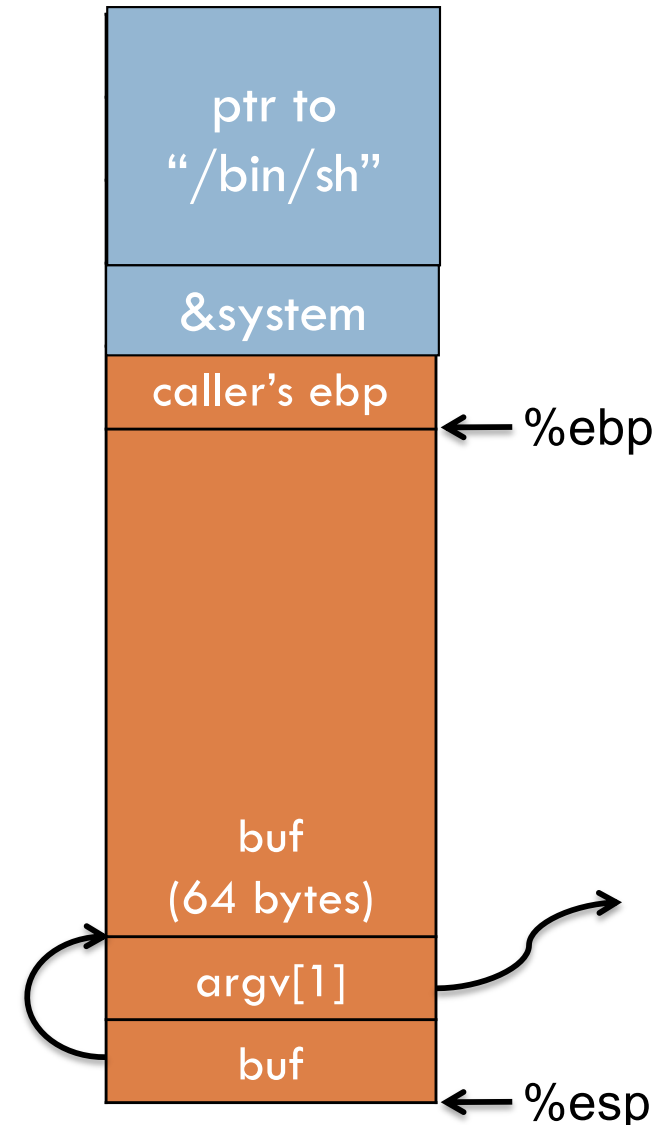
Motivation: Return-to-libc Attack

4

Overwrite return address with address of libc function

- setup fake return address and argument(s)
- ret will “call” libc function

No injected code!



Motivation: Return-to-libc Attack

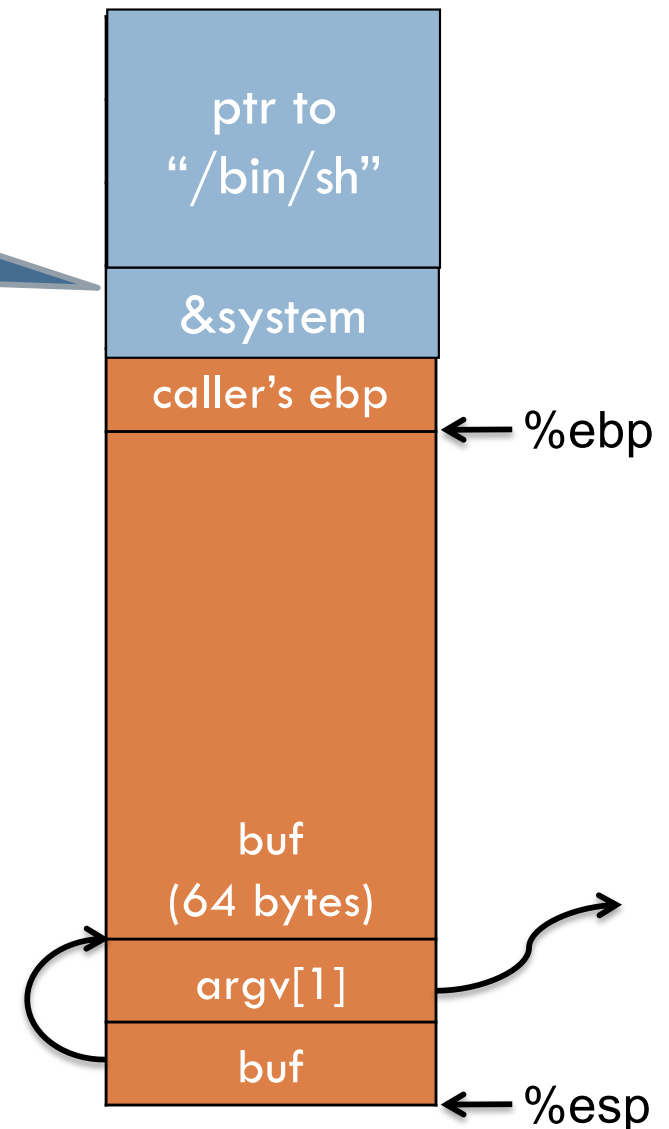
4

ret transfers control to system,
which finds arguments on stack

Overwrite return address with
address of libc function

- setup fake return address and
argument(s)
- ret will “call” libc function

No injected code!



The New York Times

Saturday, January 6, 2007

Daily Blog Tips awarded the

Last week Darren Rowse, from the famous Prologger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among

the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

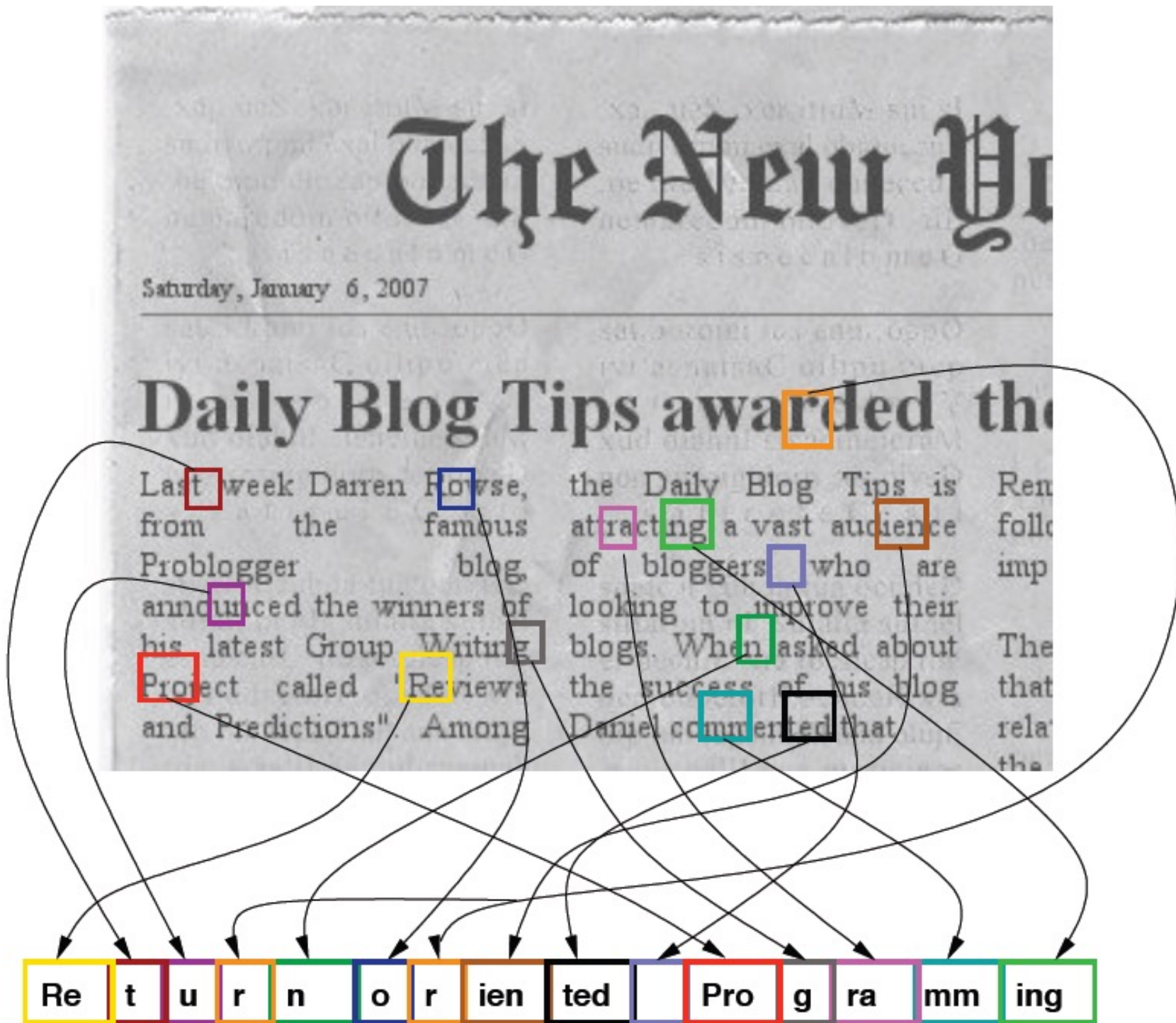
Ren
follo
imp
The
that
rela
the

The New York Times

Saturday, January 6, 2007

Daily Blog Tips awarded the

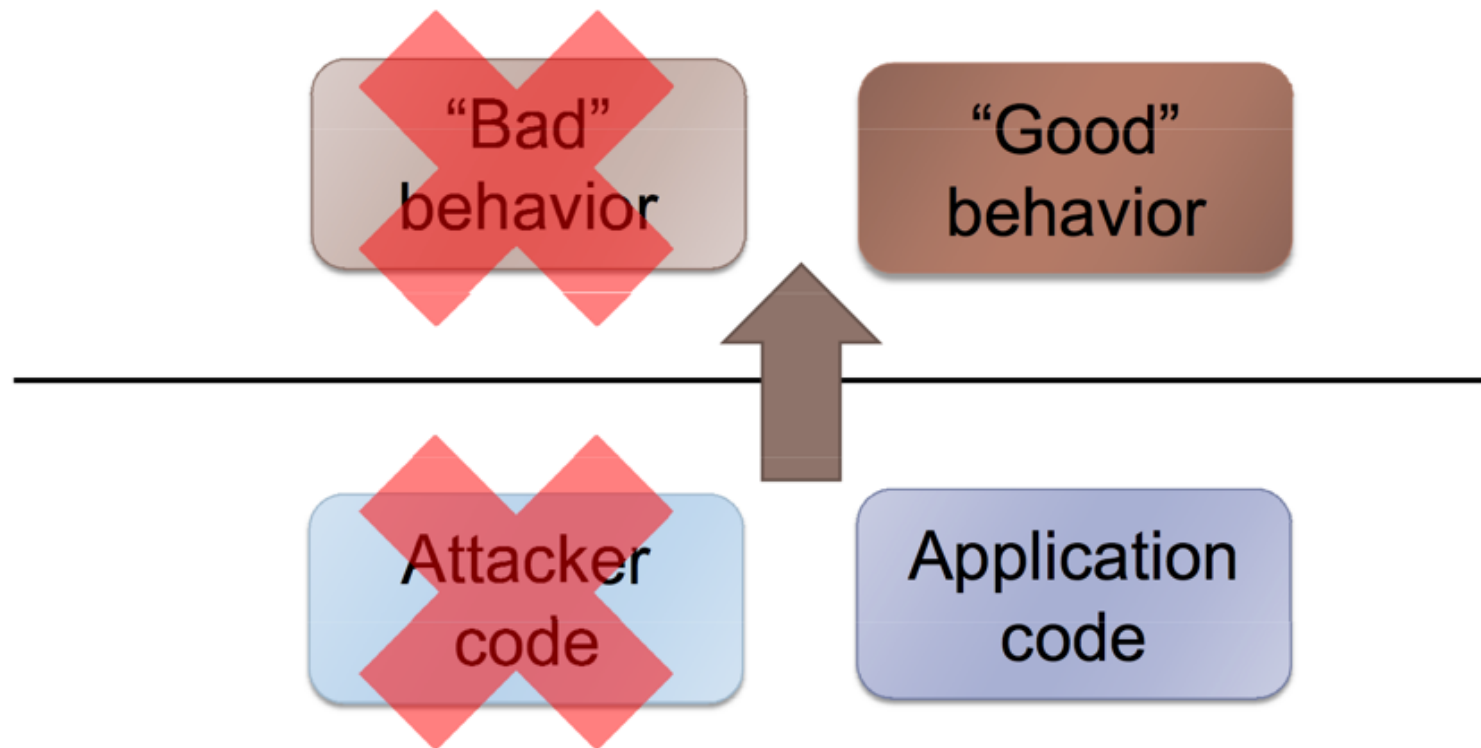
Last week Darren Rowse, from the famous Pro Blogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that



ROP Programming

13

Bad code versus bad behavior



Problem: this implication is false!

ROP Programming

13

attacker control of stack



arbitrary attacker computation and behavior
via return-into-libc techniques

(given any sufficiently large codebase to draw on)

ROP Programming: Key Steps

13

1. Disassemble code
2. Identify useful code sequences as gadgets
3. Assemble gadgets into desired shellcode

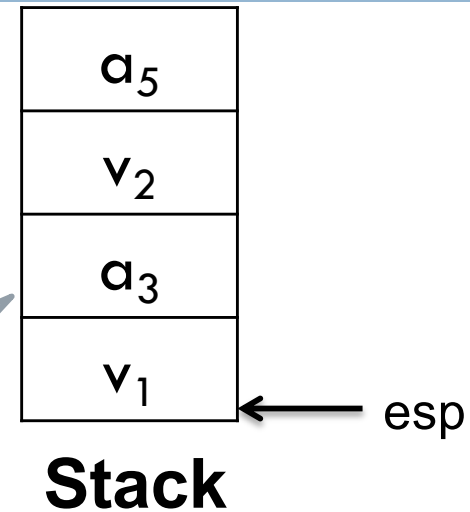
Gadgets

19

Mem[v2] = v1

Desired Logic

Suppose a_3
and a_5 on
stack



eax	
ebx	
eip	a_1

```
 $a_1$ : pop eax;  
 $a_2$ : ret  
 $a_3$ : pop ebx;  
 $a_4$ : ret  
 $a_5$ : mov [ebx], eax
```

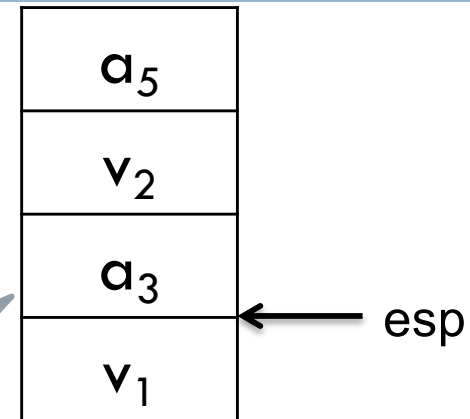
Gadgets

19

Mem[v2] = v1

Desired Logic

Suppose a_3
and a_5 on
stack



Stack

eax	v_1
ebx	
eip	a_1

```
 $a_1$ : pop eax;  
 $a_2$ : ret  
 $a_3$ : pop ebx;  
 $a_4$ : ret  
 $a_5$ : mov [ebx], eax
```

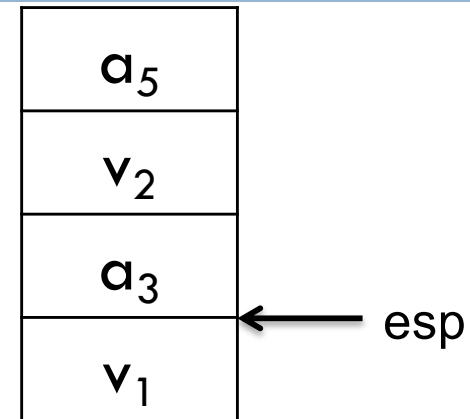
Gadgets

20

Mem[v2] = v1

Desired Logic

eax	v ₁
ebx	
eip	a ₁



Stack

```
a1: pop eax;  
a2: ret  
a3: pop ebx;  
a4: ret  
a5: mov [ebx], eax
```

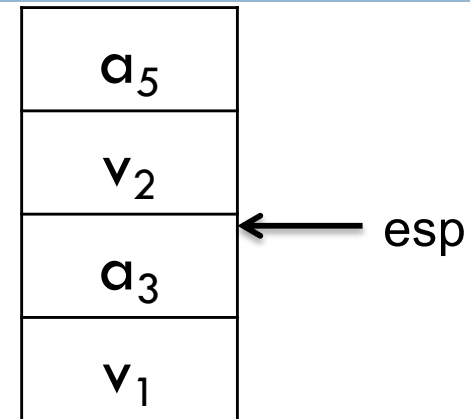

Gadgets

20

Mem[v2] = v1

Desired Logic

eax	v ₁
ebx	
eip	a ₃



Stack

```
a1: pop eax;  
a2: ret  
a3: pop ebx;  
a4: ret  
a5: mov [ebx], eax
```

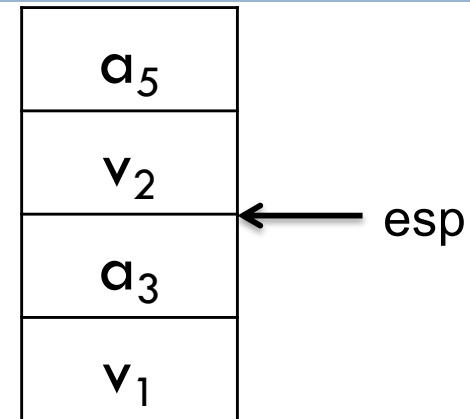
Gadgets

21

Mem[v2] = v1

Desired Logic

eax	v ₁
ebx	
eip	a ₃



Stack

```
a1: pop eax;  
a2: ret  
a3: pop ebx;  
a4: ret  
a5: mov [ebx], eax
```

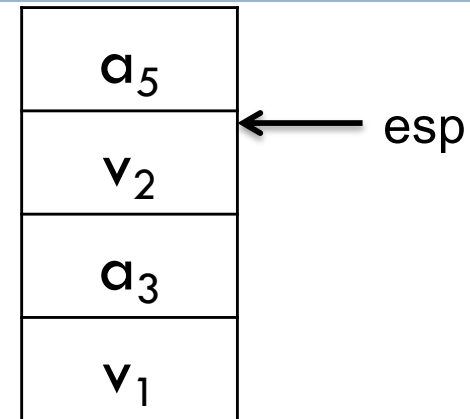
Gadgets

21

Mem[v2] = v1

Desired Logic

eax	v ₁
ebx	v ₂
eip	a ₃



Stack

```
a1: pop eax;  
a2: ret  
a3: pop ebx;  
a4: ret  
a5: mov [ebx], eax
```

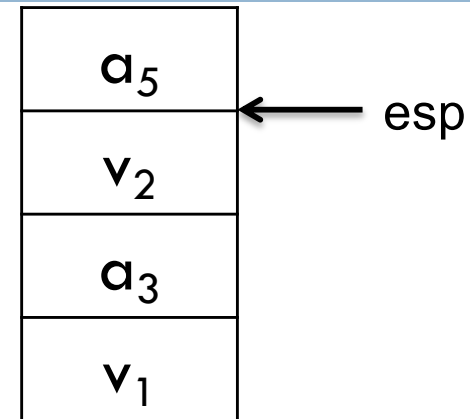
Gadgets

22

Mem[v2] = v1

Desired Logic

eax	v ₁
ebx	v ₂
eip	a ₄



Stack

```
a1: pop eax;  
a2: ret  
a3: pop ebx;  
a4: ret  
a5: mov [ebx], eax
```

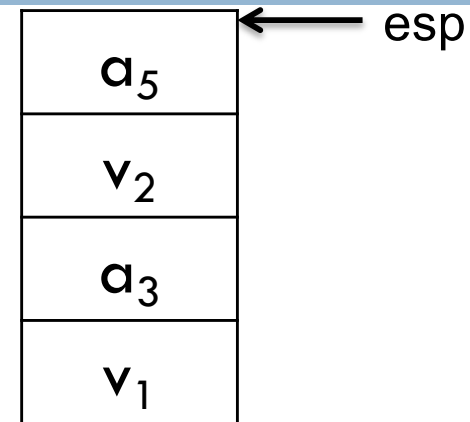
Gadgets

22

Mem[v2] = v1

Desired Logic

eax	v ₁
ebx	v ₂
eip	a ₅



Stack

```
a1: pop eax;  
a2: ret  
a3: pop ebx;  
a4: ret  
a5: mov [ebx], eax
```

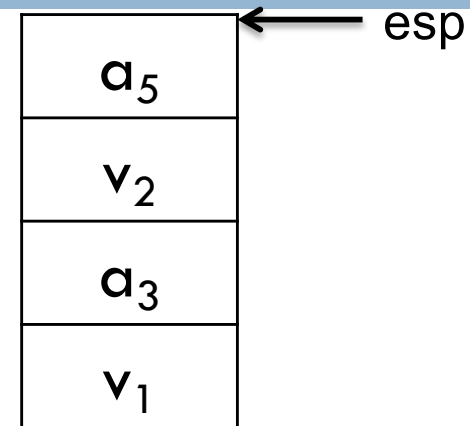
Gadgets

23

Mem[v2] = v1

Desired Logic

eax	v ₁
ebx	v ₂
eip	a ₅



Stack

```
a1: pop eax;  
a2: ret  
a3: pop ebx;  
a4: ret  
a5: mov [ebx], eax
```

Control Hijack

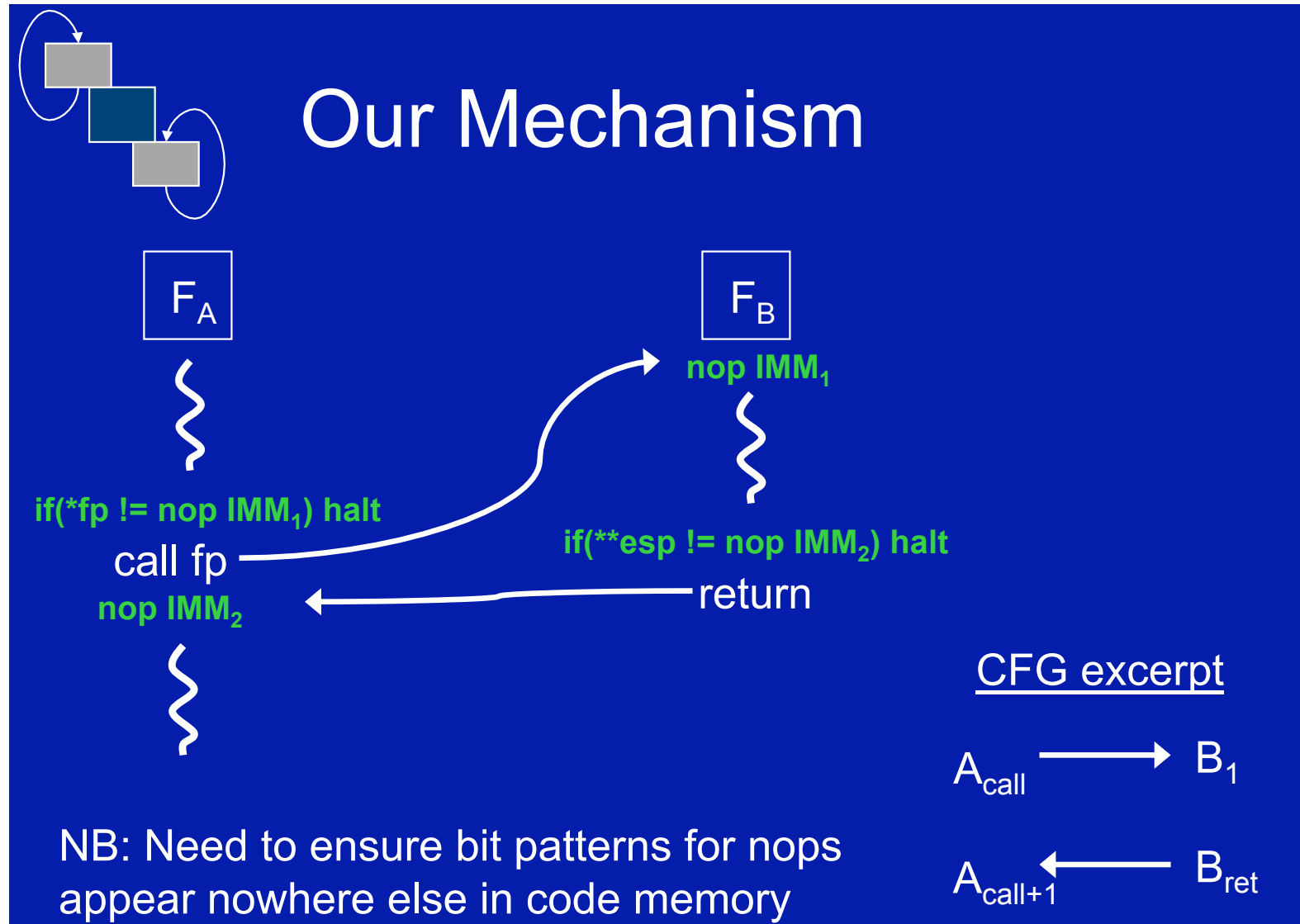
- Two main ways that C/C++ allows code targets to be computed at runtime
 - ▣ **Return address** (stack) – choose instruction to run on “ret” (i.e., function return)
 - *Why is the return address determined dynamically?*
 - ▣ **Function pointer** (stack or heap) – chooses instruction to run when invoked
 - Also called an **indirect call**
- If adversary can change either they can hijack control
- Difficult to prevent modification of code pointers
 - ▣ No broad defense at present (too expensive)

Indirect Call

- A function call using a function pointer
 - ▣ What happens?

```
int F_A()  
{  
    int (*fp)();  
    ...  
    fp = &F_B;  
    ...  
    fp();  
    ...  
}
```


Control-Flow Integrity

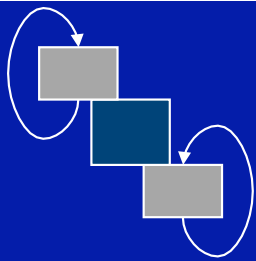


Indirect Call

- A function call using a function pointer
 - ▣ What happens?

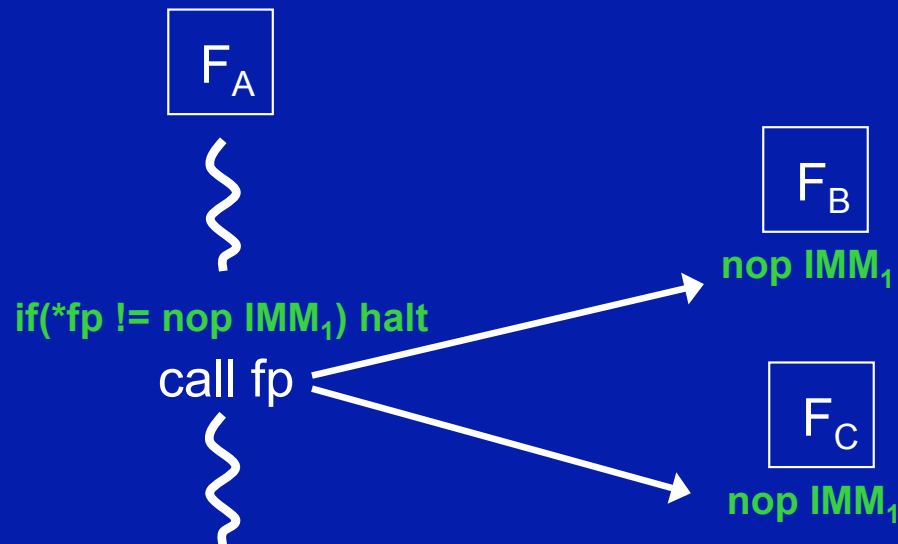
```
int F_A()  
{  
    int (*fp)();  
    ...  
    if (a > 0) fp = &F_B;  
    else fp = &F_C;  
    ...  
    fp();  
    ...  
}
```

Control-Flow Integrity

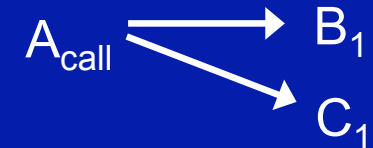


More Complex CFGs

Maybe statically all we know is that F_A can call any $\text{int} \rightarrow \text{int}$ function



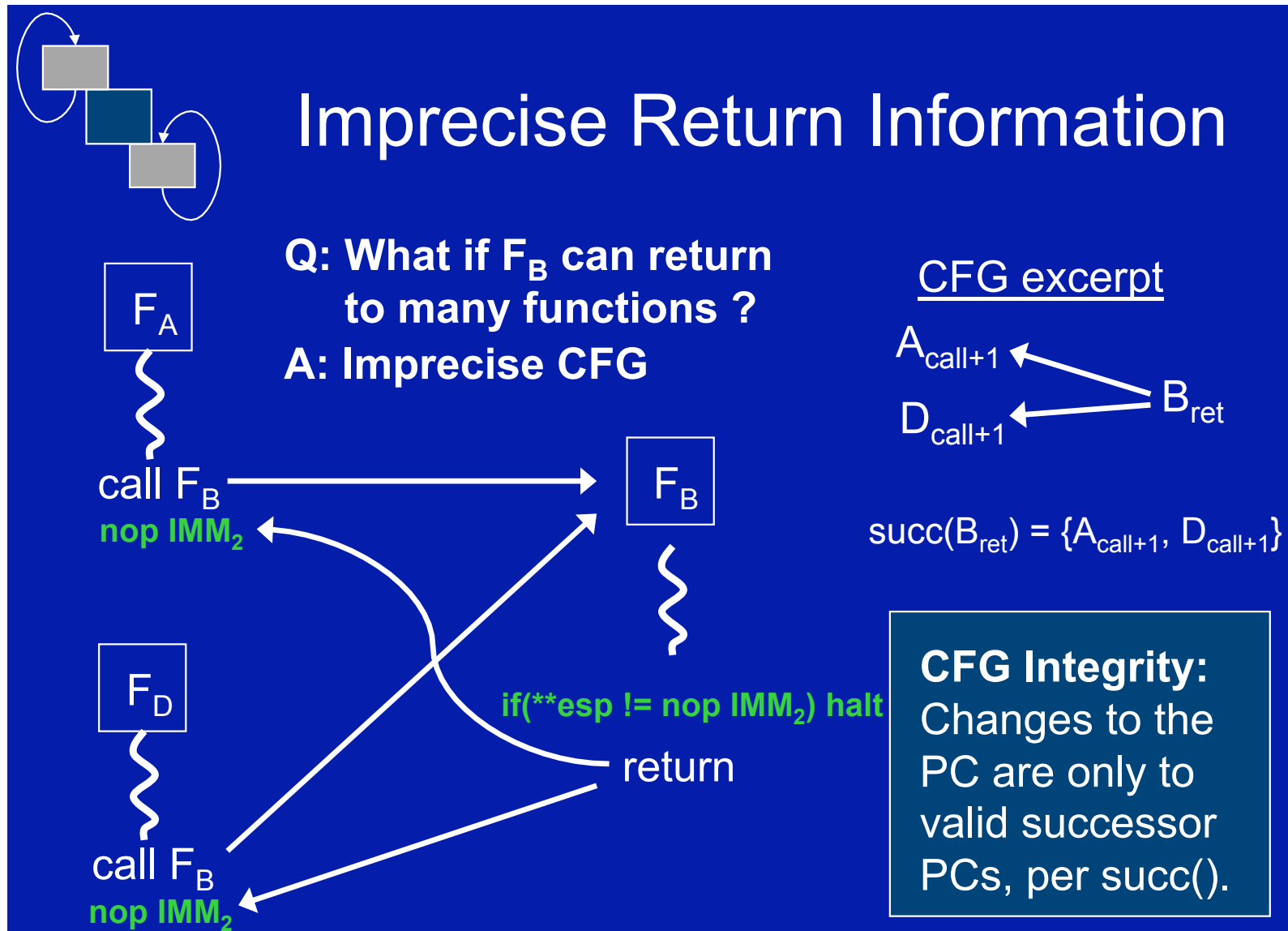
CFG excerpt



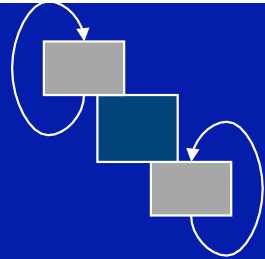
$$\text{succ}(A_{\text{call}}) = \{B_1, C_1\}$$

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

Control-Flow Integrity



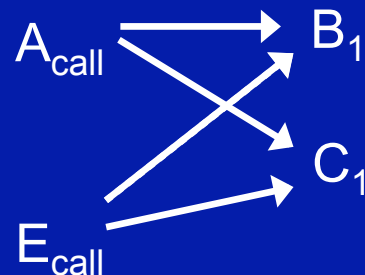
Control-Flow Integrity



No “Zig-Zag” Imprecision

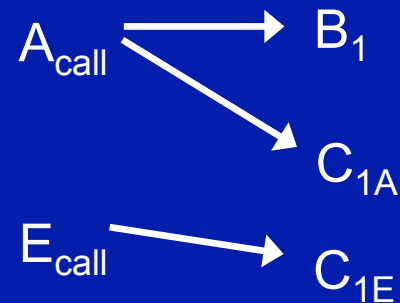
Solution I: Allow the imprecision

CFG excerpt



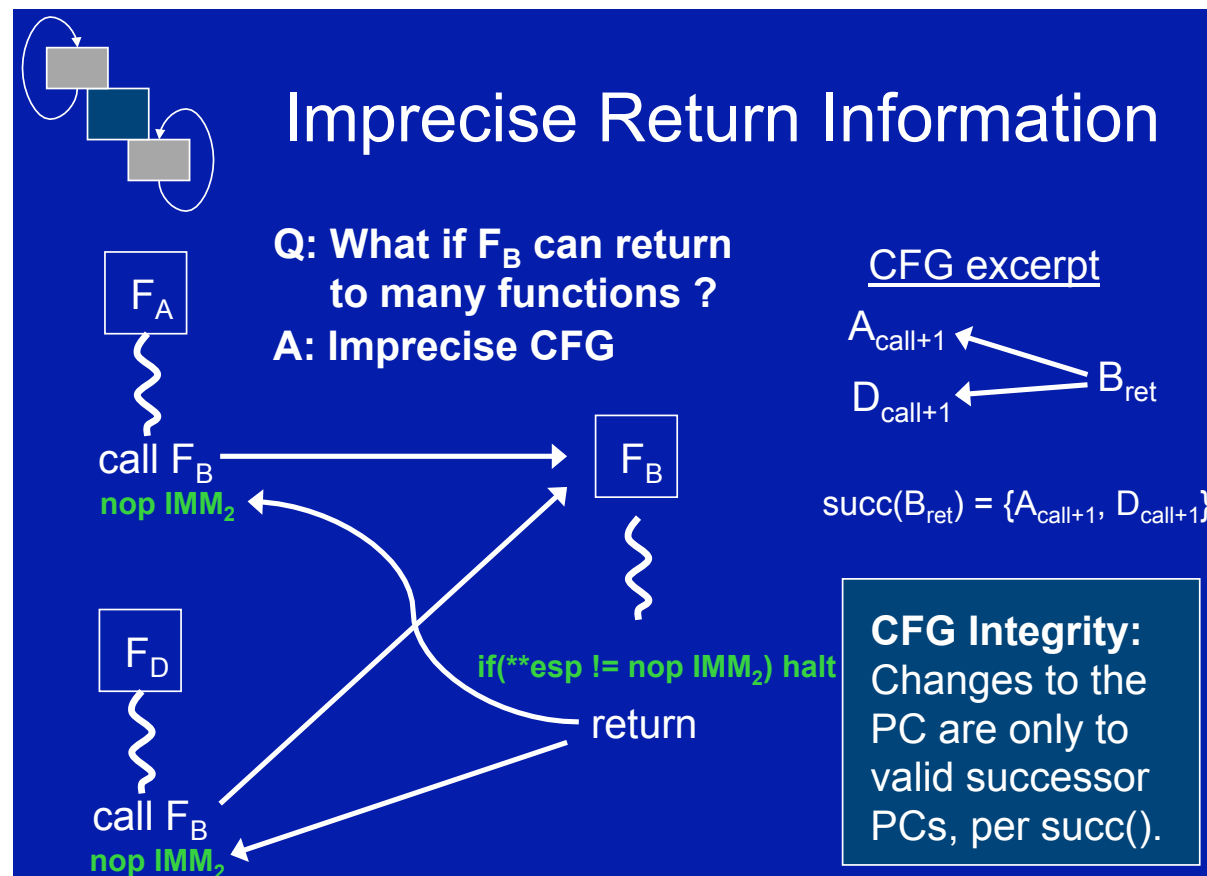
Solution II: Duplicate code to remove zig-zags

CFG excerpt



Limiting Returns

- Can't we do better for limiting returns
 - ▣ Don't we **know where a return should go**?



Shadow Stack



- Store the return address in a secure (**shadow**) location
 - ▣ Then, check that the **return address matches the shadow**



SoK: Sanitizing for Security

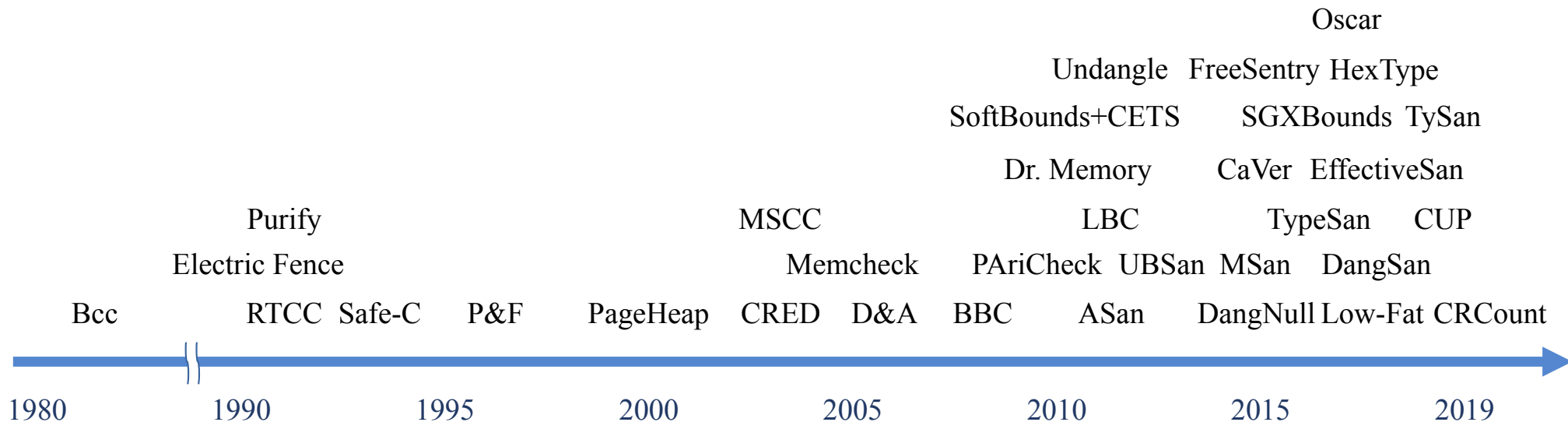
Dokyung Song, Julian Lettner, Prabhu Rajasekaran,
Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz

University of California, Irvine

Slides from Dokyung Song's Oakland presentation

Dynamic Analysis Tools for C/C++

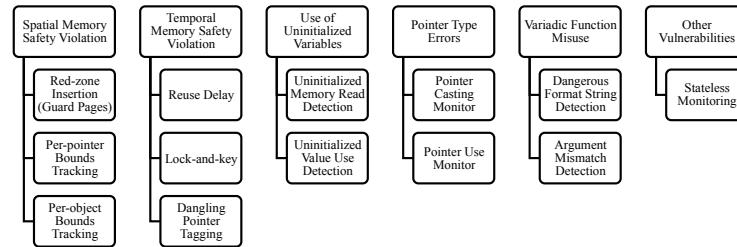
- More than 35 years of research in Dynamic Analysis Tools – often-called “*Sanitizers*” – that find vulnerabilities specific to C/C++



Sanitizers and Defenses

	Exploit Mitigation	Sanitization
<i>The goal is to ...</i>	Mitigate attacks	Find vulnerabilities
<i>Used in ...</i>	Production	Pre-release
<i>Performance budget is ...</i>	Very limited	Much higher
<i>Policy violation leads to ...</i>	Program termination	Problem diagnosis
<i>Violations triggered at location of bug</i>	Sometimes	Always
<i>Tolerance for FPs is ...</i>	Zero	Somewhat higher
<i>Surviving benign errors is ...</i>	Desired	Not desired

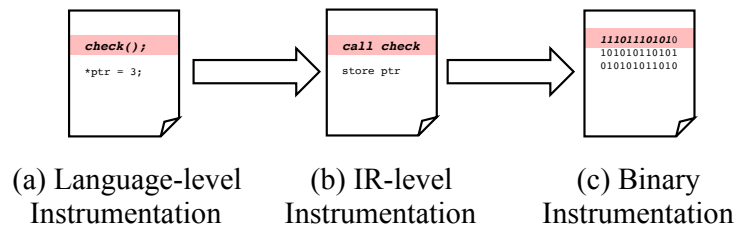
Implementation



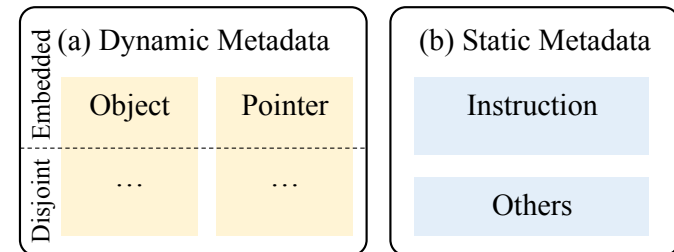
Bug Finding Technique



Program Instrumentation

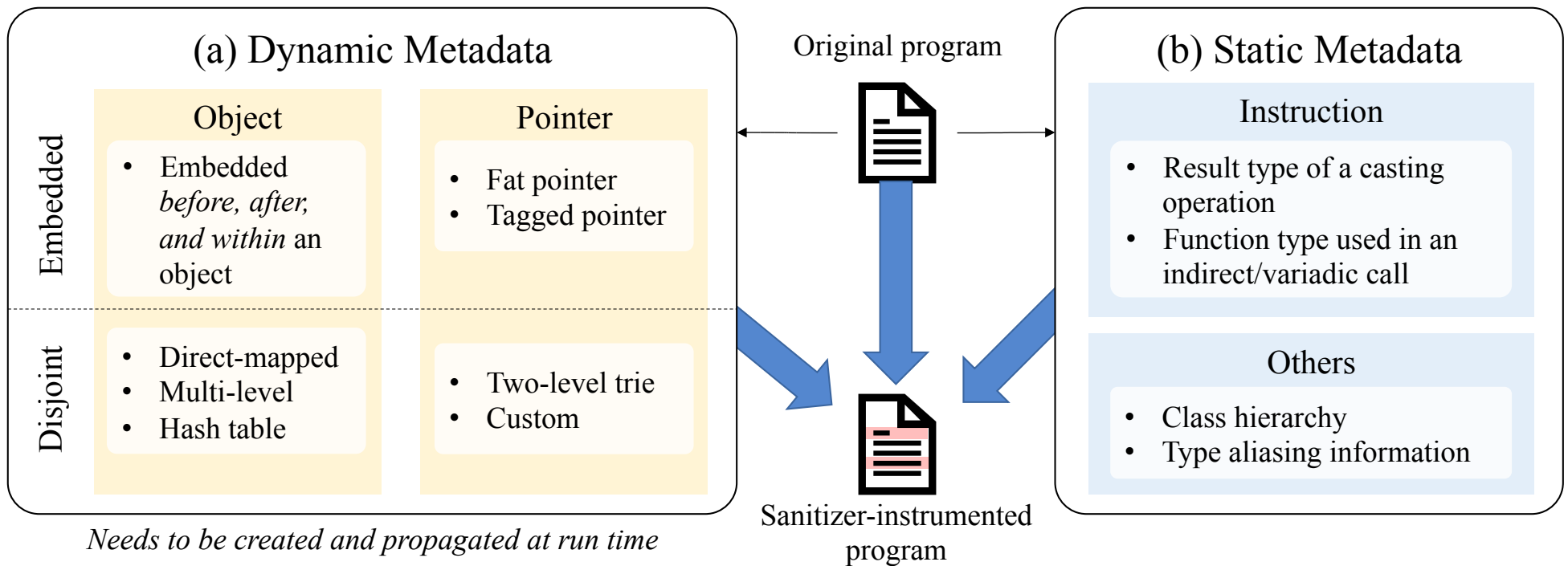


Metadata Management






Implementation

Metadata Management



Tools

- Our analysis of 37 tools
 - We benchmarked 10 publicly available sanitizers on the same experimental platform (<https://github.com/seccure-systems-lab/sanitizing-for-security-benchmarks>)
- Main observations
 - Performance is not a primary concern
 - Many false positives (marked as ) in tools other than widely-used ones such as ASan
 - Most c ly have partial coverage of bugs ()
 - Widely deployed tools such as ASan have even smaller coverage

[illegible]

Take Away



- Current defenses for memory safety are incomplete and can be evaded
 - ▣ Return-oriented programming attacks bypass all defenses
- Defenses such as CFI can thwart many ROP vectors
 - ▣ But, not all
- Sanitizers can prevent attacks/detect bugs, but at a non-trivial cost currently
 - ▣ And are incomplete for testing
 - ▣ How do we improve the situation?

Questions

43

