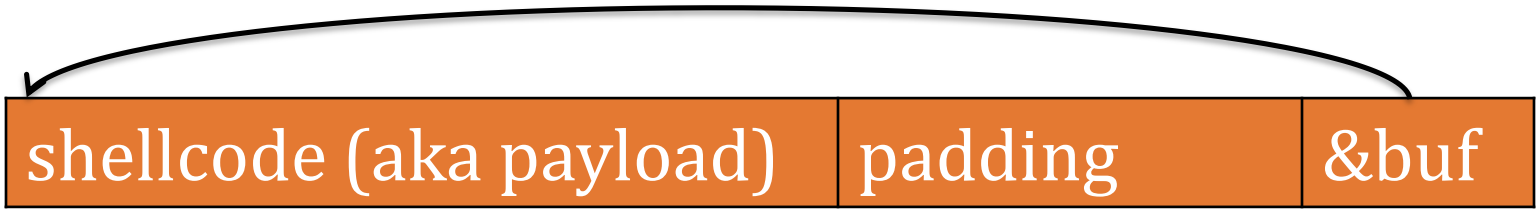
David Brumley
Carnegie Mellon University

Credit: Some slides from Ed Schwartz

Control Flow Hijack: Always control + computation

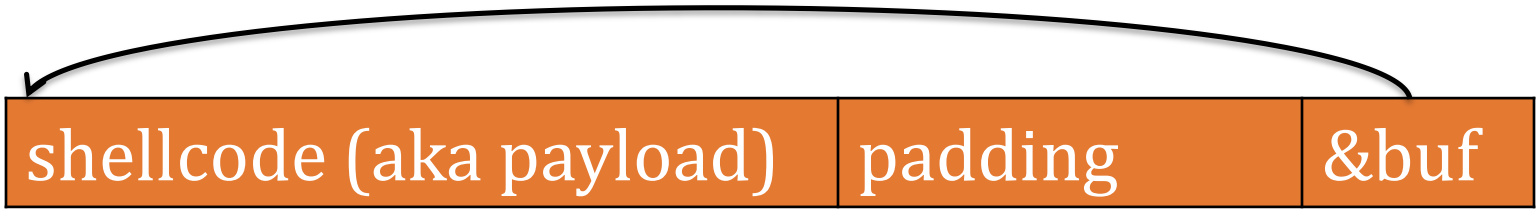


computation + ***control***

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f”
"\x73\x68\x68\x2f\x62\x69\x6e\x89”
"\xe3\xb0\x0b\xcd\x80”;
```

Previously: Executable code as input

Control Flow Hijack: Always control + computation



computation + ***control***

Today: Return Oriented Programming
Execution without injecting code

ROP Overview

Idea:

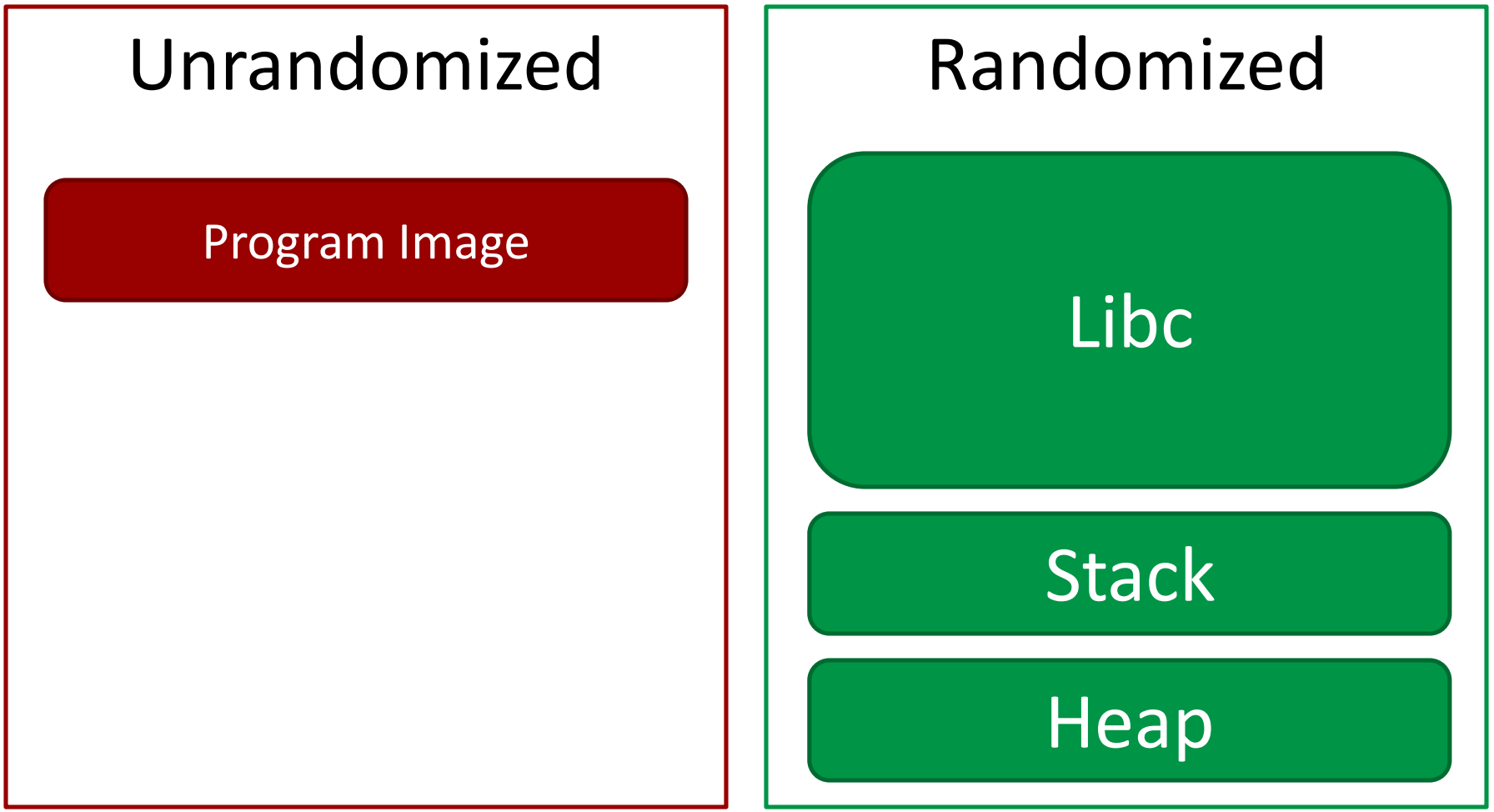
We forge shell code out of existing application logic gadgets

Requirements:

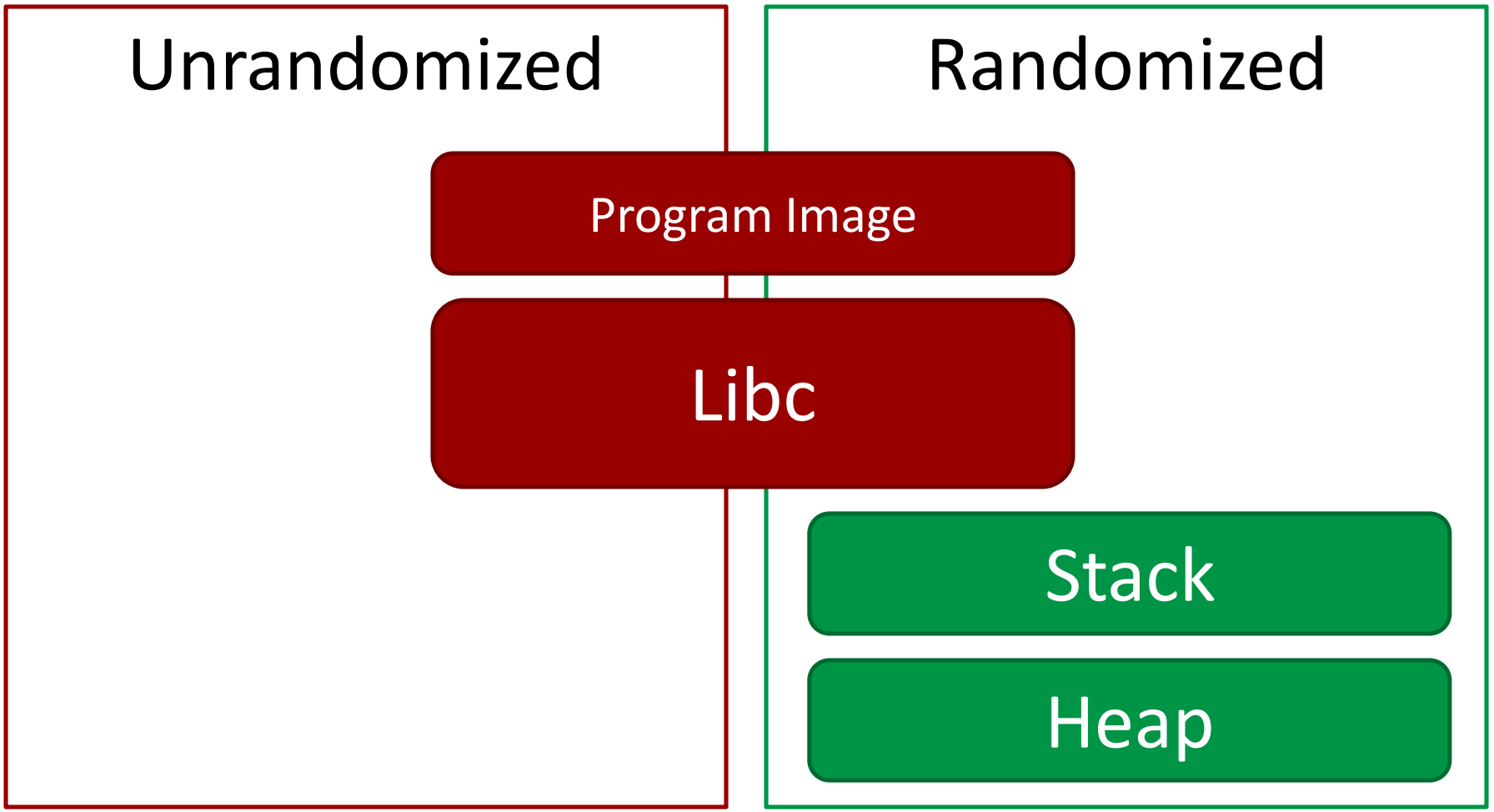
vulnerability + gadgets + some *unrandomized* code

(we need to know the addresses of gadgets)

ASLR on Linux



ASLR on Windows



Check with EMET tool

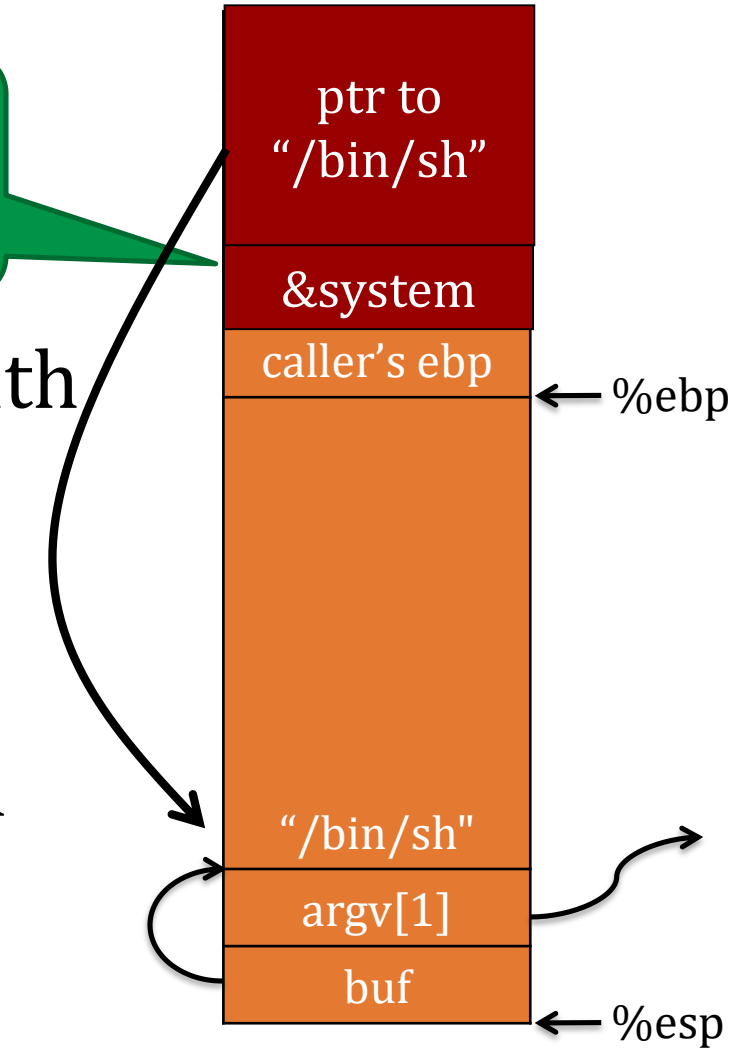
Motivation: Return-to-libc Attack

ret transfers control to `system`, which finds arguments on stack

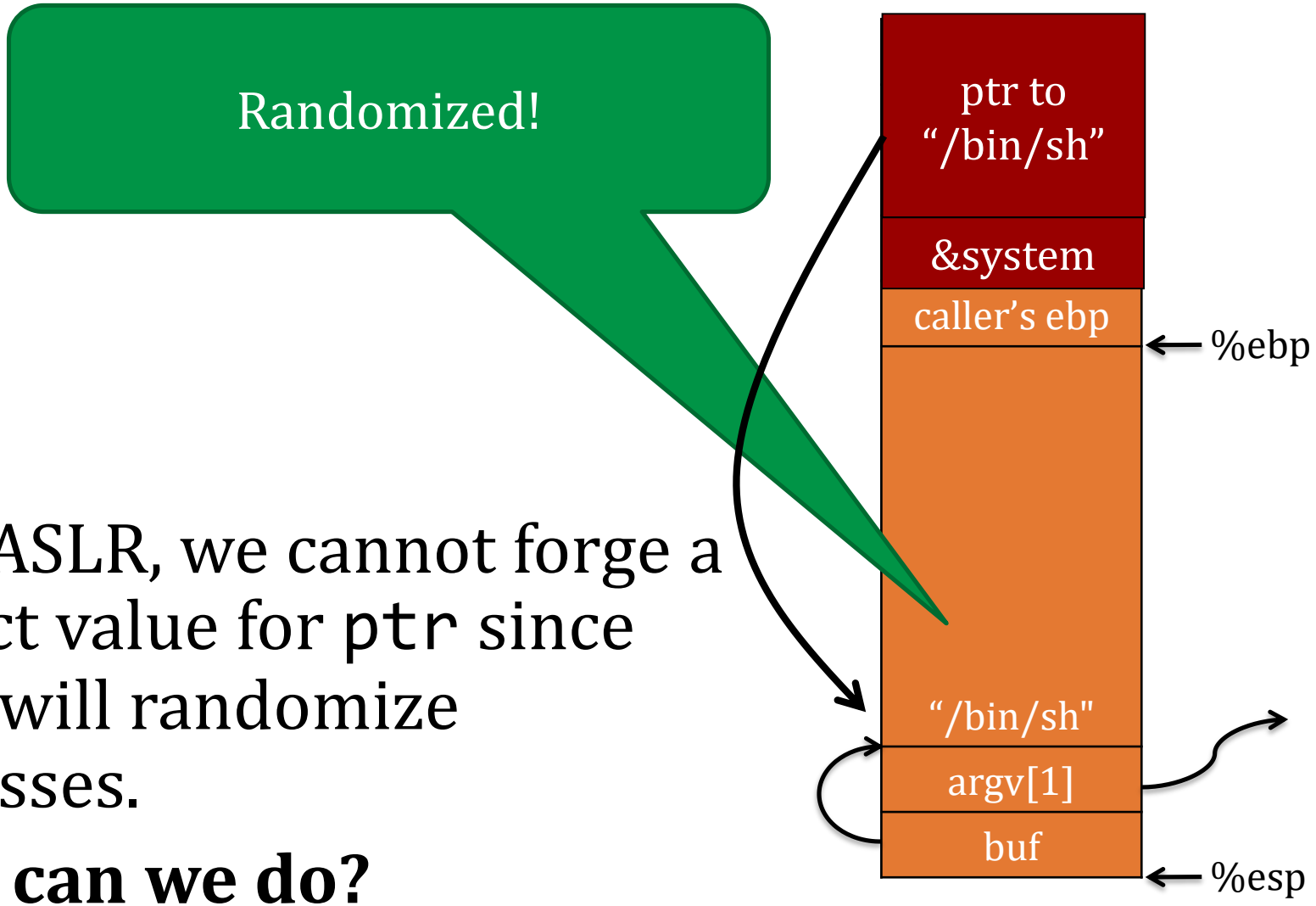
Overwrite return address with *address* of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!



Question



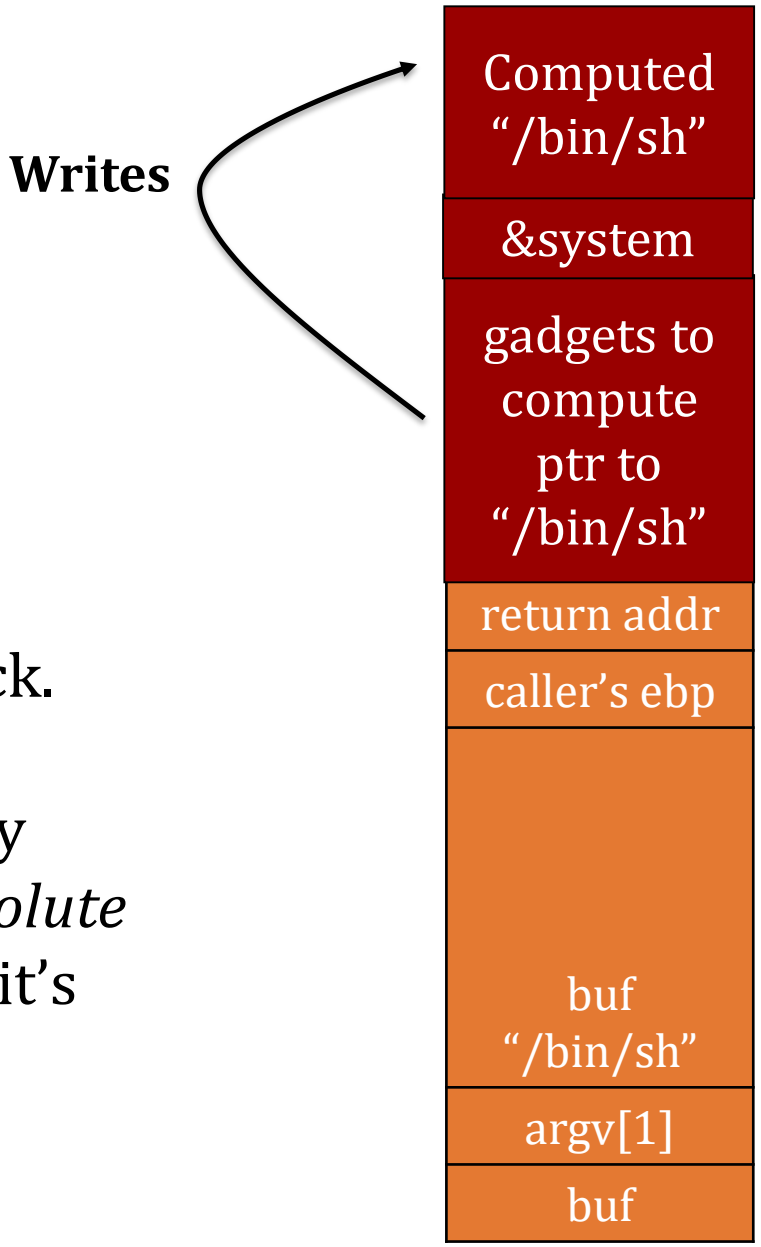
With ASLR, we cannot forge a correct value for `ptr` since ASLR will randomize addresses.

What can we do?

Idea!

Get a copy of ESP to calculate address of "/bin/sh" on randomized stack.

This works because ASLR only protects against knowing *absolute* addresses, while we will find it's *relative address*.



Return Oriented Programming Techniques

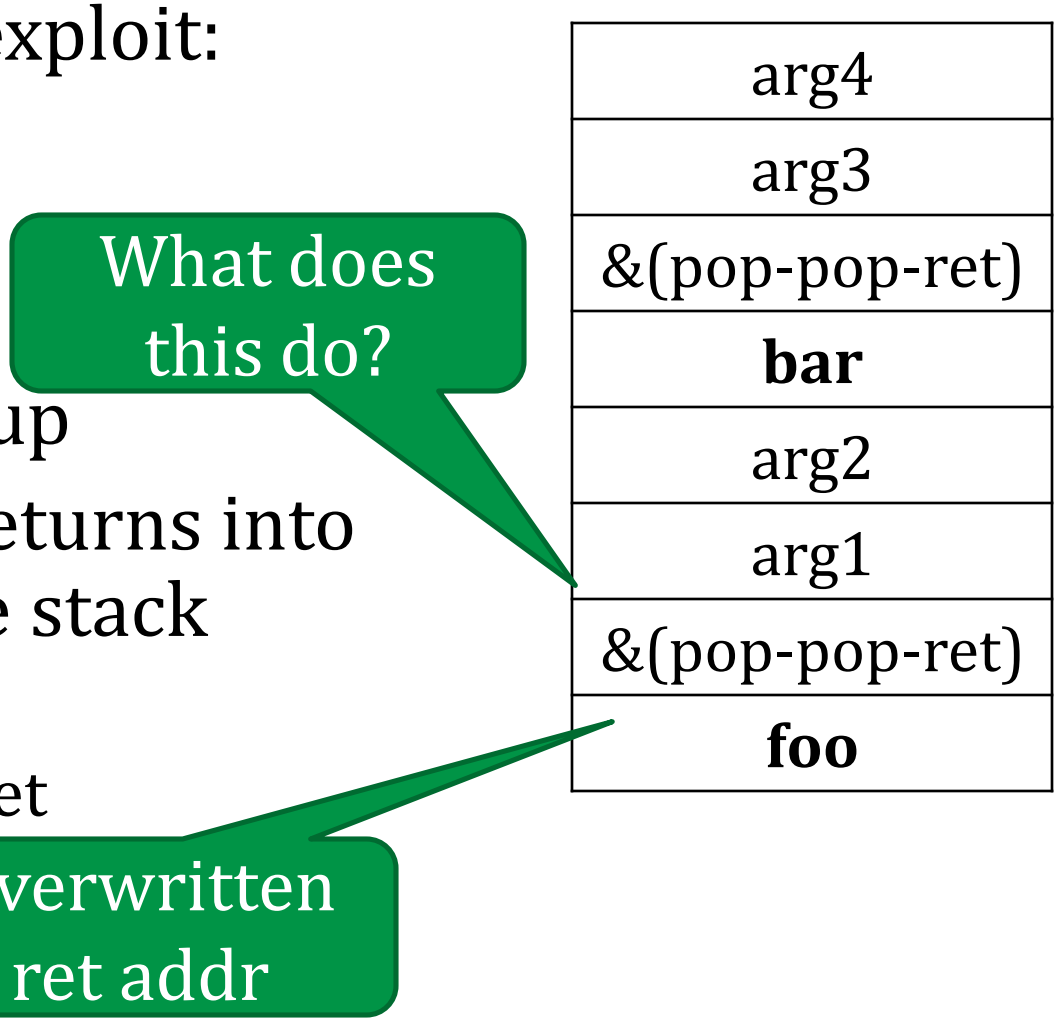
1. Return chaining
2. Semantic equivalence
3. ROP on Windows

Return Chaining

Suppose we want to call 2 functions in our exploit:

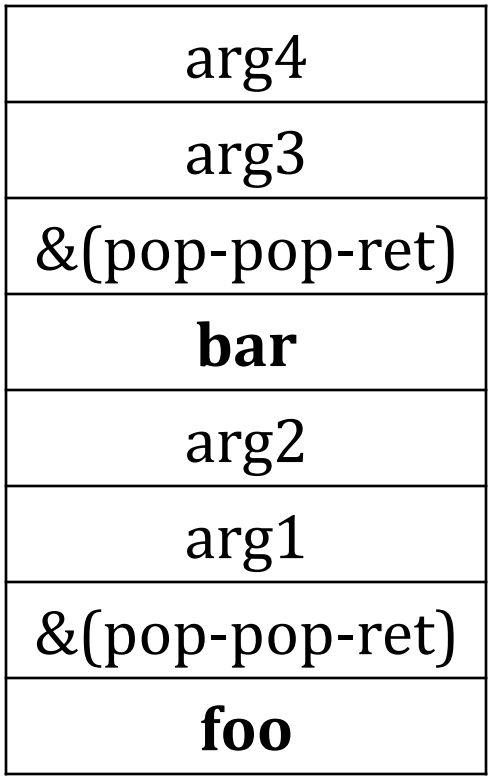
foo(arg1, arg2)
bar(arg3, arg4)

- Stack unwinds up
- First function returns into code to advance stack pointer
  - e.g., pop; pop; ret



Return Chaining

- When **foo** is executing, &pop-pop-ret is at the saved EIP slot.
- When **foo** returns, it executes pop-pop-ret to clear up arg1 (pop), arg2 (pop), and transfer control to **bar** (ret)



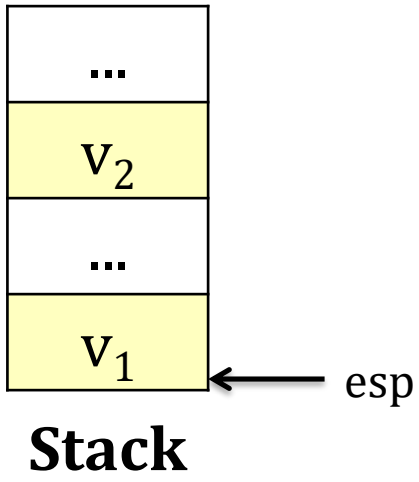
There are many
semantically equivalent
ways to achieve the same net shellcode effect

Let's practice thinking in gadgets

An example operation

`Mem[v2] = v1`

Desired Logic



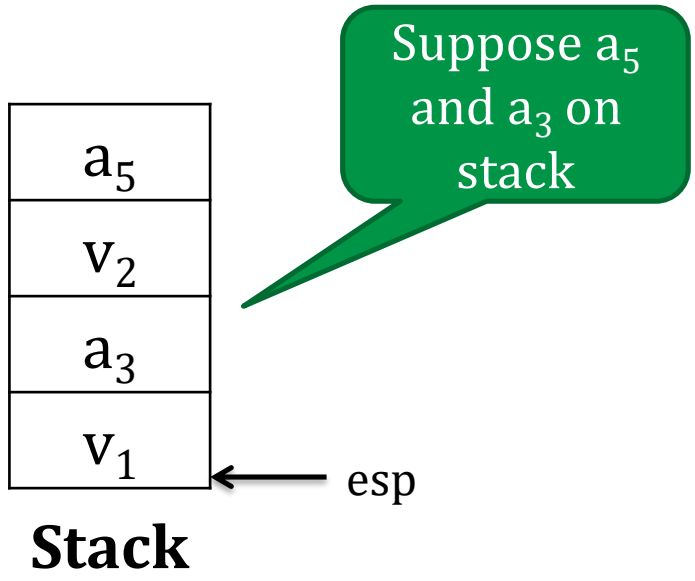

```
a1: mov eax, [esp] ; eax has v1
a2: mov ebx, [esp+8] ; ebx has v2
a3: mov [ebx], eax ; Mem[v2] = eax
```

Implementation 1

implementing with gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	
eip	a_1

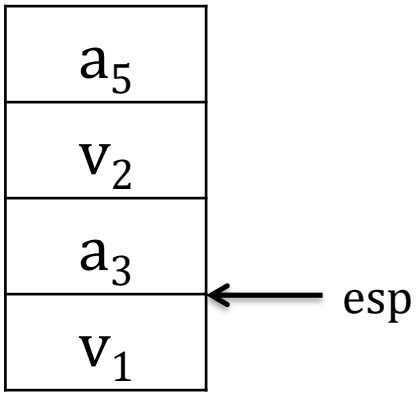
```
a1: pop eax
a2: ret
a3: pop ebx
a4: ret
a5: mov [ebx], eax
```

Implementation 2

implementing with gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	
eip	a_3

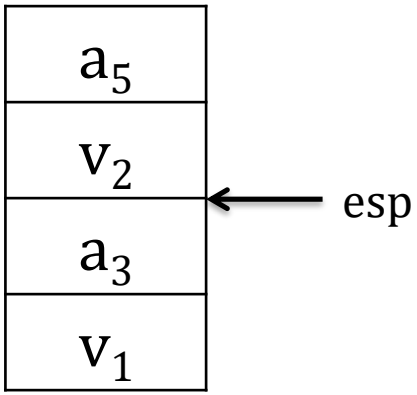
```
a1: pop eax
a2: ret
a3: pop ebx
a4: ret
a5: mov [ebx], eax
```

Implementation 2

implementing with gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_3

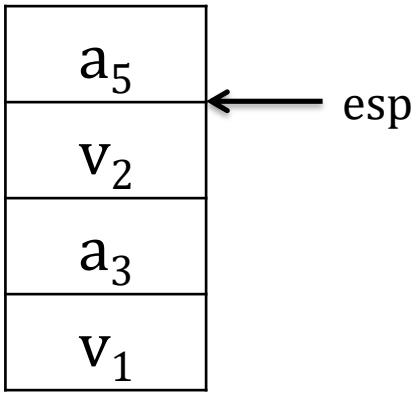
```
a1: pop eax
a2: ret
a3: pop ebx
a4: ret
a5: mov [ebx], eax
```

Implementation 2

implementing with gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_4

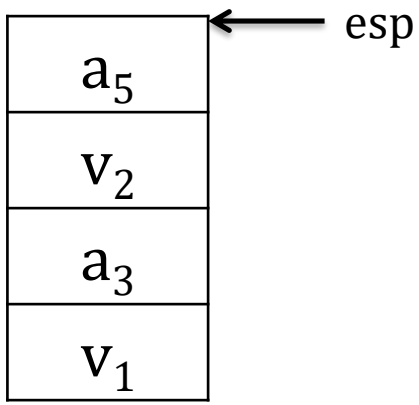
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

implementing with gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_5

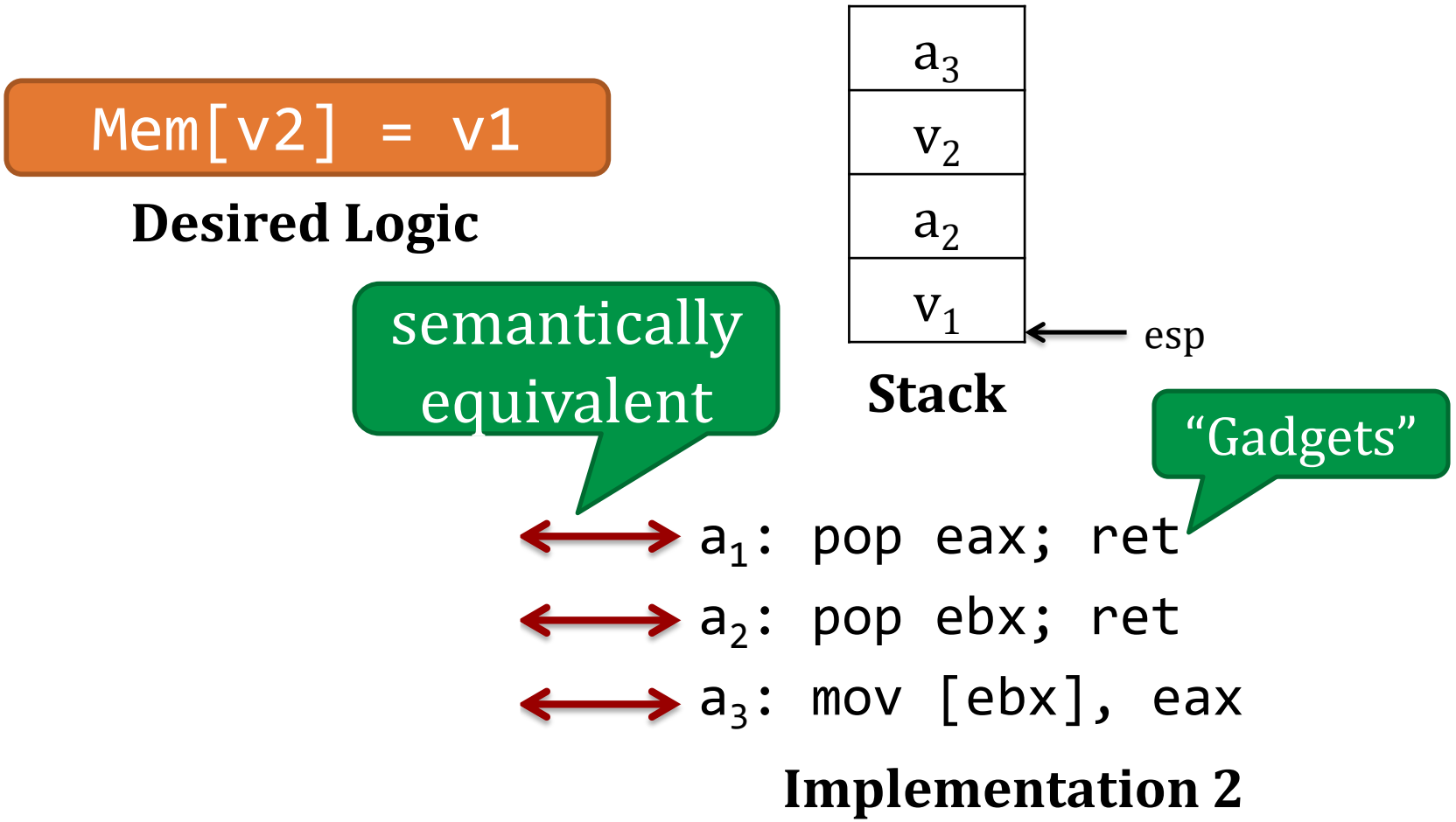
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Gadget 1

Gadget 2

Implementation 2

Equivalence



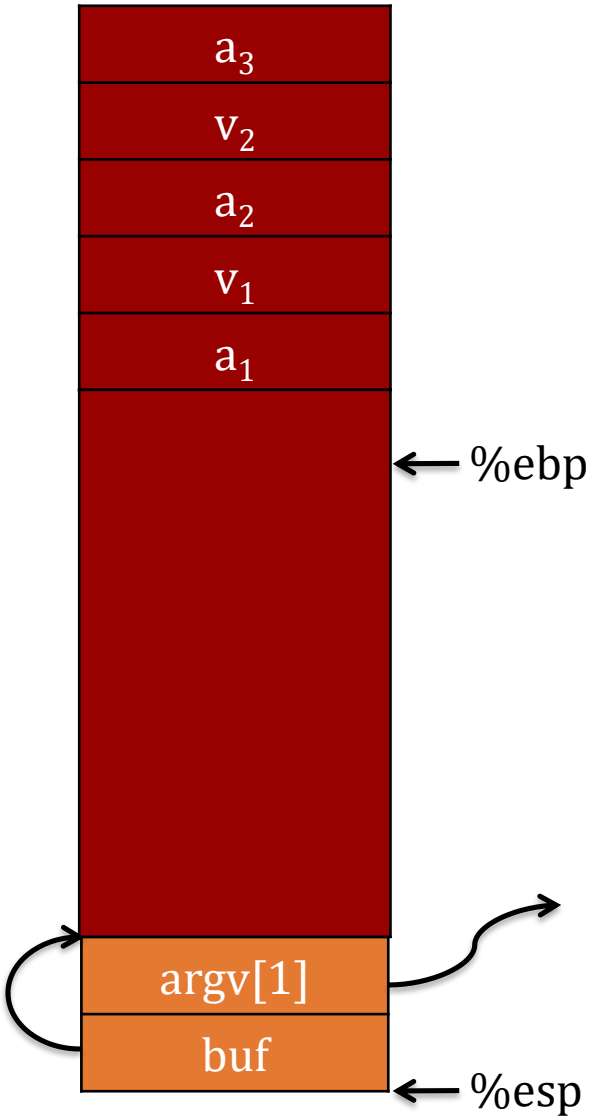
Return-Oriented Programming (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

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

Desired store executed!



Gadgets

- A gadget is a set of instructions for carrying out a semantic action
  - mov, add, etc.
- Gadgets typically have a number of instructions
  - One instruction = native instruction set
  - More instructions = synthesize <- ROP
- Gadgets in ROP generally (but not always) end in return

Image by Dino Dai Zovi

Quiz

```
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
```

ret instr

a_1: add $0x80, %eax; pop %ebp; ret

a_2: pop %eax; ret

Known Gadgets

Draw a stack diagram and ROP exploit to pop a value 0xBBBBBBBB into eax and add 0x80.

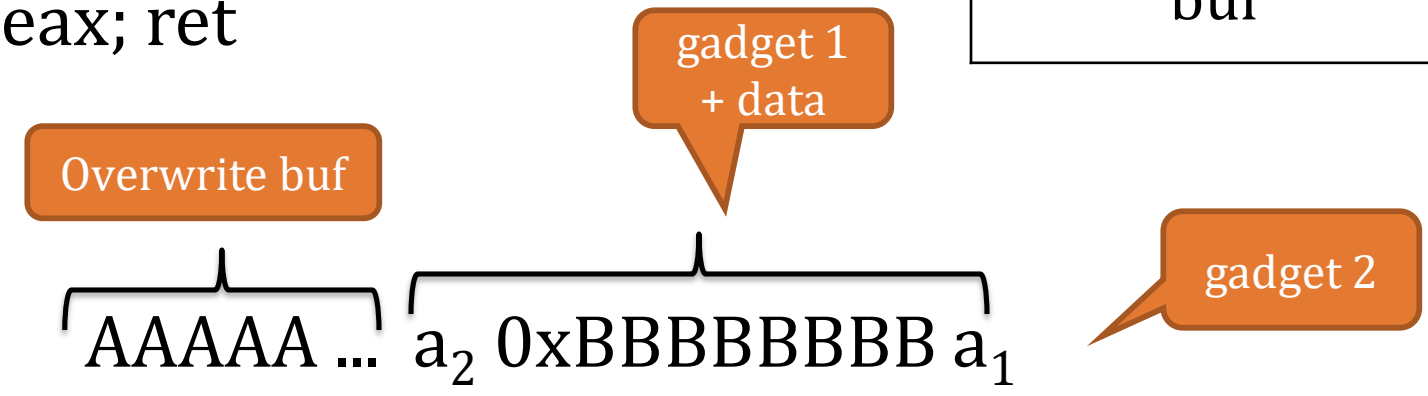
Quiz

```
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
```

a_1: add eax, 0x80; pop %ebp; ret

a_2: pop %eax; ret

<data for pop ebp>
a_1
0xBBBBBBBB
a_2
saved ebp
buf

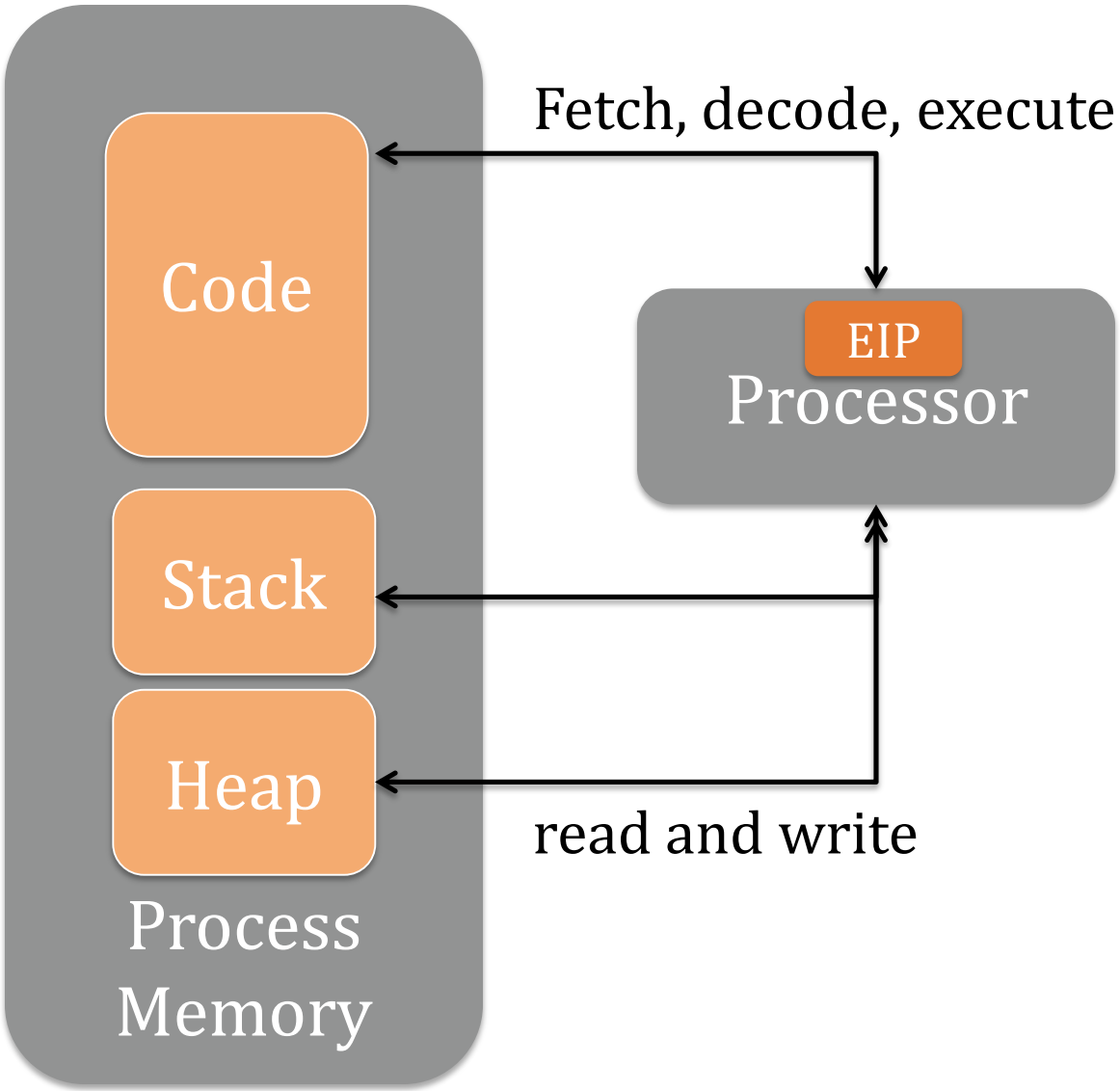


RO(P?) Programming

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

Disassembling Code

Recall: Execution Model



Disassembly

```
user@box:~/l2$ objdump -d ./file
...
00000000 <even_sum>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 10                sub    $0x10,%esp
   6:   8b 45 0c                mov    0xc(%ebp),%eax
   9:   03 45 08                add    0x8(%ebp),%eax
   c:   03 45 10                add    0x10(%ebp),%eax
   f:   89 45 fc                mov    %eax,0xfffffffc(%ebp)
  12:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
  15:   83 e0 01                and    $0x1,%eax
  18:   84 c0                   test   %al,%al
  1a:   74 03                   je     1f <even_sum+0x1f>
  1c:   ff 45 fc                incl   0xfffffffc(%ebp)
  1f:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
  22:   c9                      leave
  23:   c3                      ret
```

Disassemble

Address

Executable instructions

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler EIP

Algorithm:

1. Decode Instruction
2. Advance EIP by len

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	Valid	Valid	Push *r/m16*.
FF /6	PUSH *r/m32*	N.E.	Valid	Push *r/m32*.
FF /6	PUSH *r/m64*	Valid	N.E.	Push *r/m64*. Default operand size 64-bits.
50+*rw*	PUSH *r16*	Valid	Valid	Push *r16*.
50+*rd*	PUSH *r32*	N.E.	Valid	Push *r32*.
50+*rd*	PUSH *r64*	Valid	N.E.	Push *r64*. Default operand size 64-bits.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5

```
push  ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler EIP

MOV—Move

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8**** *r8****	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	Valid	Valid	Move *r32* to *r/m32*.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

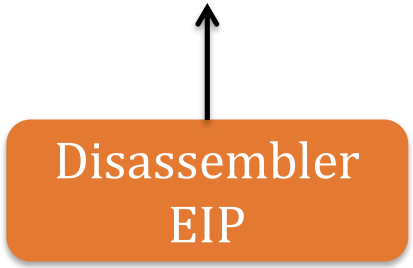
r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
...										
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD

```
push ebp
mov %esp, %ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9



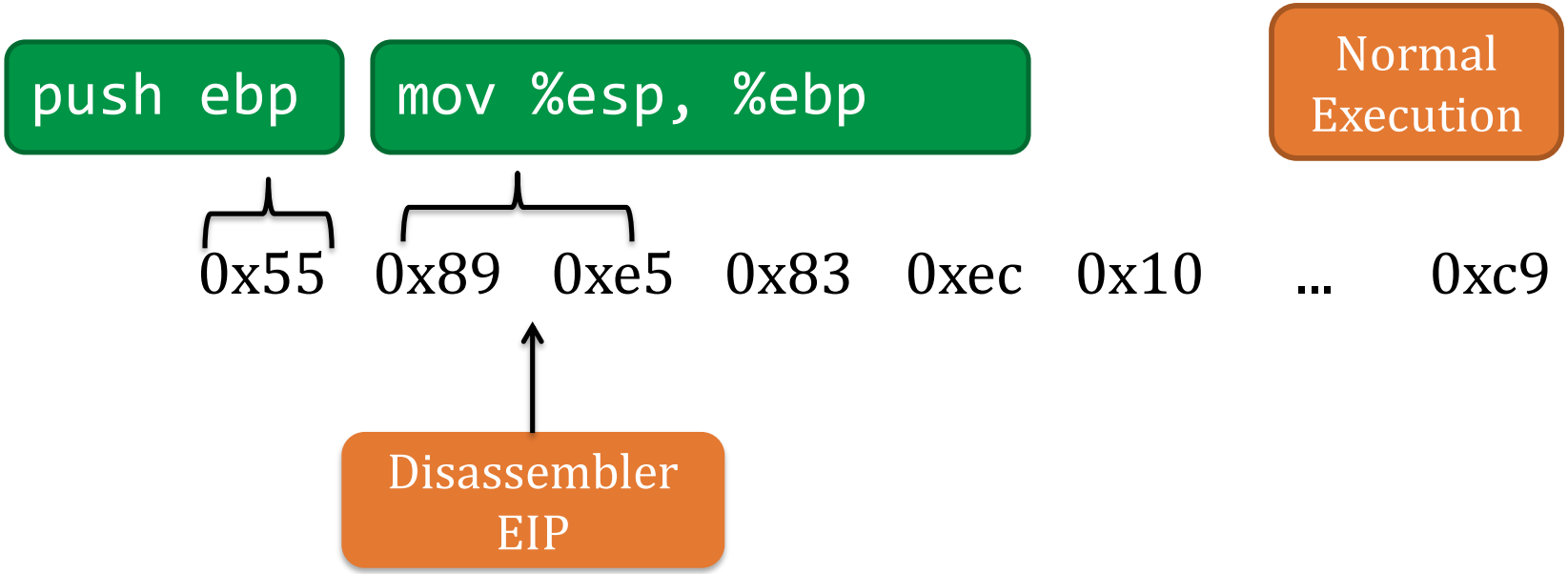
Note we don't follow jumps: we just increment by instruction length

Algorithm:

1. Decode Instruction
2. Advance EIP by len

```
push ebp
mov %esp, %ebp
```

Disassemble from any address



It's perfectly valid to start disassembling from *any* address and all byte sequences will have a unique disassembly

Recursive Descent

- Follow jumps and returns instead of linear sweep

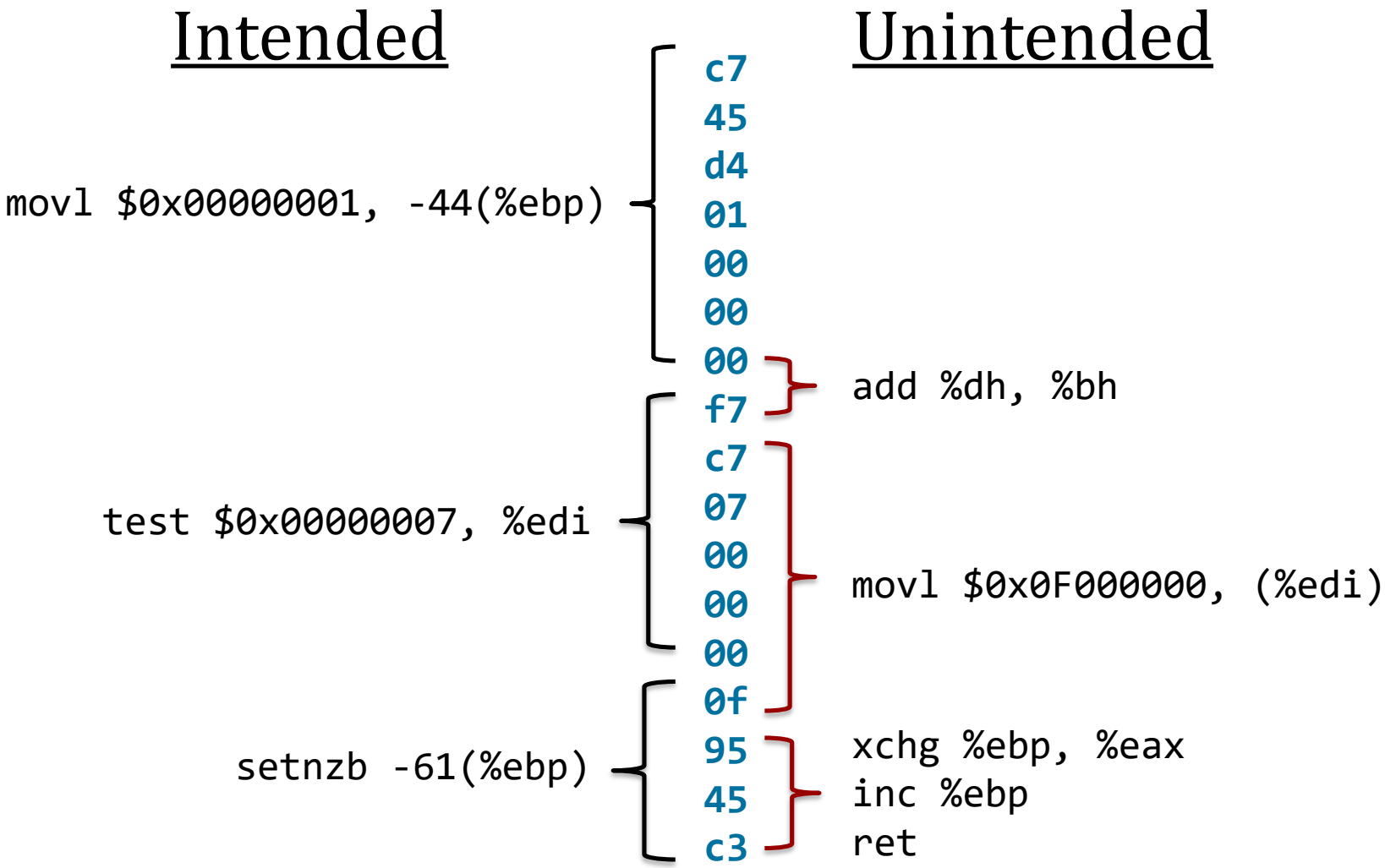
- Undecidable: indirect jumps
  - Where does `jmp *eax` go?

ROP and Disassembly

Compiler-created gadget: A sequence of instructions inserted by the compiler ending in `ret`.

Unintended gadget: A sequence of instructions not created by the compiler, e.g., by starting disassembly at an unaligned start.

Example: ecb_crypt()



Slide credit: Hovav Shacham Blackhat 2008 Talk

ROP Programming

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

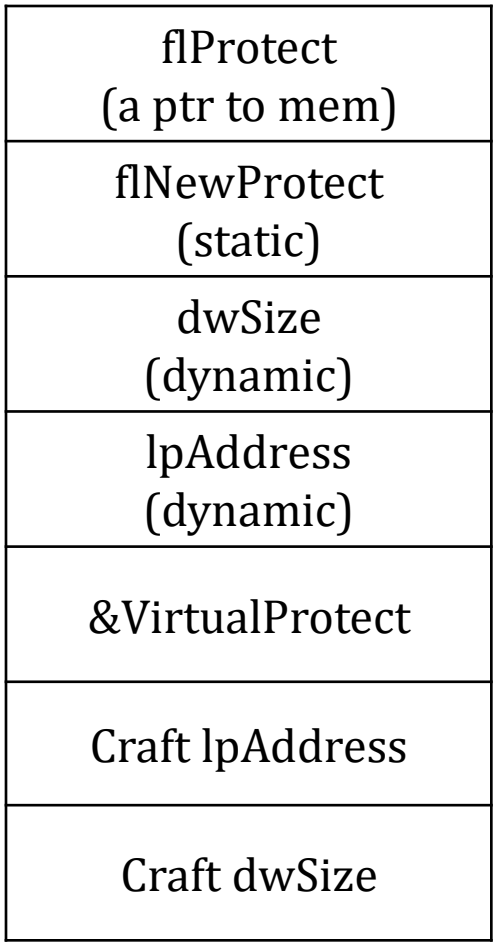
ROPing Windows

```
BOOL WINAPI VirtualProtect(
    LPVOID lpAddress, // dynamically determined base addr
to pages to change
    SIZE_T dwSize, // size of the region in bytes
    DWORD flNewProtect, // 0x40 = EXECUTE_READWRITE
    PDWORD flProtect // A ptr to a variable for prev. arg
);
```

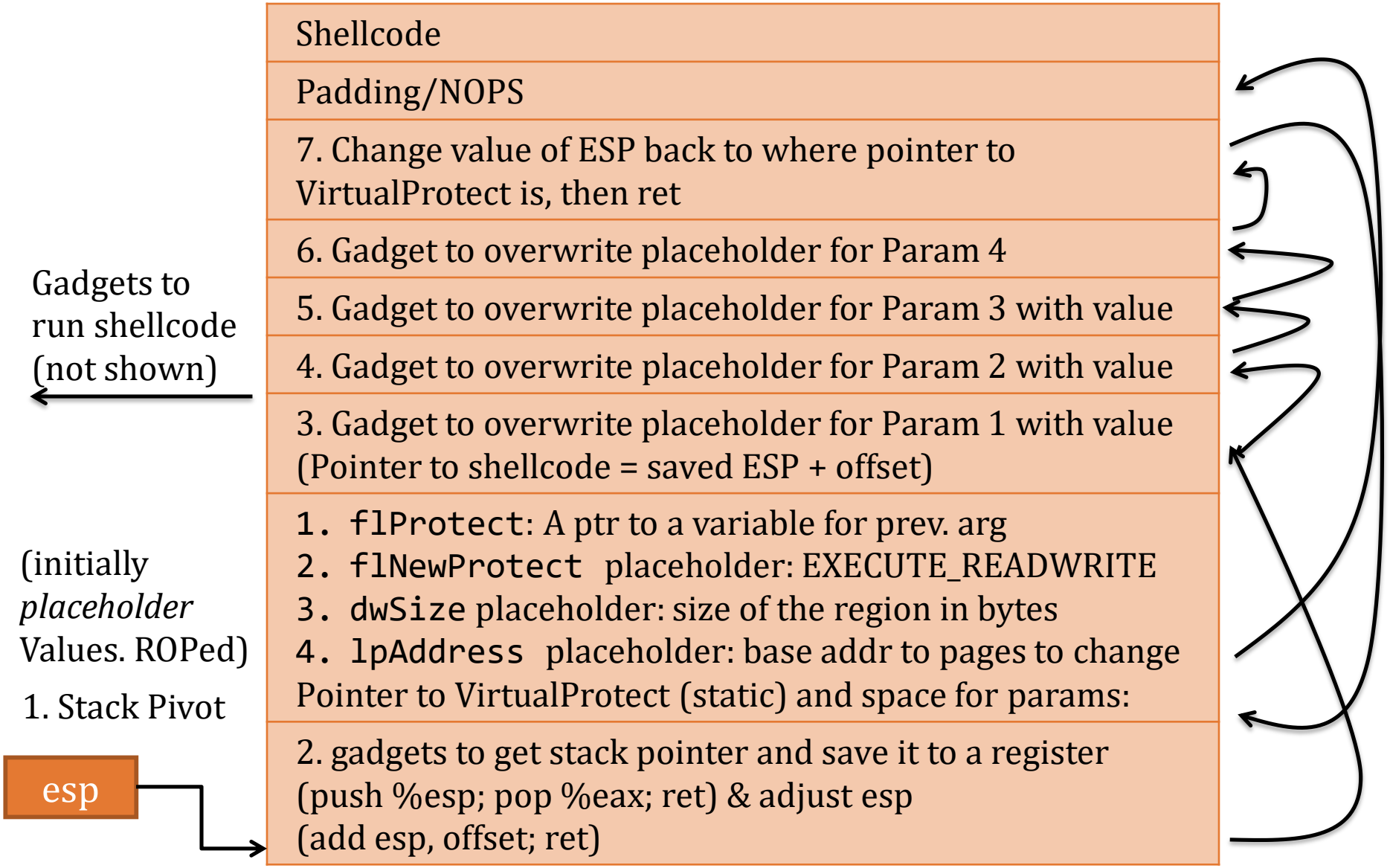
VirtualProtect() can un-DEP a memory region

VirtualProtect Diagram

```
LPVOID WINAPI VirtualProtect(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD DWORD flNewProtect,
    DWORD flProtect
);
```



ROPing Windows: An Example Exploit (pre-Win 8)



From https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

Stack Pivots
Pointing `esp` to controlled data

Fact: Functions often access arguments with respect to esp

Defn: A stack pivot redirects esp at attacker-controlled data

Example: Attacker controls heap data pointed to be `ESI`. One stack pivot may be:

```
xchg esi, esp; ret
```

Now `esp` points to the attacker-controlled data.

Other References

Thorough introduction:

https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

Adopting to Win8:

http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/

ROP Programming

1. Disassemble code
2. Identify *useful* code sequences ending in ret as gadgets
3. Assemble gadgets into desired shellcode

Disassemble all sequences ending in `ret`

ROP: Shacham et al.

1. Disassemble code
2. Identify *useful* code sequences as gadgets ending in ret
3. Assemble gadgets into desired shellcode

Automatic

Hard work*

* There is research like Q (a CMU ROP compiler) that automates this

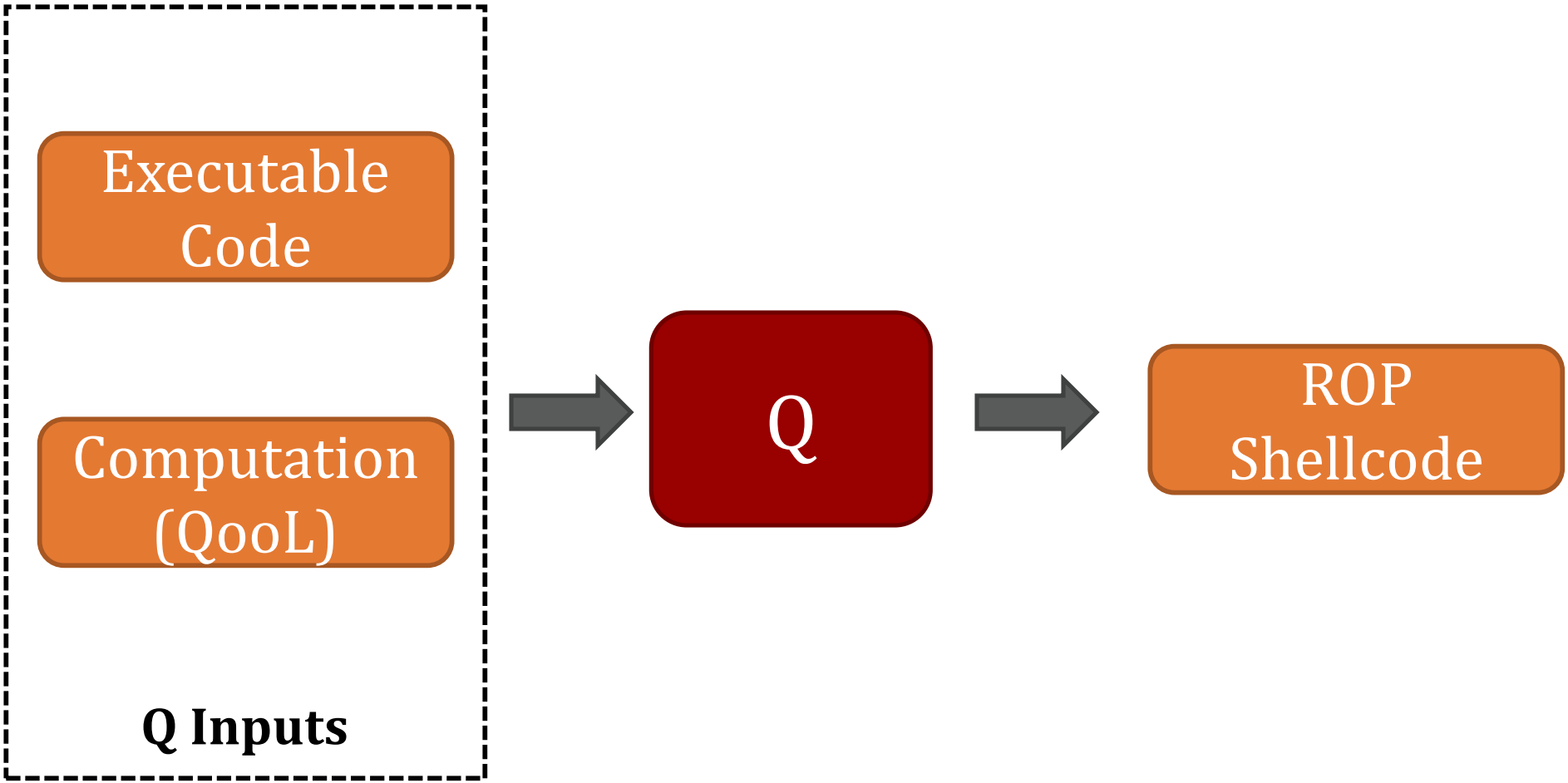

Questions?

END

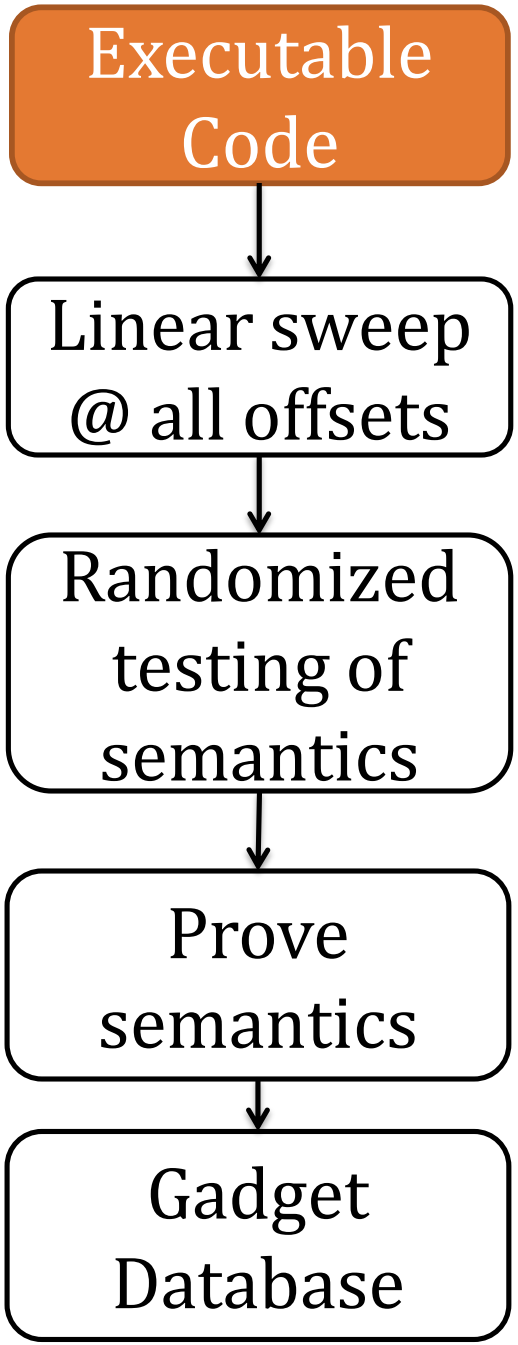
Q: Automating ROP

1. Q: Exploit Hardening Made Easy, Schwartz et al, USENIX Security 2011

Overview*



* Exploit hardening step not discussed here.



Like before

Step 1: Disassemble code

Step 2: Identify *useful* code sequences (not necessarily ending in ret)

"useful" = Q-Op

Q-Ops (aka Q Semantic Types) (think instruction set architecture)

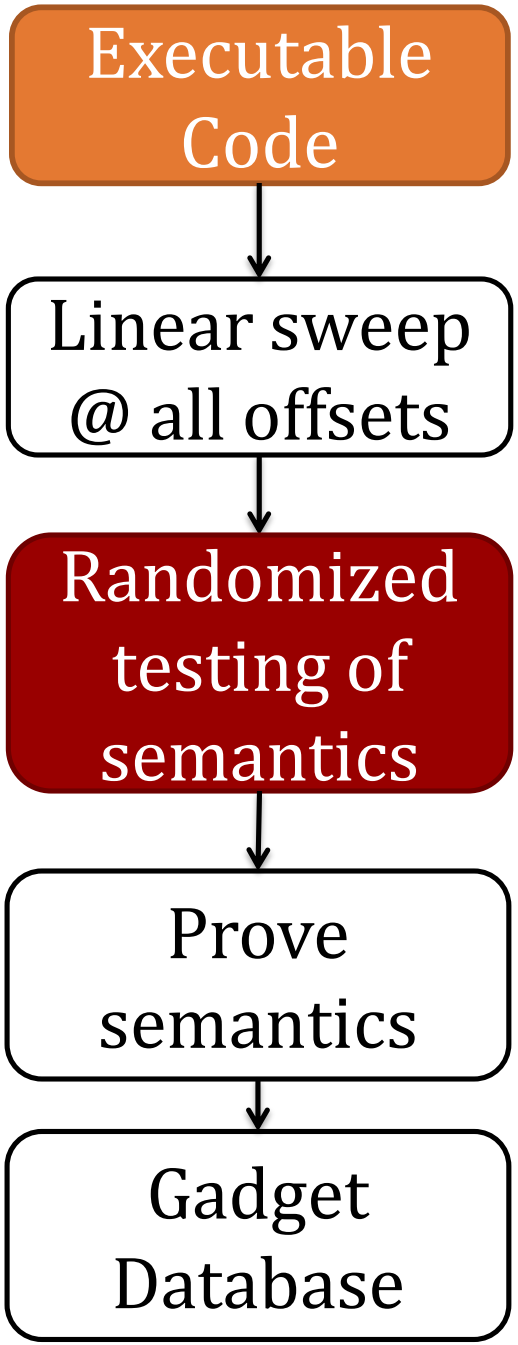
Q-Op	Semantics	Real World Example
MoveRegG(t_1, t_2)	$t_1 := t_2$	xchg %eax, %ebp; ret
LoadConstG(t_1, c)	t_1 := c	pop %ebp; ret
ArithmeticG(t_1, t_2, t_3, *op*)	$t_1 := t_2$ op t_3;	add %edx, %eax; ret
LoadMemG(t_1, t_2, c)	$t_1 := [t_2 + c]$	movl 0x60(%eax), %eax; ret
StoreMemG(t_1,c, t_2)	$[t_1+c] := t_2$	mov %dl, 0x13(%eax); ret
ArithmeticLoadG(t_1, t_2, c, *op*)	$t_1 := t_1$ *op* $[t_2 + c]$	add0x1376dbe4(%ebx), %ecx; (…); ret
ArithmeticStoreG(t_1, t_2, c, *op*)	$[t_1+c] := [t_1+c]$ op t_2	add %al, 0x5de474c0(%ebp); ret

Q-Ops (aka Q Semantic Types) (think instruction set architecture)

Q-Op	Semantics	Real World Example
MoveRe		xch
LoadCo		po
ArithmeticG(t_1, t_2, t_3, *op*)	t_1 := t_2	ad
LoadMemG(t_1, t_2, c)	t_1:= [t_2 + c]	
StoreMemG(t_1,c, t_2)	[t_1+c] := t_2	mo
ArithmeticLoadG(t_1, t_2, c, *op*)	t_1 := t_1 *op* [t_2 + c]	add 1376dbe4(%ebx), %ecx; (…); ret
ArithmeticStoreG(t_1, t_2, c, *op*)	[t_1+c] := [t_1+c] op t_2	add %al, 0x5de474c0(%ebp); ret

Must be careful attackers, e.g., give c-60 to get c

This is not RISC: more Q-Ops gives more opportunities later



- Randomized testing tells us we **likely** found a gadget that implements a Q-Op
  - Fast: filters out many candidates
  - Enables more expensive second stage

Randomized Testing Example

Before Simulation

EAX	0x0298a7bc
CF	0x1
ESP	0x81e4f104

Semantically EAX := CF (MoveRegG)

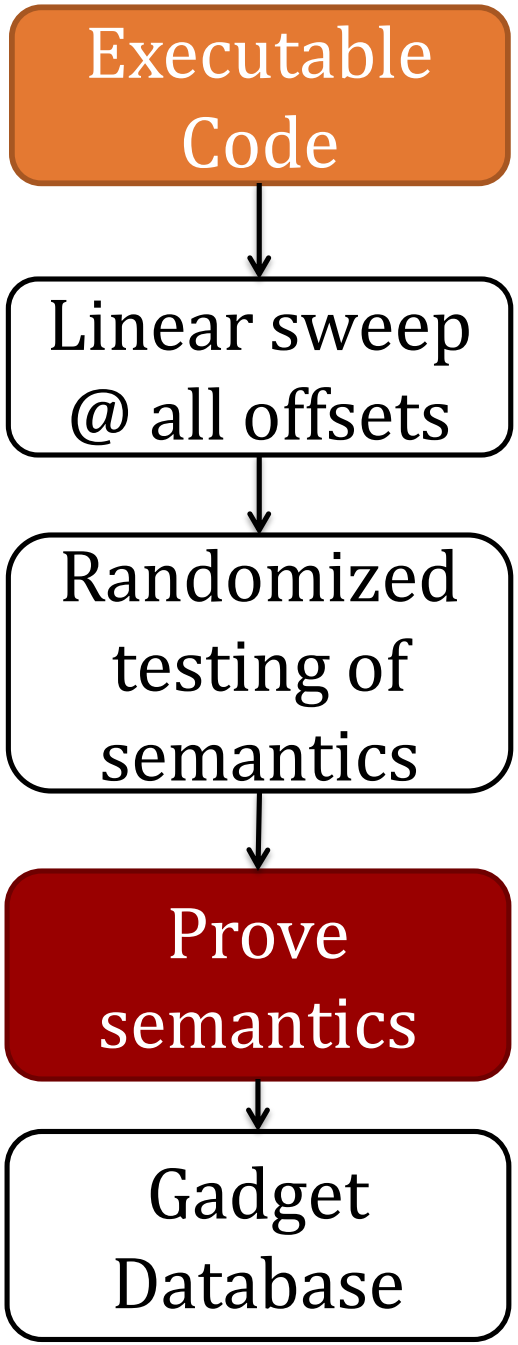
What does this do?

sbb %eax, %eax;
neg %eax; ret

Probably

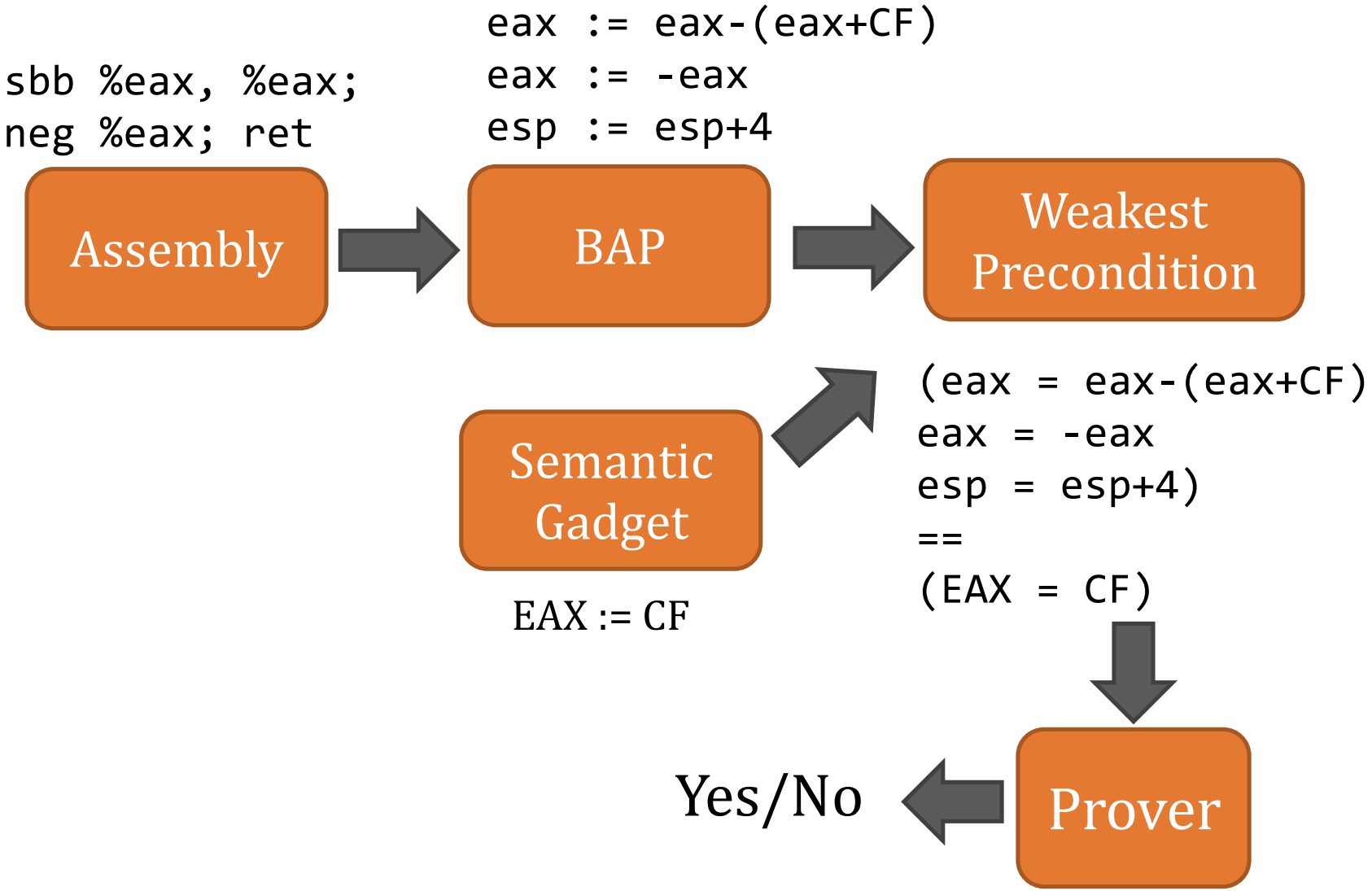
After Simulation

EAX	0x1
ESP	0x81e4f108
CF	0x1



Turn
probably into a proof
that a gadget
implements a Q-Op

Proving equivalence

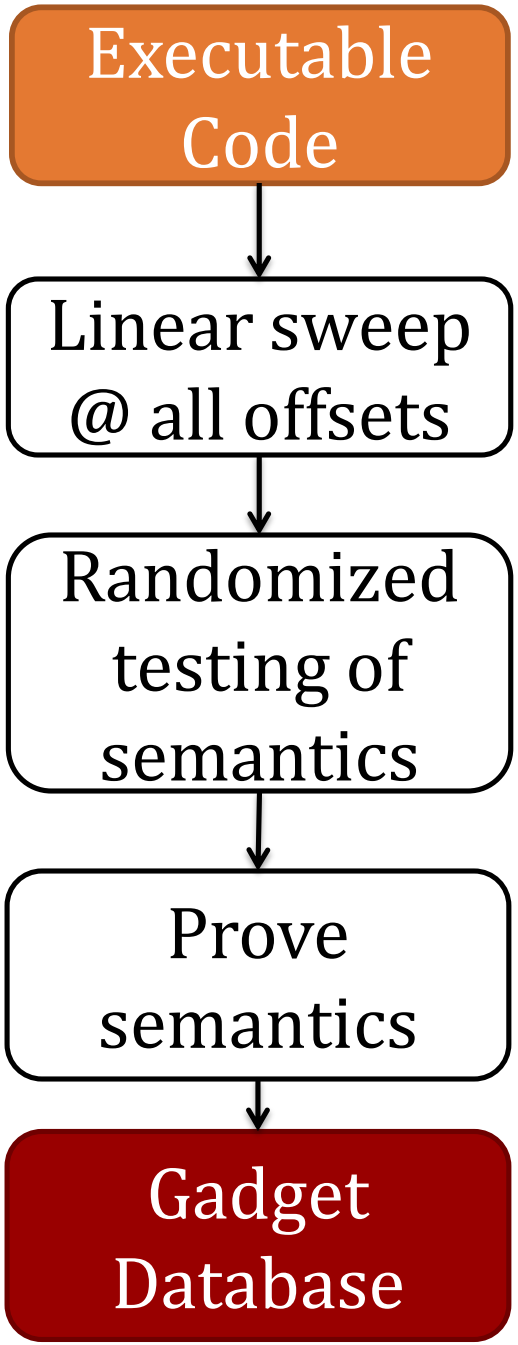


Proving Equivalence

- Weakest precondition [Dijkstra76] is an algorithm for reducing a program to a statement in logic
  - Q uses predicate logic

- Satisfiability Modulo Theories (SMT) solver, a "Decision Procedure", determine if statement is true
  - true ➔ semantic gadget
  - Note: "Theorem prover"=undecidable, "SAT solver" = propositional logic

- WP details not discussed here. (It's a textbook verification technique)



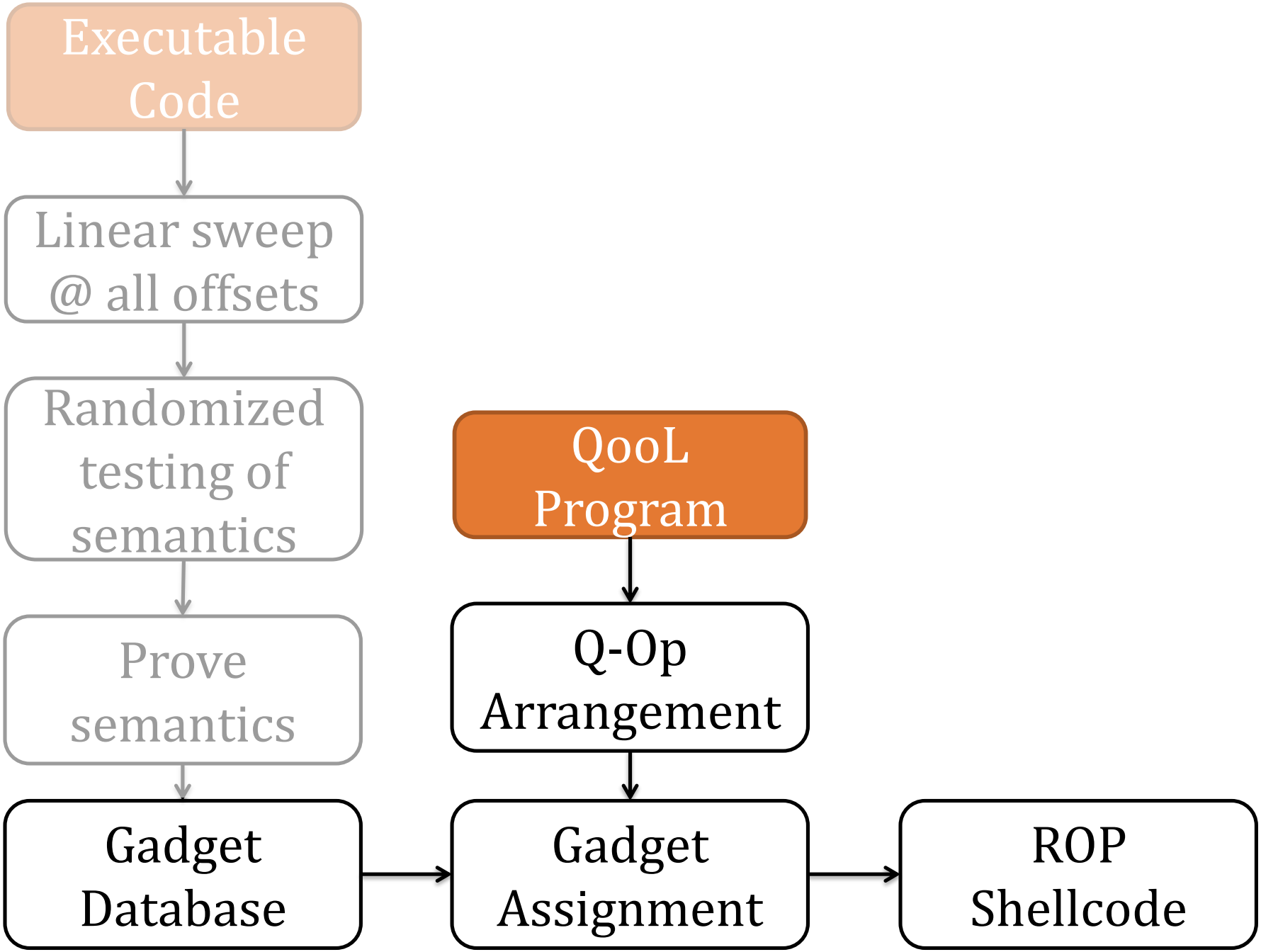
- ✓ Disassemble code
- ✓ Identify *useful* code sequences as gadgets
- Assemble gadgets into desired shellcode

Q-Op Gadget Examples

Q-Op	Gadget
eax := value	pop %ebp; ret; xchg %eax, %ebp; ret
ebx := *value*	pop %ebx; pop %ebp; ret
[ebx +0x5e5b3cc4] := [ebx + *offset*] | al	or %al, 0x5e5b3cc4(%ebx); pop %edi; pop %ebp; ret
eax := *value*	pop %ebp; ret; xchg %eax, %ebp; ret
ebp := *value*	pop %ebp; ret
[ebp + 0xf3774ff] := [ebp + offset] + al	add %al,0xf3774ff(%ebp); movl $0x85, %dh; ret

apt-get Gadgets

Note: extra side-effects handled by Q


Executable Code
Linear sweep @ all offsets
Randomized testing of semantics
Prove semantics
Gadget Database
QooL Program
Q-Op Arrangement
Gadget Assignment
ROP Shellcode

QooL Language

Motivation:
Write shellcode in high-level language,
not assembly

$$
\begin{array}{lll}
\langle \text{exp} \rangle & ::= & \textsc{LoadMem}\ \langle \text{exp} \rangle\ \langle \text{type} \rangle \\
 & & \mid \textsc{BinOp}\ \langle \text{optype} \rangle\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle \\
 & & \mid \textsc{Const}\ \langle \text{exp} \rangle\ \langle \text{type} \rangle \\
\langle \text{stmt} \rangle & ::= & \textsc{StoreMem}\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle\ \langle \text{type} \rangle \\
 & & \mid \textsc{Assign}\ \langle \text{var} \rangle\ \langle \text{exp} \rangle \\
 & & \mid \textsc{CallExternal}\ \langle \text{func} \rangle\ \langle \text{exp list} \rangle \\
 & & \mid \textsc{SysCall}
\end{array}
$$

QooL Syntax

Example

$$
\begin{array}{lll}
\langle \text{exp} \rangle & ::= & \textsc{LoadMem}\ \langle \text{exp} \rangle\ \langle \text{type} \rangle \\
 & & \mid \textsc{BinOp}\ \langle \text{optype} \rangle\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle \\
 & & \mid \textsc{Const}\ \langle \text{exp} \rangle\ \langle \text{type} \rangle \\
\langle \text{stmt} \rangle & ::= & \textsc{StoreMem}\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle\ \langle \text{type} \rangle \\
 & & \mid \textsc{Assign}\ \langle \text{var} \rangle\ \langle \text{exp} \rangle \\
 & & \mid \textsc{CallExternal}\ \langle \text{func} \rangle\ \langle \text{exp list} \rangle \\
 & & \mid \textsc{SysCall}
\end{array}
$$

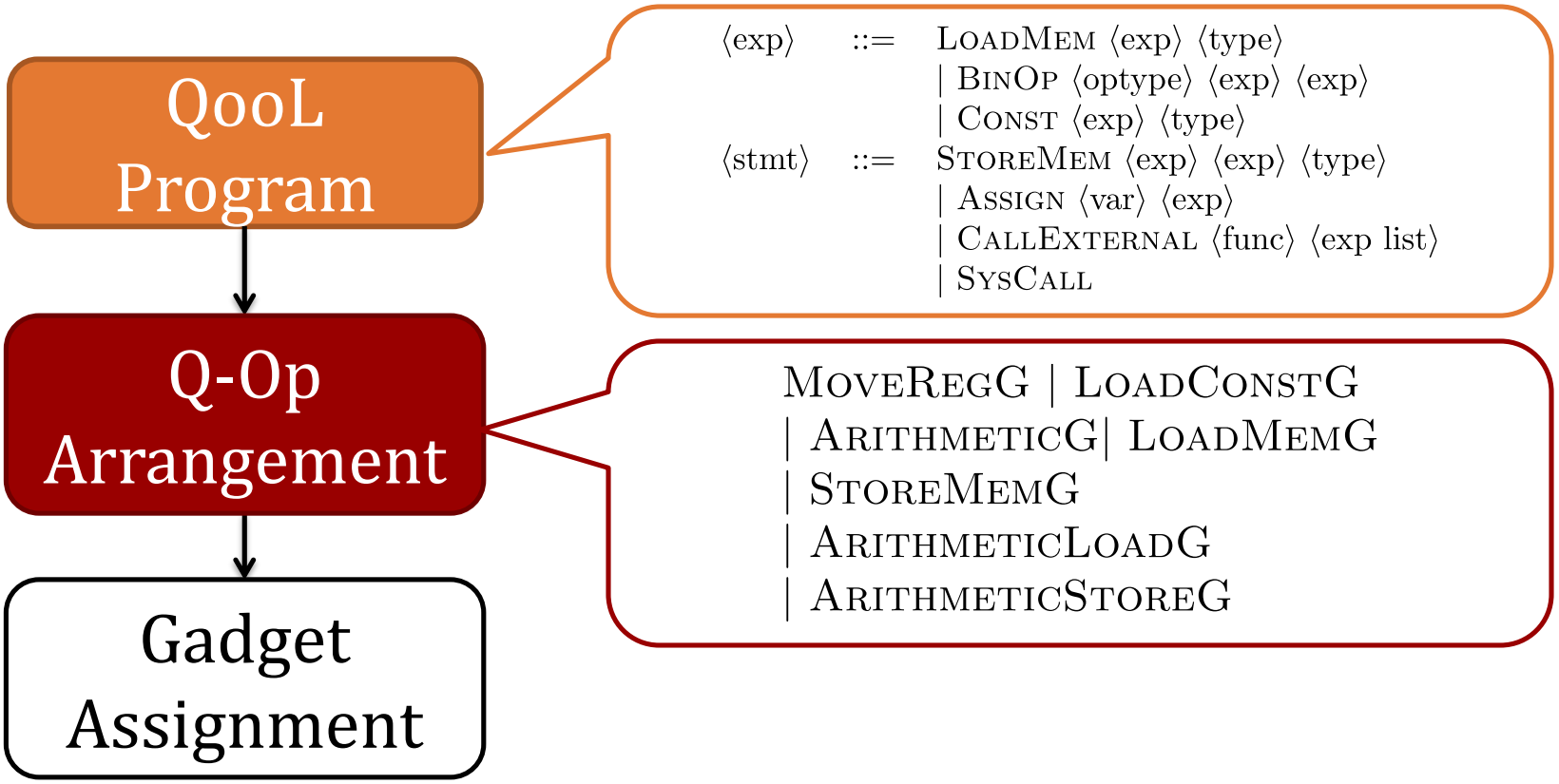
f = [got offset execve]
f("/bin/sh")

Semantics

f = LoadMem[got execve offset]
arg = "/bin/sh" (in hex)
StoreMem(t, adrr)
f(addr)

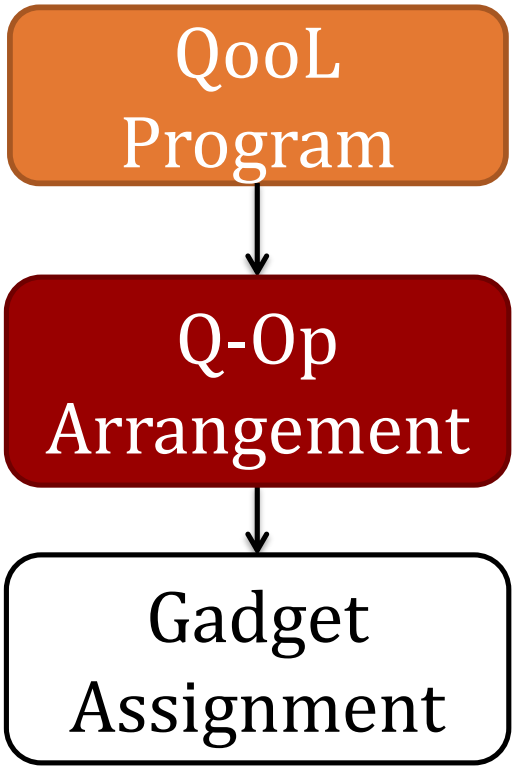
QooL Program

Q-Op Arrangement



Analogy:
Compiling C down to assembly

Every-Munch Algorithm



- Every Munch: Conceptually compile QooL program into a *set* of Q-Op programs
  - Each member tries to use different Q-Ops for the same high-level instructions
- Analogy: Compile C statement to a set of assembly instructions
  - C: a = a*2;
  - Assembly: a = a *2;
    a = a << 1;
    a = a + a;

StoreMem[a] = v

QooL

- Ultimately pick the smallest Q-Op program that has corresponding gadgets in the target program
- Optimization: Q uses lazy evaluation so programs generated on demand

[a] := v

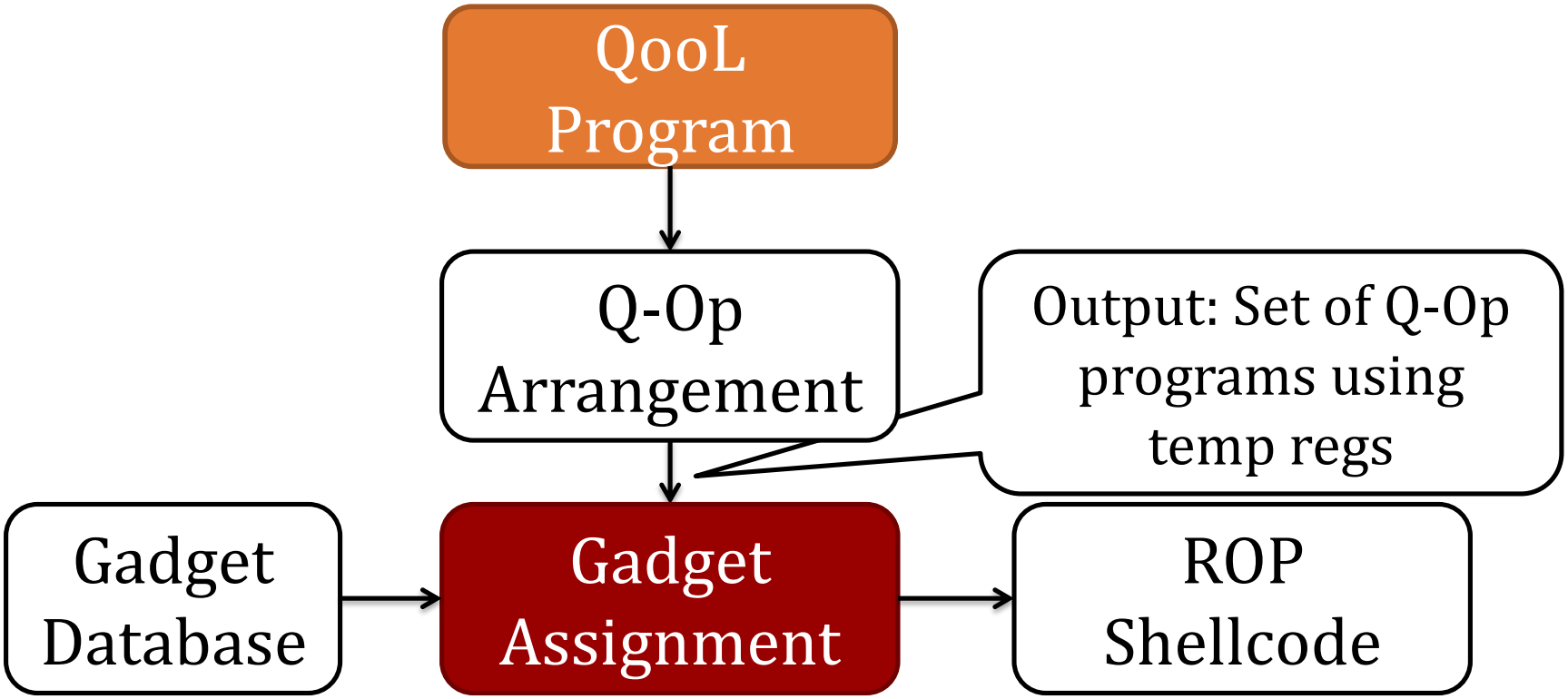
t_1 := a;
t_2 := v;
[t_1] = t_2;

t_1 := a;
t_2 := -1;
[t_1] := [t_1] | t_2;
t3 := v + 1;
[t_1] := [t_1] + t_3;

...

Q-Op Programs

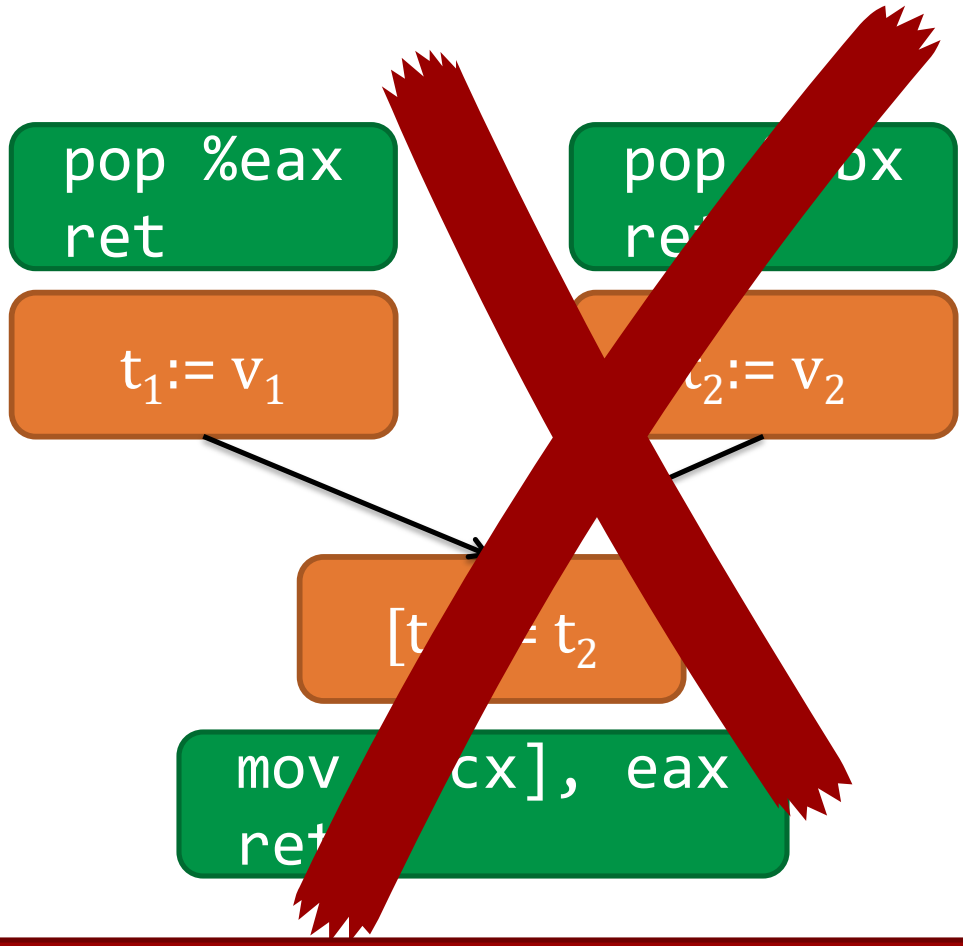
Gadget Assignment



Assignment chooses a single Q-Op program using real gadgets and register names

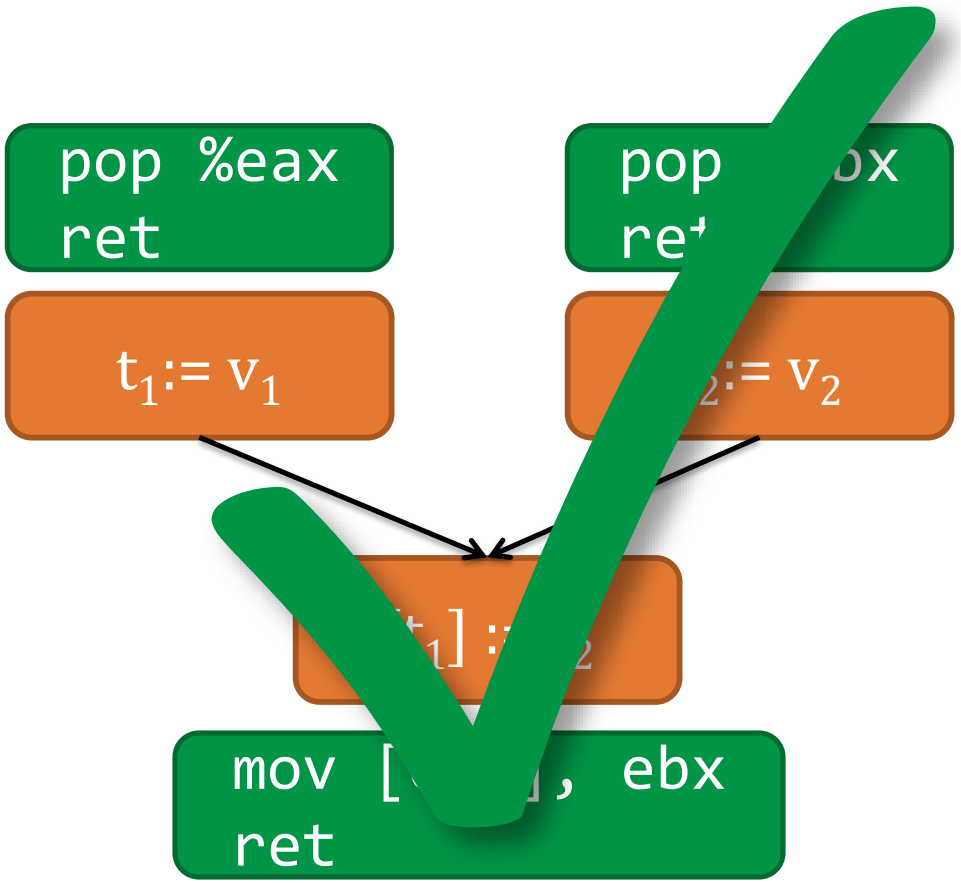
Analogy: Register assignment in a compiler

Example



Conflict: %ebx and %ecx mismatch

Example

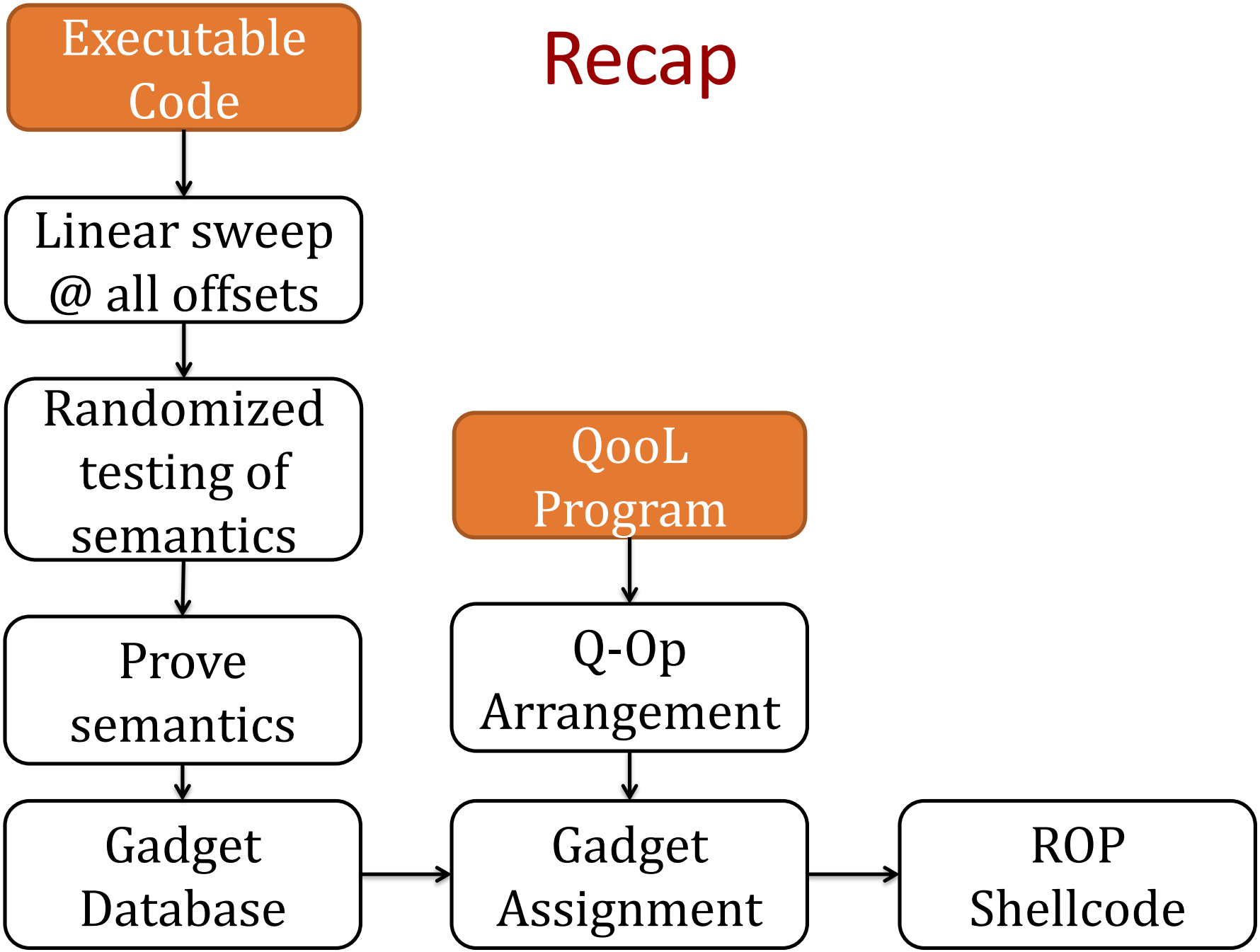


Legend

Q-Op

Assembly Gadget

Recap


Executable Code
Linear sweep @ all offsets
Randomized testing of semantics
Prove semantics
Gadget Database
QooL Program
Q-Op Arrangement
Gadget Assignment
ROP Shellcode

Real Exploits

Q ROP'ed (and hardened) 9 exploits

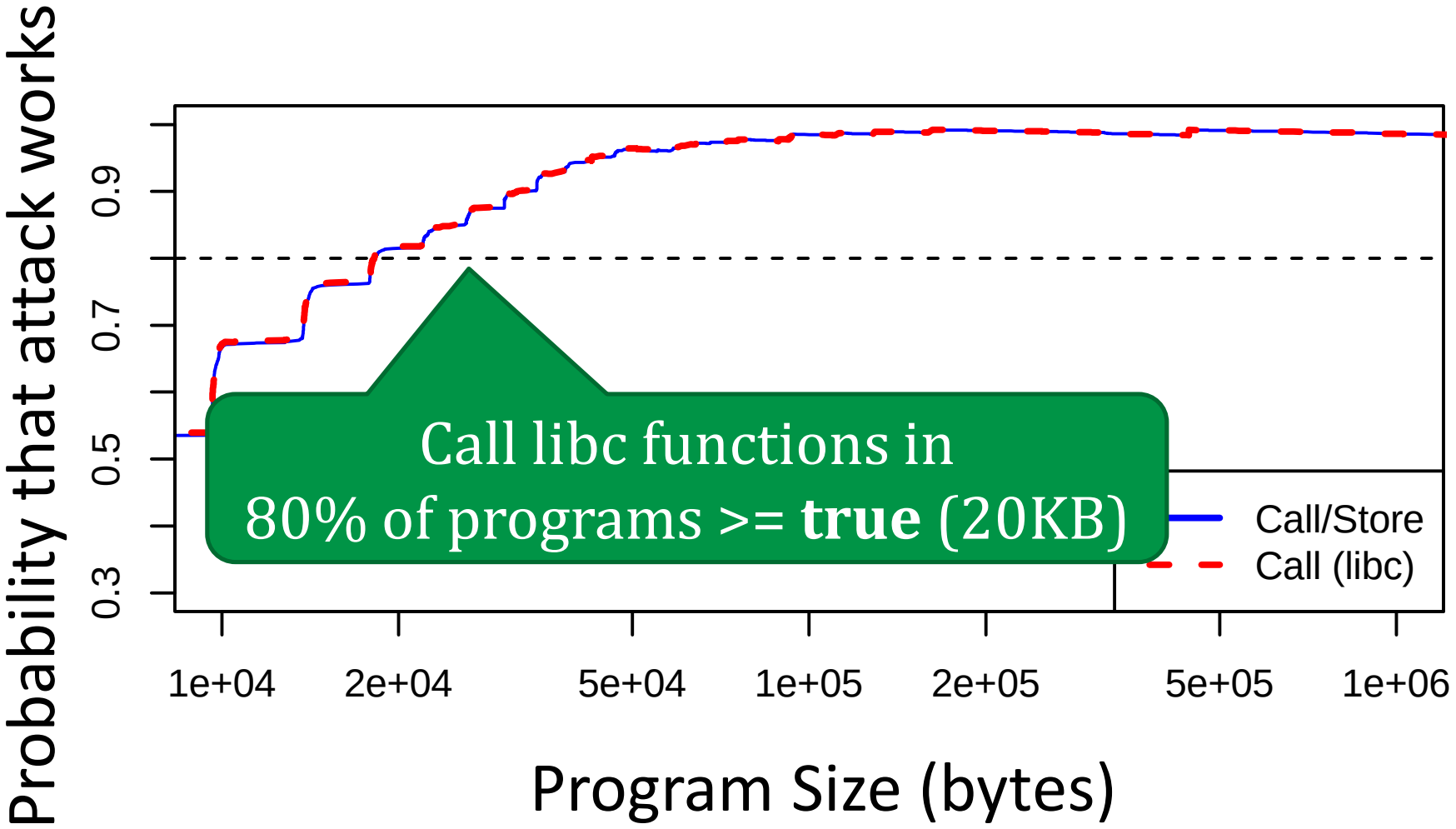
Name	Total Time	OS
Free CD to MP3 Converter	130s	Windows 7
Fatplayer	133s	Windows 7
A-PDF Converter	378s	Windows 7
A-PDF Converter (SEH exploit)	357s	Windows 7
MP3 CD Converter Pro	158s	Windows 7
rsync	65s	Linux
opendchub	225s	Linux
gv	237s	Linux
Proftpd	44s	Linux

ROP Probability

- Given program size, what is the probability Q can create a payload?
  - Measure over all programs in /usr/bin

- Depends on target computation
  - Call libc function in GOT
  - Call libc function not in GOT

ROP Probability



Q ROP Limitations

- Q's gadgets types are not Turing-complete
  - Calling system("/bin/sh") or mprotect() usually enough
  - Shacham showed libc has a Turing-complete set of gadgets.
- Q does not find conditional gadgets
  - Potential automation of interesting work on ROP without Returns **[CDSSW10]**
- Q does not minimize ROP payload size

Research Summary

Shachem: Automatic

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

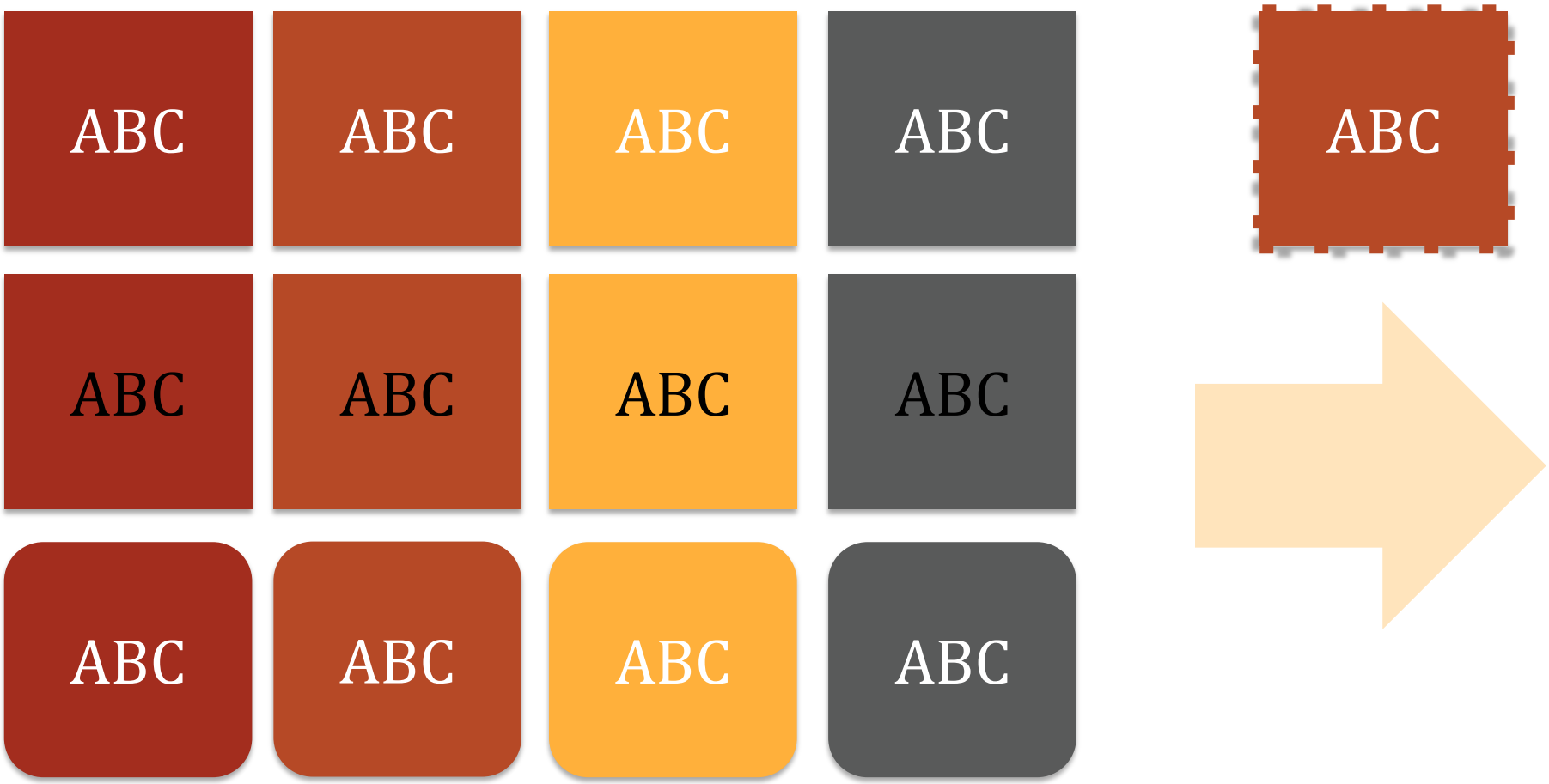
Shacham: Manual, Turing-complete

Q: Automatic, not Turing complete

Backup slides here.

- Titled cherries because they are for the pickin. (credit due to maverick for wit)

Stencils



Other Colors from Adobe Kuler

Don't use these unless absolutely necessary.
We are not making skittles, so there is no rainbow of colors necessary.

Mac application for Adobe Kuler:
http://www.lithoglyph.com/mondrianum/
http://kuler.adobe.com/

What if we don't know the address of "`/bin/sh`"?

Use existing application logic that does!

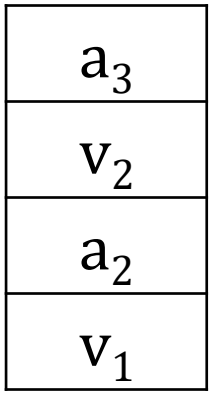
Scorecard for ret2libc

- No injected code ➔ DEP ineffective
- Requires setting up the stack frame
- ... or does it.

Gadgets, Historically

```
Mem[v2] = v1
```

Semantics




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

Gadgets

- Shacham et al. manually identified which sequences ending in ret in libc were useful gadgets
- Common shellcode was created with these gadgets.
- Everyone used libc, so gadgets and shellcode universal