Software-level Attacks on Architectural and Microarchitectural State

ACACES SUMMER SCHOOL

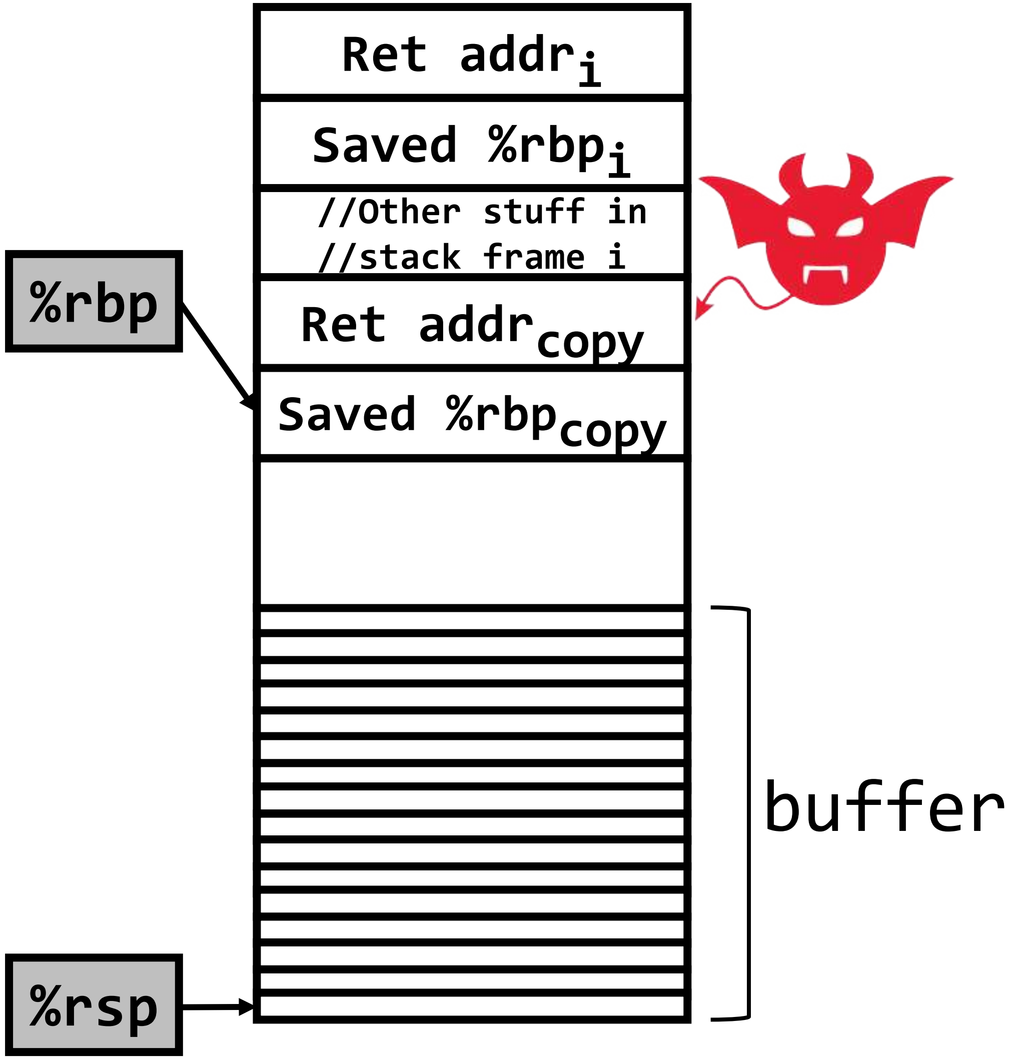
July 9th - 10th, 2020

~~Fiuggi, Rome~~ Zoom World

James Mickens

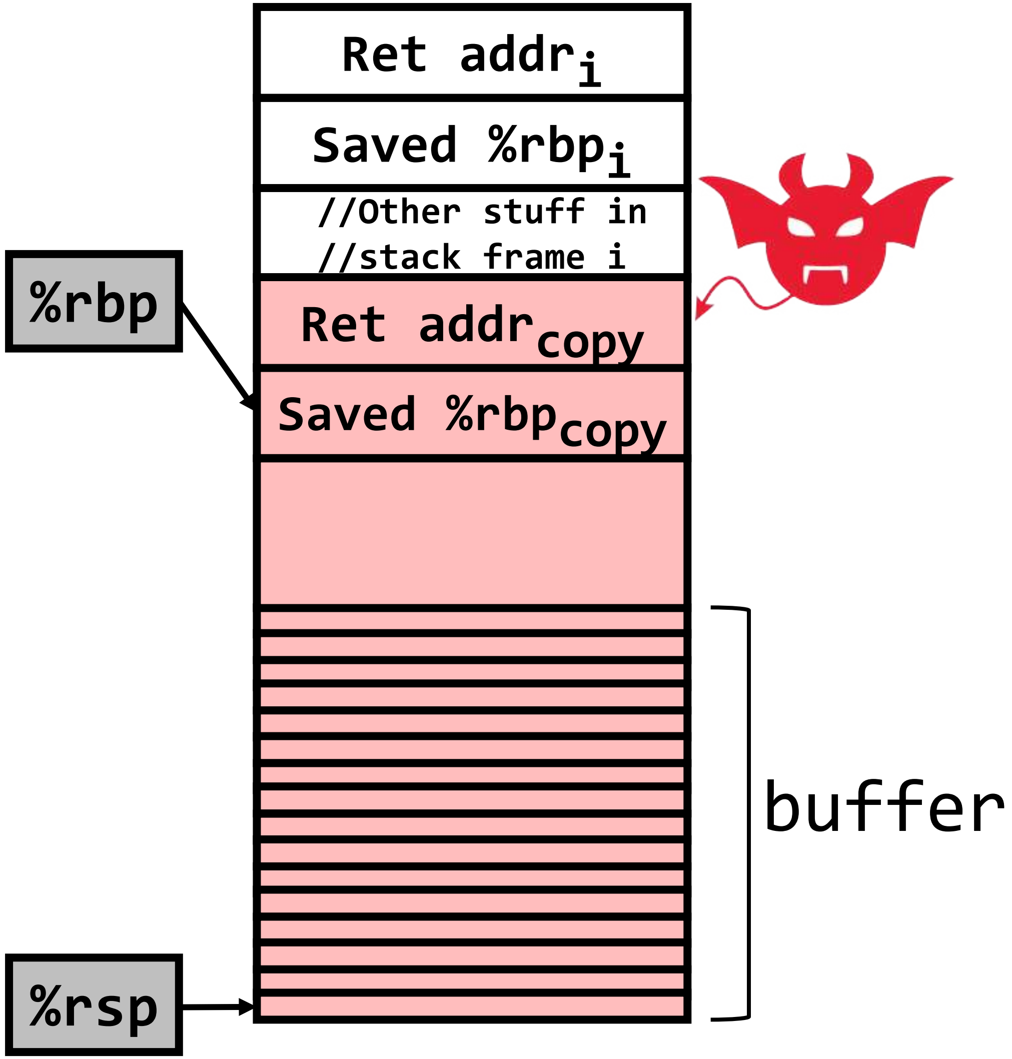
Harvard University

Summary of the Last Lecture



- Buffer overflows let attackers overwrite code pointers in the path of the overflow
- Subverting control flow allows:
  - Return to attacker-provided shellcode
  - Return to **`libc`** (or another preexisting function)

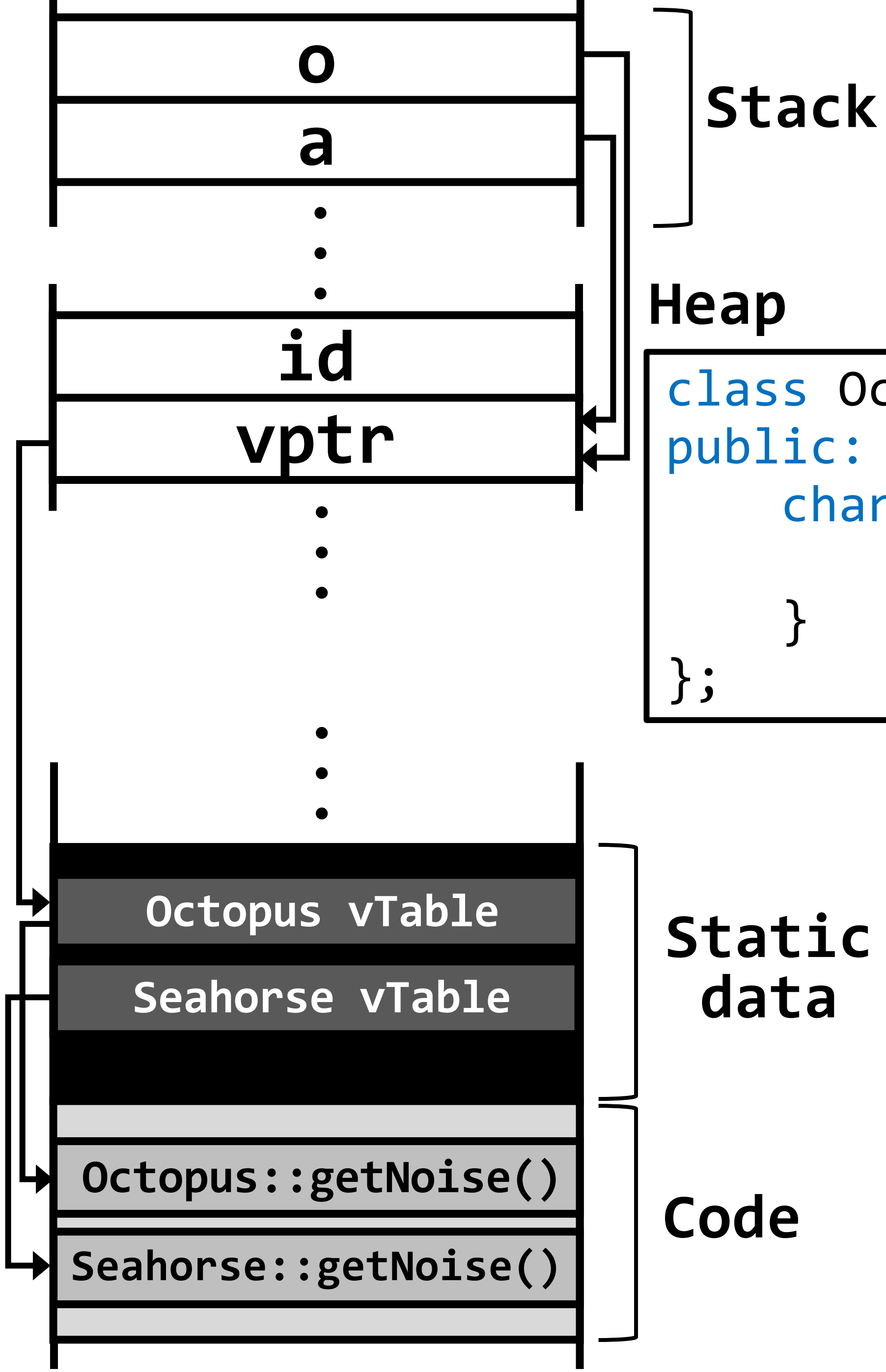
Summary of the Last Lecture



- Defenses:
  - Canaries: Detect corruption of return addresses
  - NX bits: Prevent injection of shellcode
  - ASLR + struct randomization: Prevent attacker from easily locating program state
- Given these defenses, attackers require mechanisms to:
  - Read memory (to locate program state)
  - Write memory in a non-contiguous fashion (to avoid corrupting canaries)
  - Implement arbitrary malicious behavior using only preexisting code (to escape the shackles of NX bits)

The Magic of Vptrs

- Modern languages like C++, Go, Java, C#, and Rust allow the binding of an abstract method declaration to multiple concrete implementations
  - C++ polymorphism
  - Interfaces in Go, Java, and C#
- Bindings are enabled through a layer of indirection
- By corrupting the indirection tables, the attacker can create memory exploits (both read and write)!

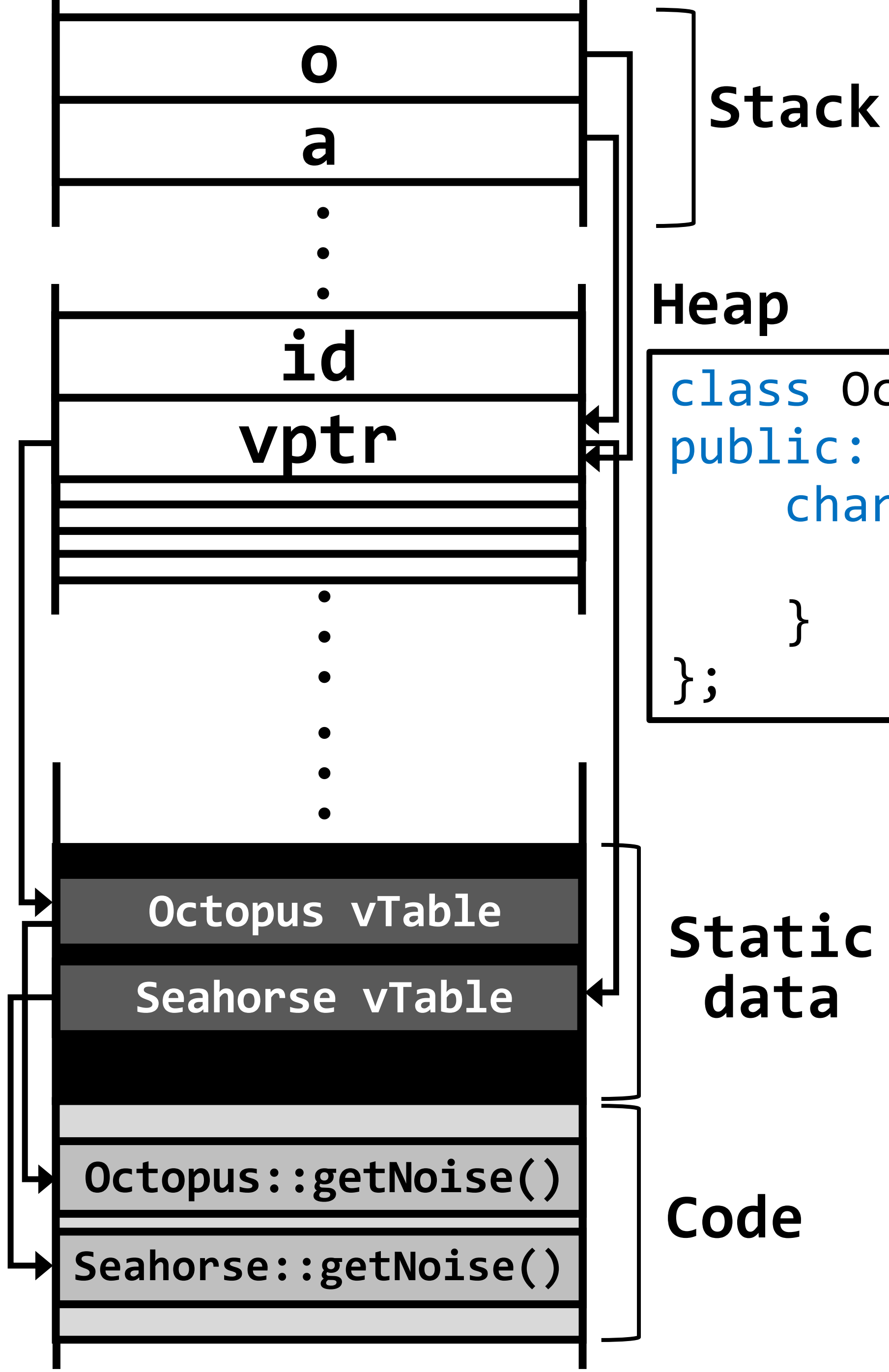


```
class Animal{
public:
    int id;
    virtual char *getNoise()=0;
};
```

```
class Octopus: public Animal{
public:
    char *getNoise(){
        return “blub”;
    }
};
```

```
class Seahorse: public Animal{
public:
    char *getNoise(){
        return “wtf”;
    }
};
```

```
void f(){
    Octopus *o = new Octopus();
    Animal *a = o;
    a->getNoise(); //Implicitly
    //becomes a->vptr->getNoise()
}
```

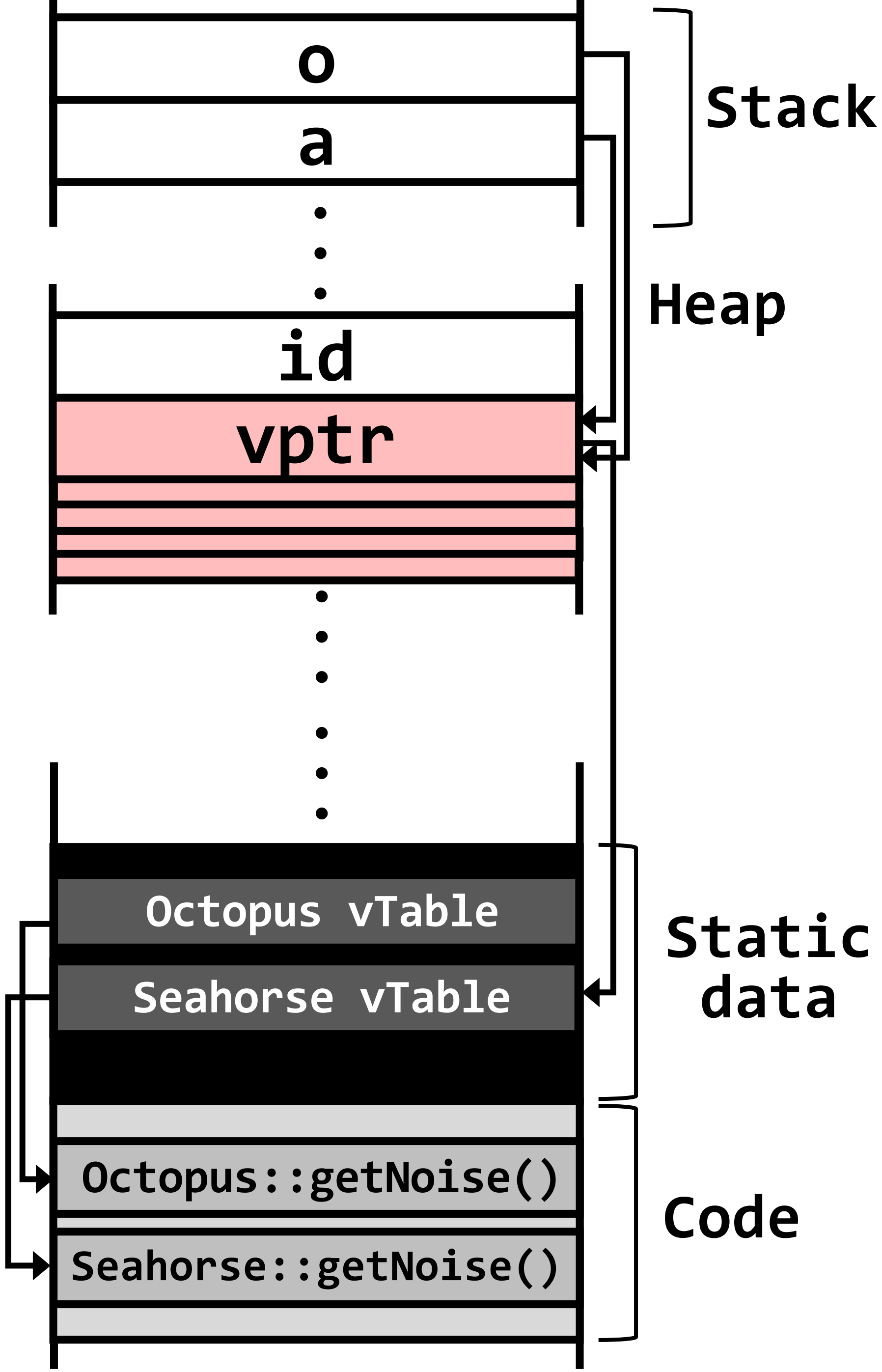


```
class Animal{
public:
    int id;
    virtual char *getNoise()=0;
};
```

```
class Octopus: public Animal{
public:
    char *getNoise(){
        return “blub”;
    }
};
```

```
class Seahorse: public Animal{
public:
    char *getNoise(){
        return “wtf”;
    }
};
```

```
void f(){
    Octopus *o = new Octopus();
    Animal *a = o;
    a->getNoise(); //Implicitly
    //becomes a->vptr->getNoise()
}
```

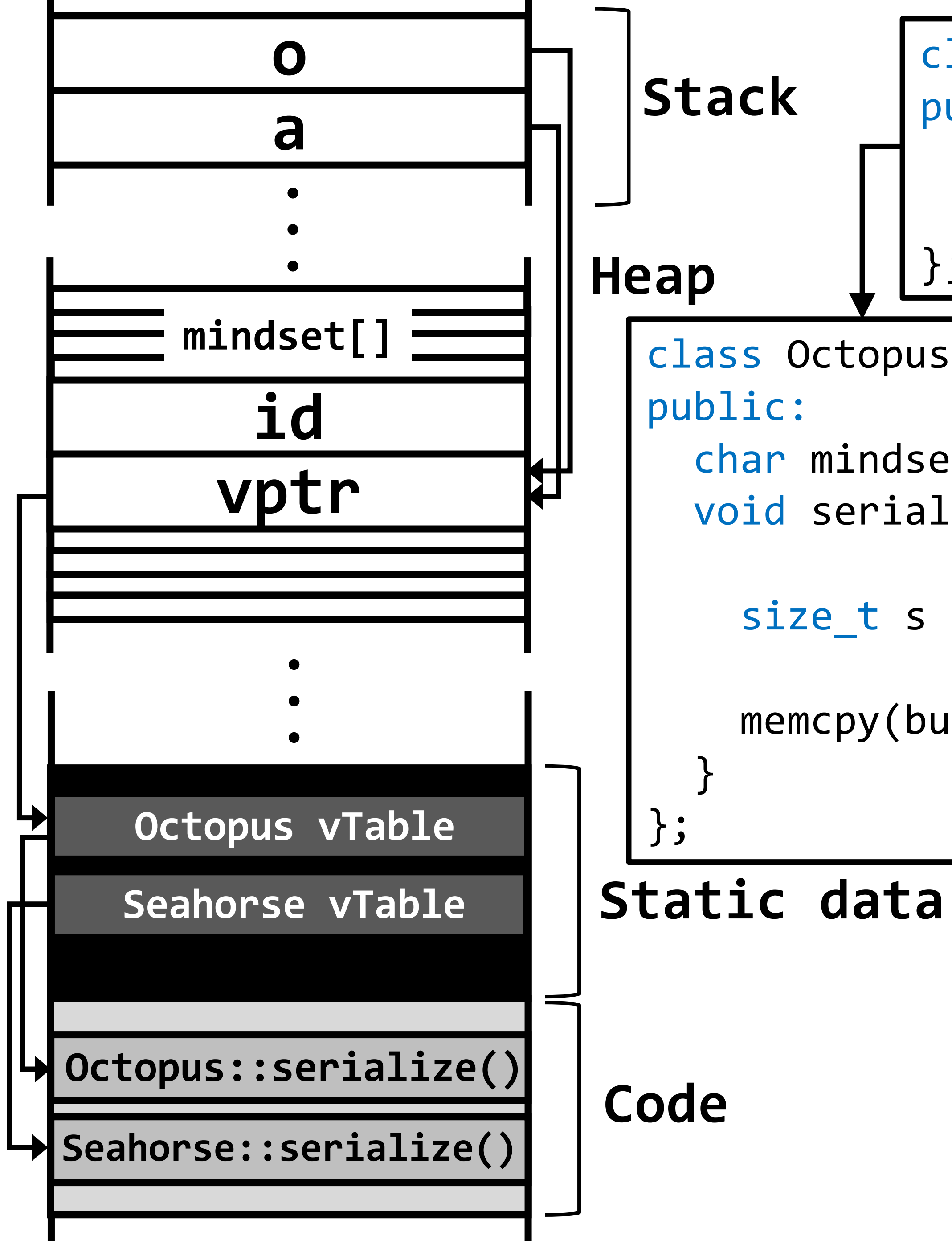


TYPE CONFUSION

```
void f(){
   Octopus *o = new Octopus();
   Animal *a = o;
   a->getNoise(); //Implicitly
   //becomes a->vptr->getNoise()
}
```

The Octopus object invokes the Seahorse method!

TYPE CONFUSION IS VERY, VERY BAD

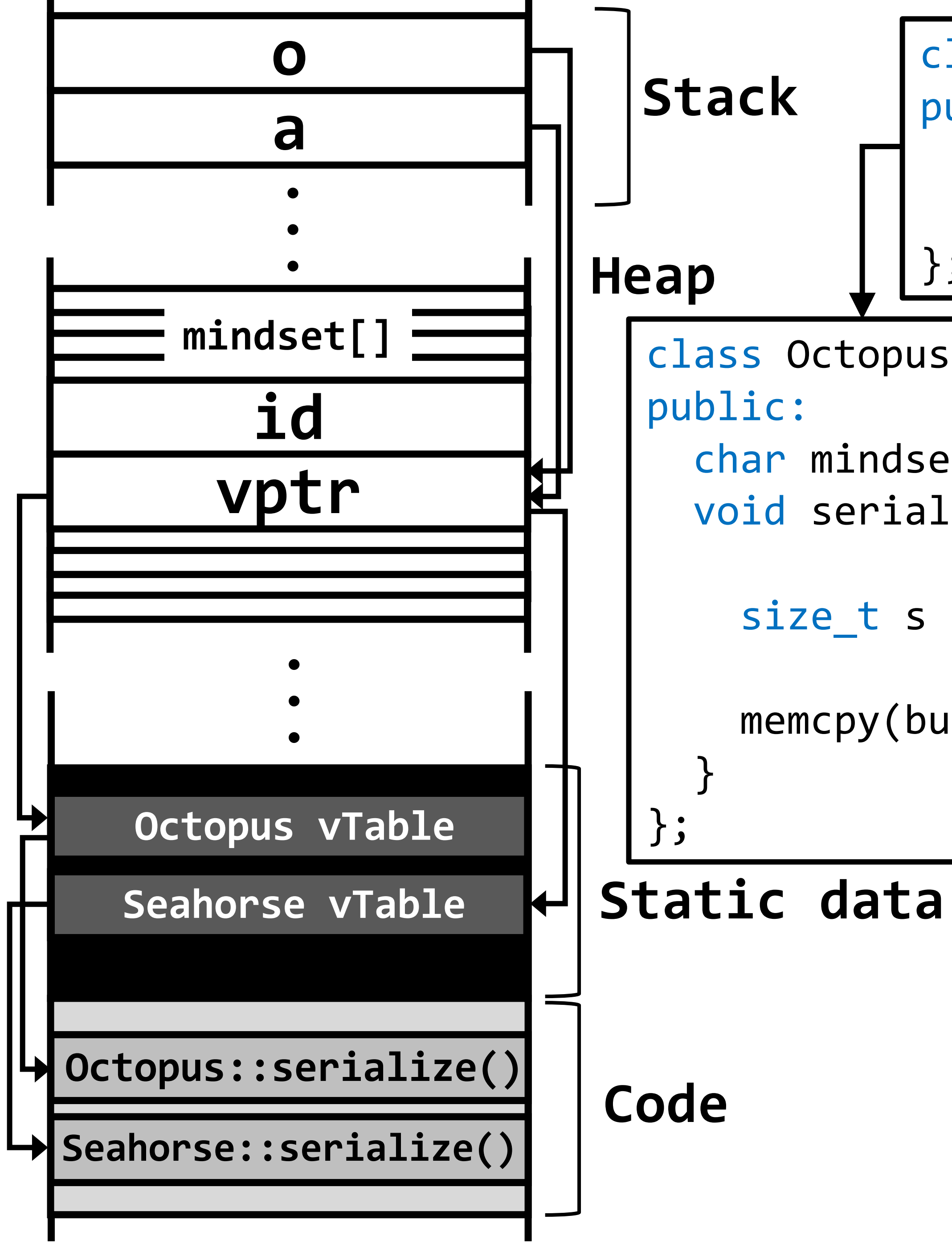


```
class Animal{
public:
    int id;
    virtual void serialize(char *buf, size_t len)=0;
};
```

```
class Octopus: public Animal{
public:
  char mindset[4] = “evil”;
  void serialize(char *buf,
                 size_t len){
    size_t s = min(len,
               sizeof(Octopus));
    memcpy(buf, this, s);
  }
};
```

```
class Seahorse: public Animal{
public:
  char facts[128] = “The movie
      ‘Aquaman’ is actually good
      although the DC franchise
      is otherwise terrible.”;
  void serialize(char *buf,
                 size_t len){
    size_t s = min(len,
               sizeof(Seahorse));
    memcpy(buf, this, s);
  }
};
```

```
void f(){
    char buf[1024];
    Octopus *o = new Octopus();
    Animal *a = o;
    a->serialize(buf, sizeof(buf));
}
```

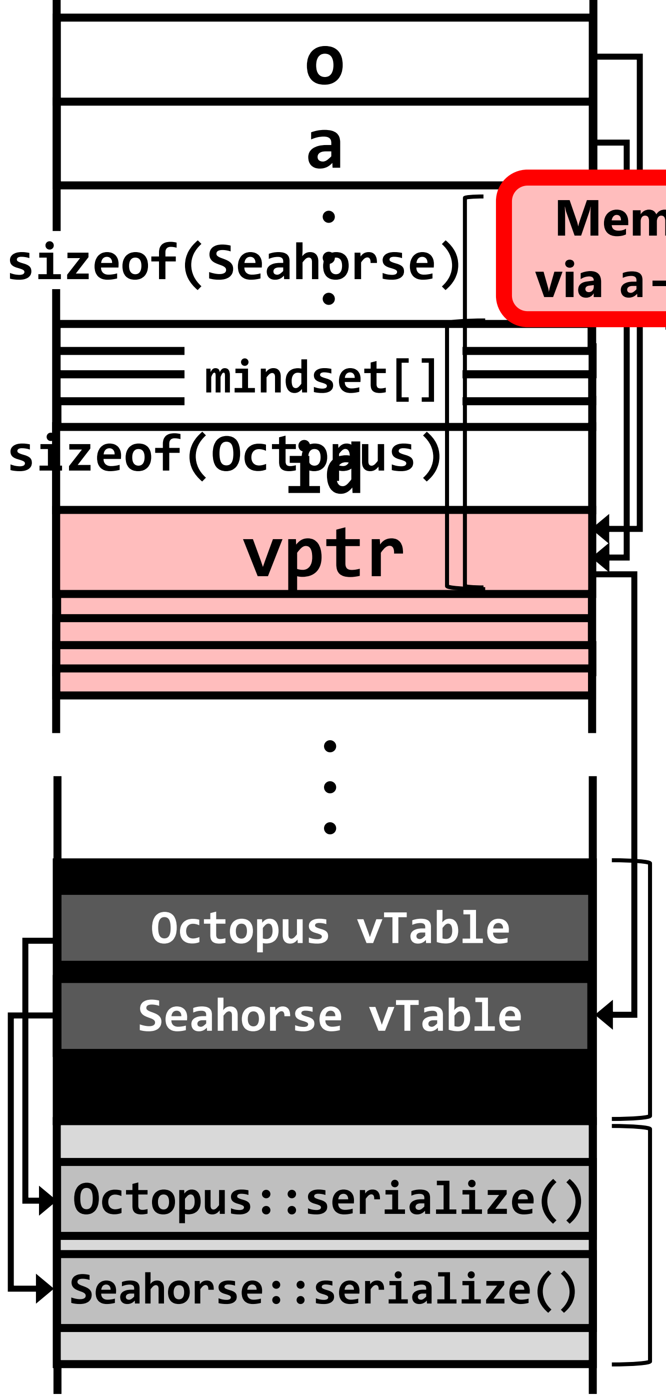


```
class Animal{
public:
    int id;
    virtual void serialize(char *buf, size_t len)=0;
};
```

```
class Octopus: public Animal{
public:
  char mindset[4] = “evil”;
  void serialize(char *buf,
                 size_t len){
    size_t s = min(len,
                   sizeof(Octopus));
    memcpy(buf, this, s);
  }
};
```

```
class Seahorse: public Animal{
public:
  char facts[128] = “The movie
     ‘Aquaman’ is actually good
     although the DC franchise
     is otherwise terrible.”;
  void serialize(char *buf,
                 size_t len){
    size_t s = min(len,
                   sizeof(Seahorse));
    memcpy(buf, this, s);
  }
};
```

```
void f(){
    char buf[1024];
    Octopus *o = new Octopus();
    Animal *a = o;
    a->serialize(buf, sizeof(buf));
}
```



Memory disclosure via a->serialize()!

```
class Animal{
public:
    int id;
    virtual void serialize(char *buf, size_t len)=0;
};
```

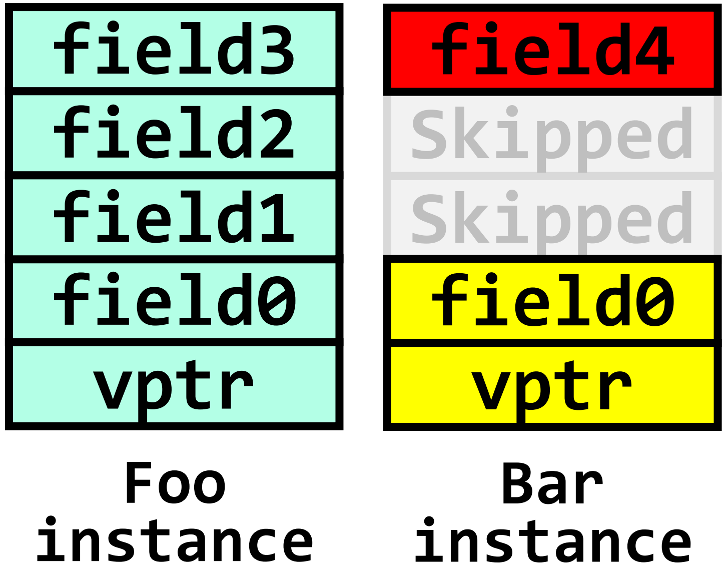
```
class Octopus: public Animal{
public:
  char mindset[4] = “evil”;
  void serialize(char *buf,
                 size_t len){
    size_t s = min(len,
                   sizeof(Octopus));
    memcpy(buf, this, s);
  }
};
```

```
class Seahorse: public Animal{
public:
  char facts[128] = “The movie
      ‘Aquaman’ is actually good
      although the DC franchise
      is otherwise terrible.”;
  void serialize(char *buf,
                 size_t len){
    size_t s = min(len,
                   sizeof(Seahorse));
    memcpy(buf, this, s);
  }
};
```

```
void f(){
    char buf[1024];
    Octopus *o = new Octopus();
    Animal *a = o;
    a->serialize(buf, sizeof(buf));
}
```

Type Confusions and Memory Read/Write Primitives

- Attackers often use type confusions to generate out-of-bounds memory accesses
    - The previous slide demonstrated a memory read vulnerability
    - Type confusion can also generate memory write vulnerabilities (e.g., assigning to an object field via a type-confused **`this`** pointer)
- Type confusions are very helpful if they enable non-contiguous read and write vulnerabilities
    - Helps writes avoid canaries
    - Reduces amount of unnecessary data read

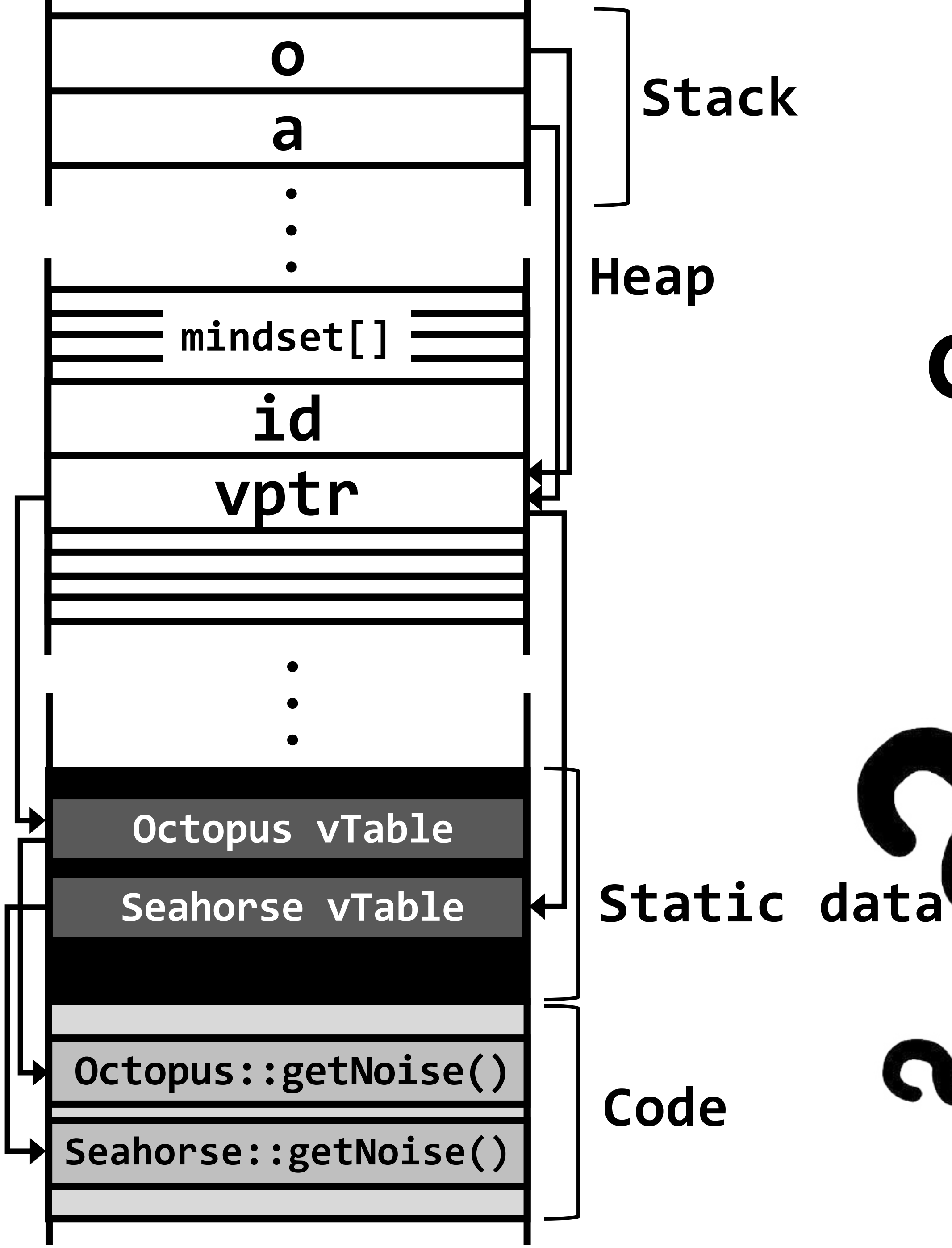


```
void Foo::set_f3(int v){
   this->field3 = v;
}

Bar *b = new Bar();
Foo *f = typeConfuse(b);
   //Exploit: f actually
   //points to a Bar!
f->set_f3(42); //This is
      //a non-contiguous
      //write vuln!
```

Type Confusions and Memory Read/Write Primitives

- Type confusions can be triggered in many ways, not just via buffer overflows!
- Ex: CVE-2015-0336
  - Malicious Flash code uses a carefully-crafted `__proto__` inheritance chain to trick a browser into interpreting a `Vector<uint>` as a `NetConnection`
  - By calling a `NetConnection` method on a `Vector<uint>`, the attacker overwrites the vector's `.length` field to be a huge value, granting read/write access to a huge swath of memory!

Overflowing a heap object

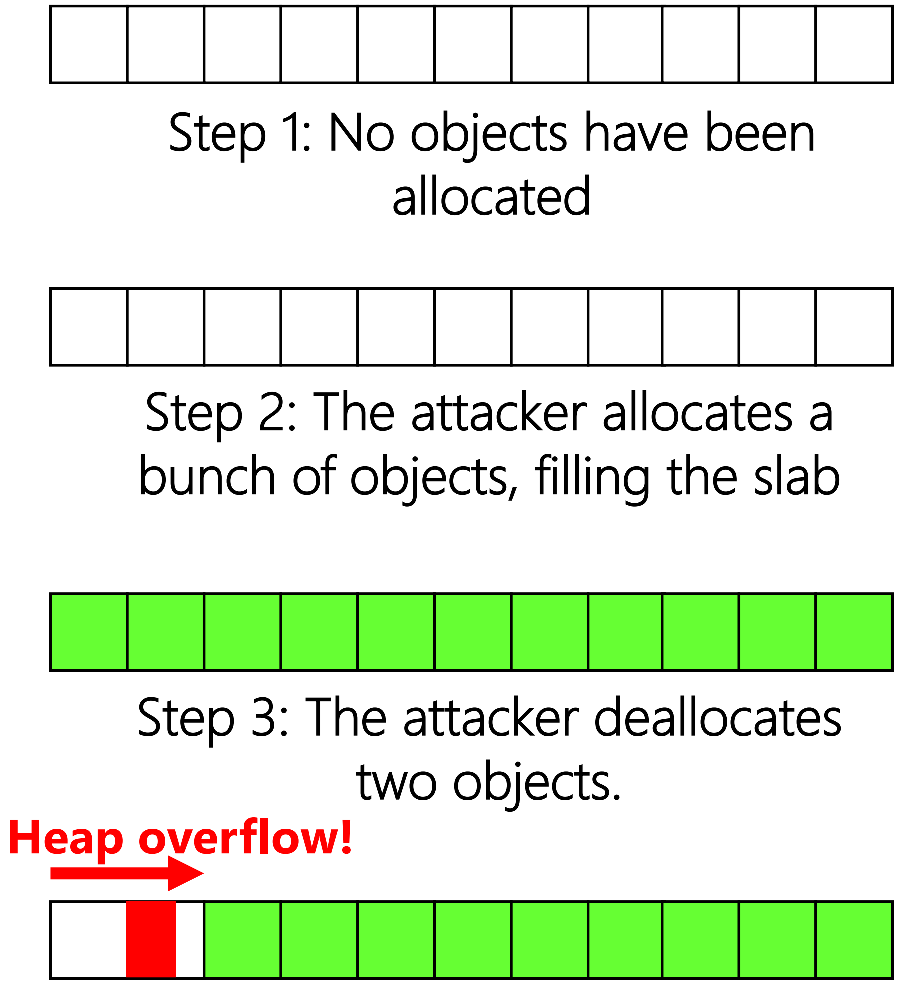


How does the attacker position the overflowable buffer beneath the vulnerable object?

Heap Feng Shui

- Suppose the attacker knows details about the heap allocation algorithm
  - Maybe it's deterministic?
  - Maybe it uses slab allocation?
  - Maybe it allocates large objects on page-aligned offsets, e.g., because **`mmap()`** is used to allocate the memory?
- Attacker can leverage this knowledge to place a vulnerable object next to an overflowable one!

Ex: A deterministic slab allocator



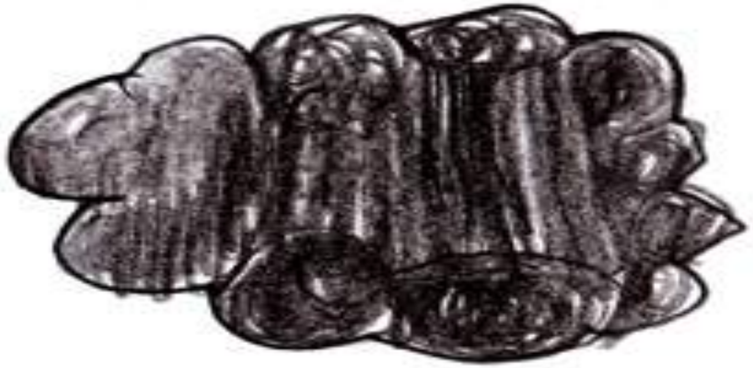
Step 1: No objects have been allocated

Step 2: The attacker allocates a bunch of objects, filling the slab

Step 3: The attacker deallocates two objects.

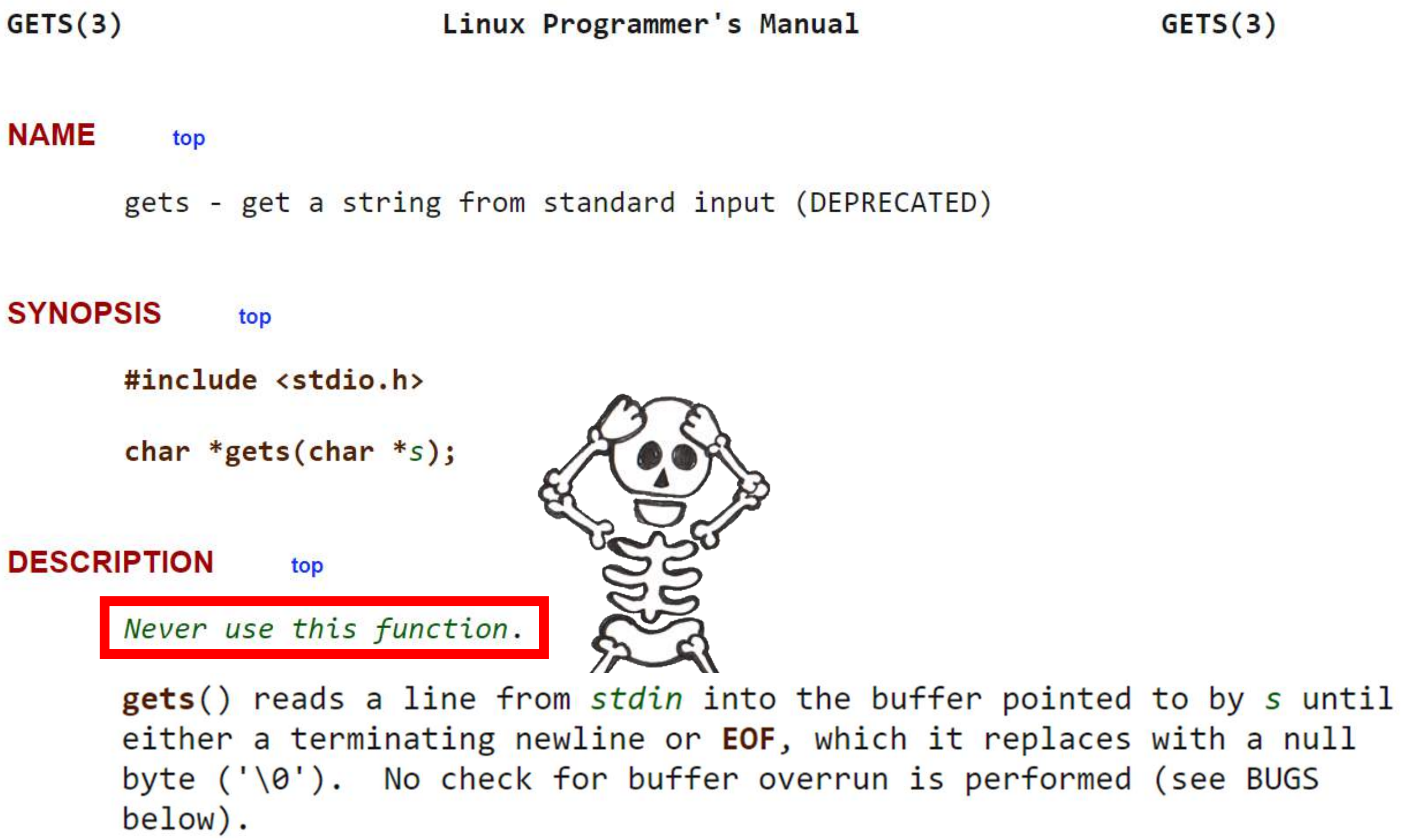
Step 4: The attacker allocates the overflowable object, then the vulnerable one!

Return-oriented Programming



NX prevents an attacker from injecting new code that does evil . . .

. . .but what if the attacker can create evil by chaining together snippets of preexisting code?



GETS(3) Linux Programmer's Manual GETS(3)

NAME top

gets - get a string from standard input (DEPRECATED)

SYNOPSIS top

```
#include <stdio.h>

char *gets(char *s);
```

DESCRIPTION top

Never use this function.

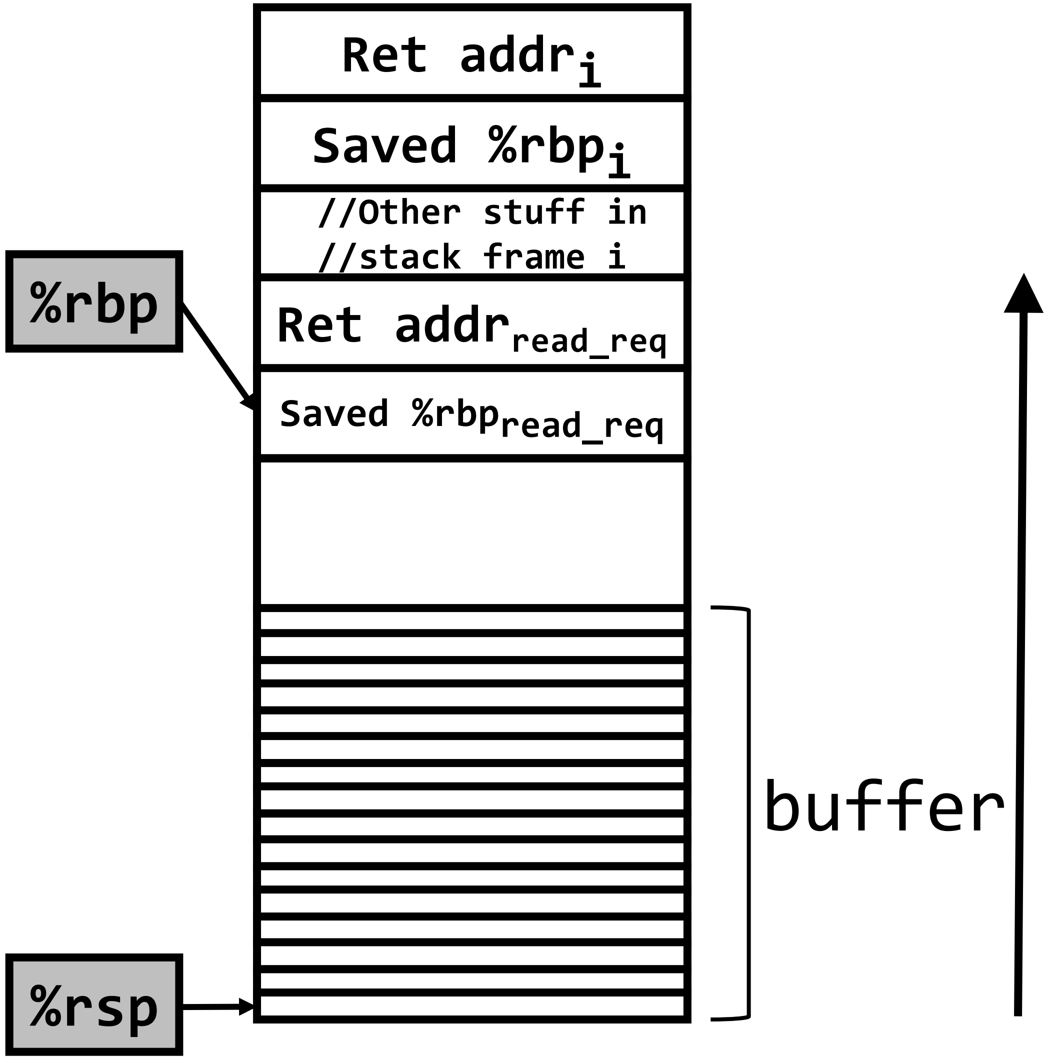
gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

Suppose that a machine uses NX, but does not use stack canaries or ASLR, and contains this code . . .

```
void read_req(){
    char buffer[128];
    gets(buf);
}

void run_shell(){
    system(“/bin/sh”);
     //Spawn a new process with fork(), then do
     //  execl("/bin/sh", "sh", "-c", command, NULL)
     //(where the command in this example is “/bin/sh”), and wait for the
     //command to complete; the spawned processes will use the parent’s
     //stdin/stdout/stdin due to fork()+exec() semantics.
}
```

Suppose that a machine uses NX, but does not use stack canaries or ASLR, and contains this code . . .

```
void read_req(){
    char buffer[128];
    gets(buf);
}

void run_shell(){
    system(“/bin/sh”);
}
```



Ret addr$_{i}$

Saved %rbp$_{i}$

//Other stuff in
//stack frame i

Ret addr$_{read\_req}$

Saved %rbp$_{read\_req}$

buffer

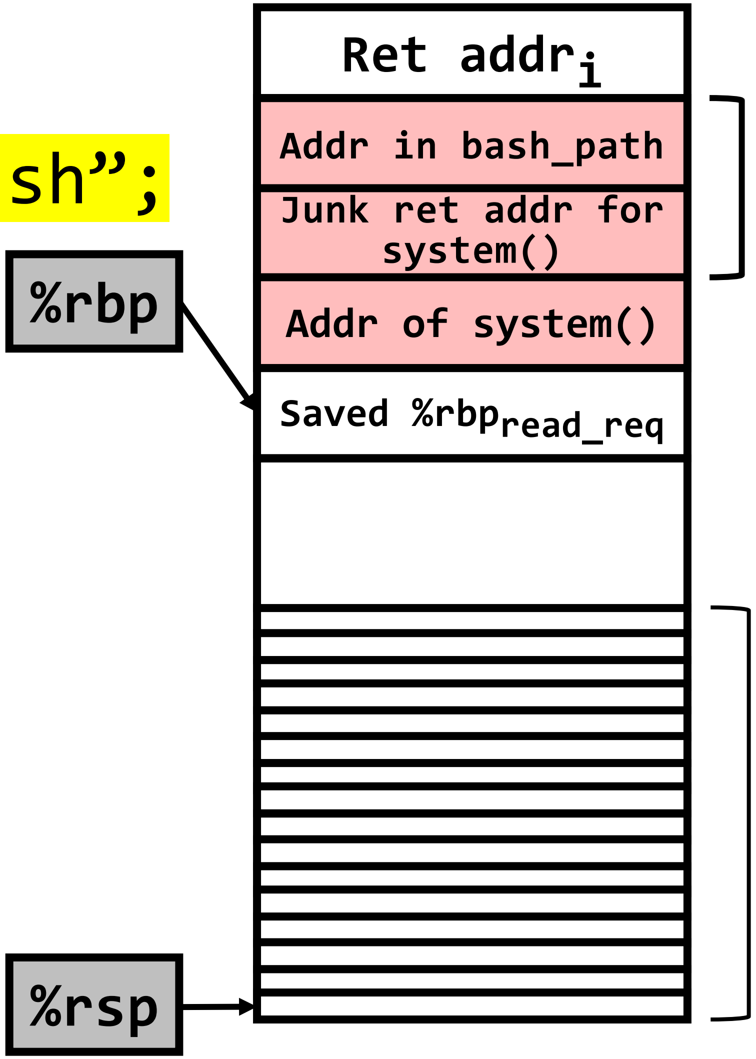
%rbp

%rsp

Overwrite return address with address of run_shell()

Suppose that a machine uses NX, but does not use stack canaries or ASLR, and contains this code . . .

```
char *bash_path = “/bin/sh”;

void read_req(){
    char buffer[128];
    gets(buf);
}

void run_ls(){
    system(“/bin/ls”);
}
```



Ret addr$_i$

Addr in bash_path

Junk ret addr for system()

Addr of system()

Saved %rbp$_{read\_req}$

%rbp

%rsp

buffer

This is what system() expects the stack to look like—system() will find its argument in the standard place!

What if the string "/bin/sh" isn't in the program at all?

Maybe attacks are impossible!

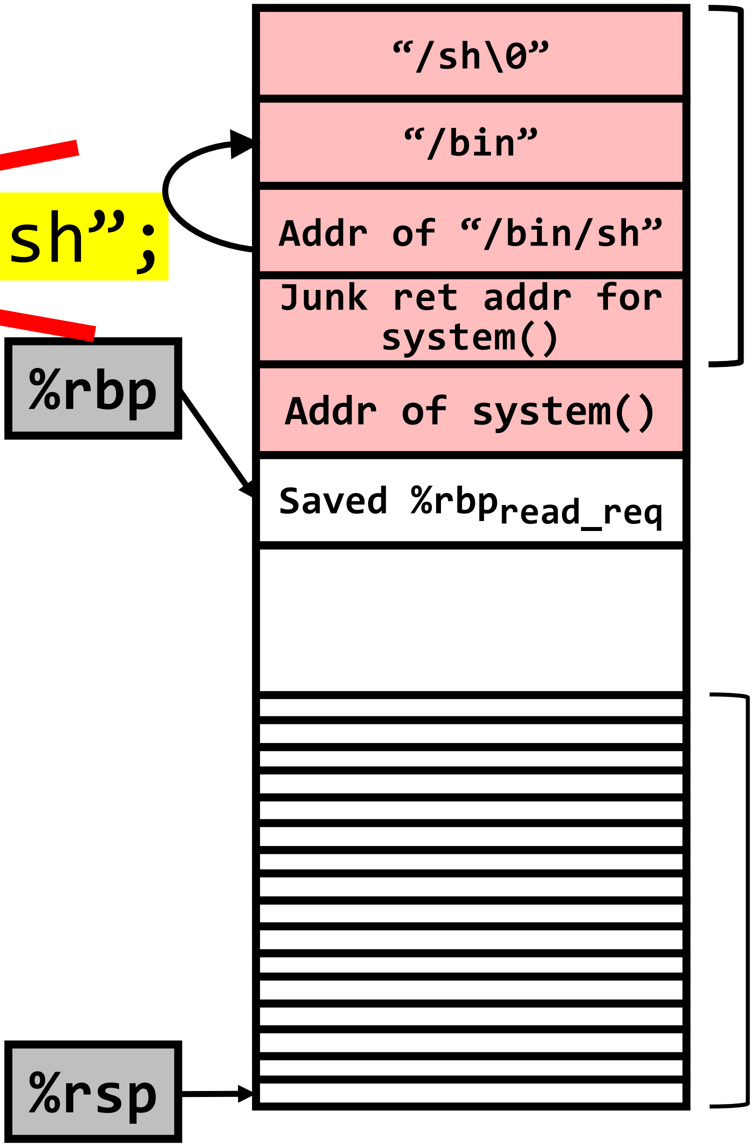
FALSE

ATTACKS ARE VERY POSSIBLE

Suppose that a machine uses NX, but does not use stack canaries or ASLR, and contains this code . . .

~~char *bash_path = “/bin/sh”;~~

```
void read_req(){
    char buffer[128];
    gets(buf);
}

void run_ls(){
    system(“/bin/ls”);
}
```



“/sh\0”

“/bin”

Addr of “/bin/sh”

Junk ret addr for system()

Addr of system()

Saved %rbp$_{read\_req}$

%rbp

%rsp

This is what system() expects the stack to look like—system() will find its argument in the standard place!

buffer

Attacker goal: Call `system(“/bin/sh”)` two times

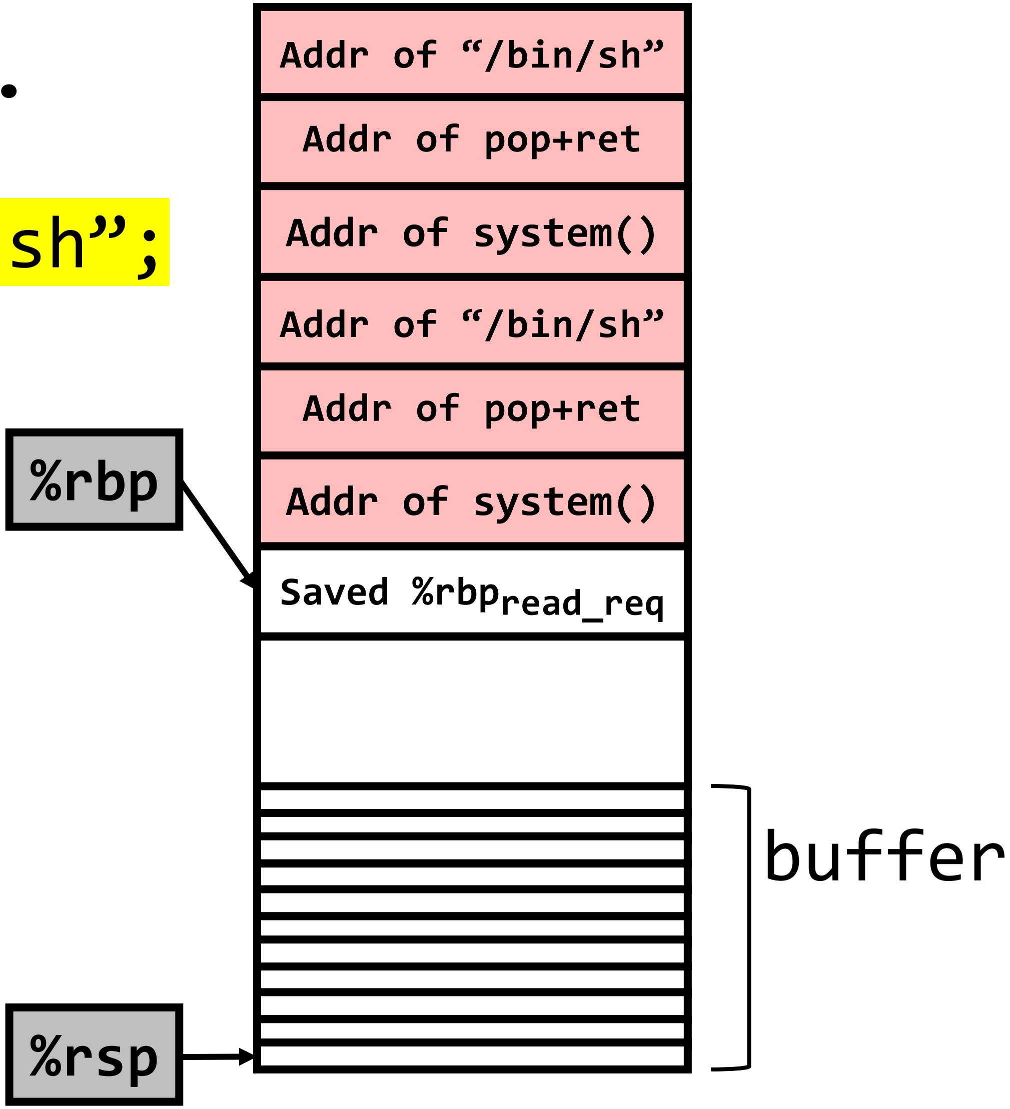
- Assume that the attacker knows three addresses:
  1. The address of `system()`
  2. The address of the string **`“/bin/sh”`** (remember that, if necessary, the attacker can push the string onto the stack!)
  3. The starting address for this block of opcodes:

     ```
     pop %eax  //Pops top-of-stack into %eax
     ret       //Pops top-of-stack and puts
               //it in %eip
     ```

     A block of opcodes is a “gadget.” [There are user-friendly tools to discover gadgets in a binary, e.g., `ROPgadget`.]

Suppose that a machine uses NX, but does not use stack canaries or ASLR, and contains this code . . .

```
char *bash_path = “/bin/sh”;

void read_req(){
    char buffer[128];
    gets(buf);
}

void run_ls(){
    system(“/bin/ls”);
}
```



Suppose that a machine uses NX, but does not use stack canaries or ASLR, and contains this code . . .

```
char *bash_path = “/bin/sh”;

void read_req(){
    char buffer[128];
    gets(buf);
}

void run_ls(){
    system(“/bin/ls”);
}
```

Addr of “/bin/sh”

Addr of pop+ret

Addr of system()

Addr of “/bin/sh”

Addr of pop+ret

[illegible]

The ret in the gadget is

The [illegible] the gadget is

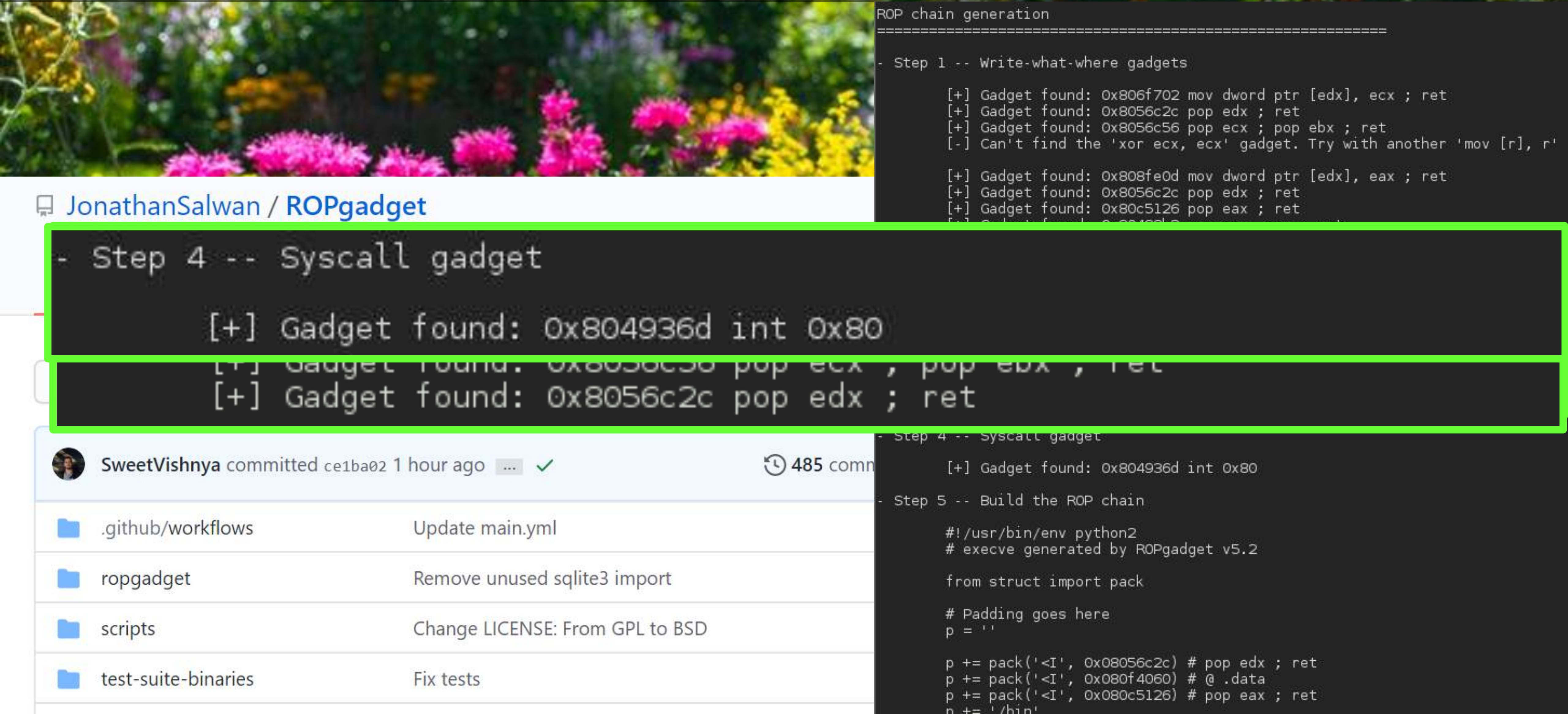
[illegible] executing,

[illegible]

[illegible] get . . .

instruction in system().

Non-trivial programs have a Turing-complete set of gadgets!



JonathanSalwan / ROPgadget

SweetVishnya committed ce1ba02 1 hour ago ... ✓ 485 comm

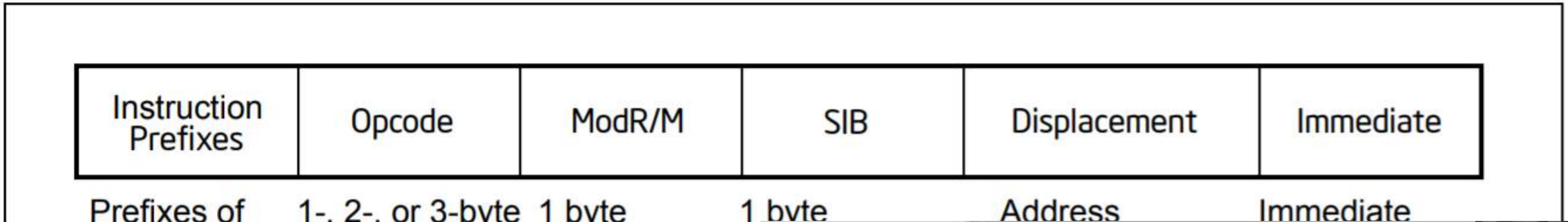
.github/workflows	Update main.yml
ropgadget	Remove unused sqlite3 import
scripts	Change LICENSE: From GPL to BSD
test-suite-binaries	Fix tests

```
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x806f702 mov dword ptr [edx], ecx ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
        [-] Can't find the 'xor ecx, ecx' gadget. Try with another 'mov [r], r'

        [+] Gadget found: 0x808fe0d mov dword ptr [edx], eax ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x80c5126 pop eax ; ret
```

```
- Step 4 -- Syscall gadget

        [+] Gadget found: 0x804936d int 0x80
```

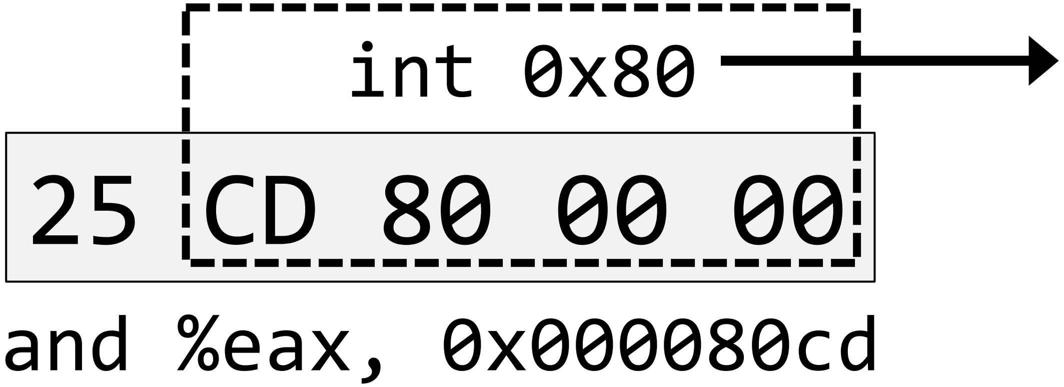
```
        [+] Gadget found: 0x8056c2c pop edx ; ret
```

```
- Step 4 -- Syscall gadget

        [+] Gadget found: 0x804936d int 0x80

- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget v5.2

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x08056c2c) # pop edx ; ret
        p += pack('<I', 0x080f4060) # @ .data
        p += pack('<I', 0x080c5126) # pop eax ; ret
        p += '/bin'
```



Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
Prefixes of 1 byte each (optional)1, 2	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (i	Address	Immediate

x86: The Oil Isn’t Washing Off

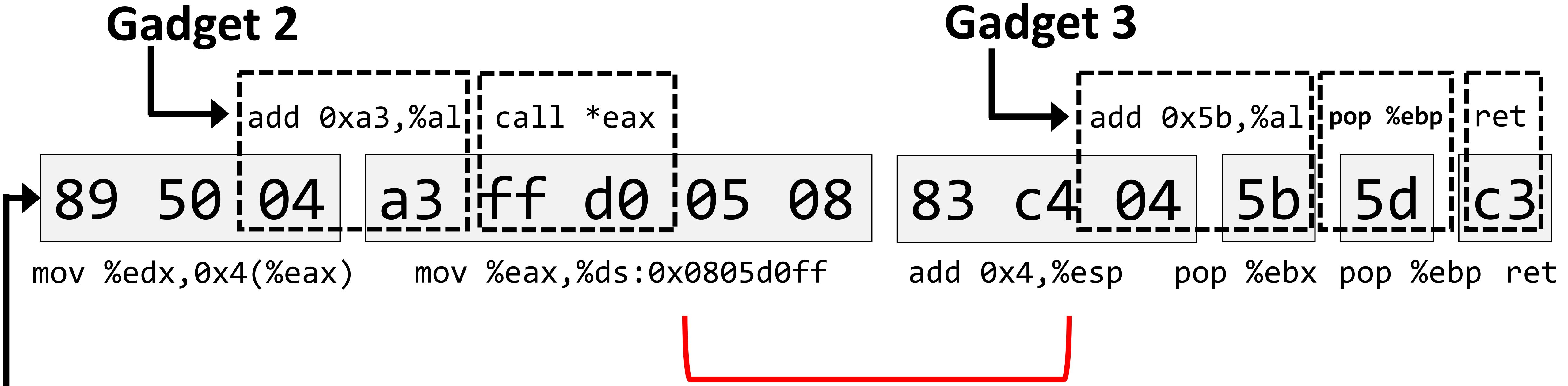


int 0x80

25 CD 80 00 00

and %eax, 0x000080cd

Used by 32-bit Linux processes to invoke a system call!



Gadget 2

Gadget 3

add 0xa3,%al

call *eax

add 0x5b,%al

pop %ebp

ret

89 50 04 a3 ff d0 05 08 83 c4 04 5b 5d c3

mov %edx,0x4(%eax)

mov %eax,%ds:0x0805d0ff

add 0x4,%esp

pop %ebx

pop %ebp

ret

Gadget 1

Jumping into this range will cause the CPU to raise an “invalid opcode” exception!

- In ROP, attacker creates a program whose execution is driven by the stack pointer!
  - As the stack pointer moves down the stack (i.e., UP in memory), the program executes gadgets whose code is preexisting in the executable
  - This attack can’t be stopped by NX bits!
- But what about stack canaries?

Defeating Stack Canaries

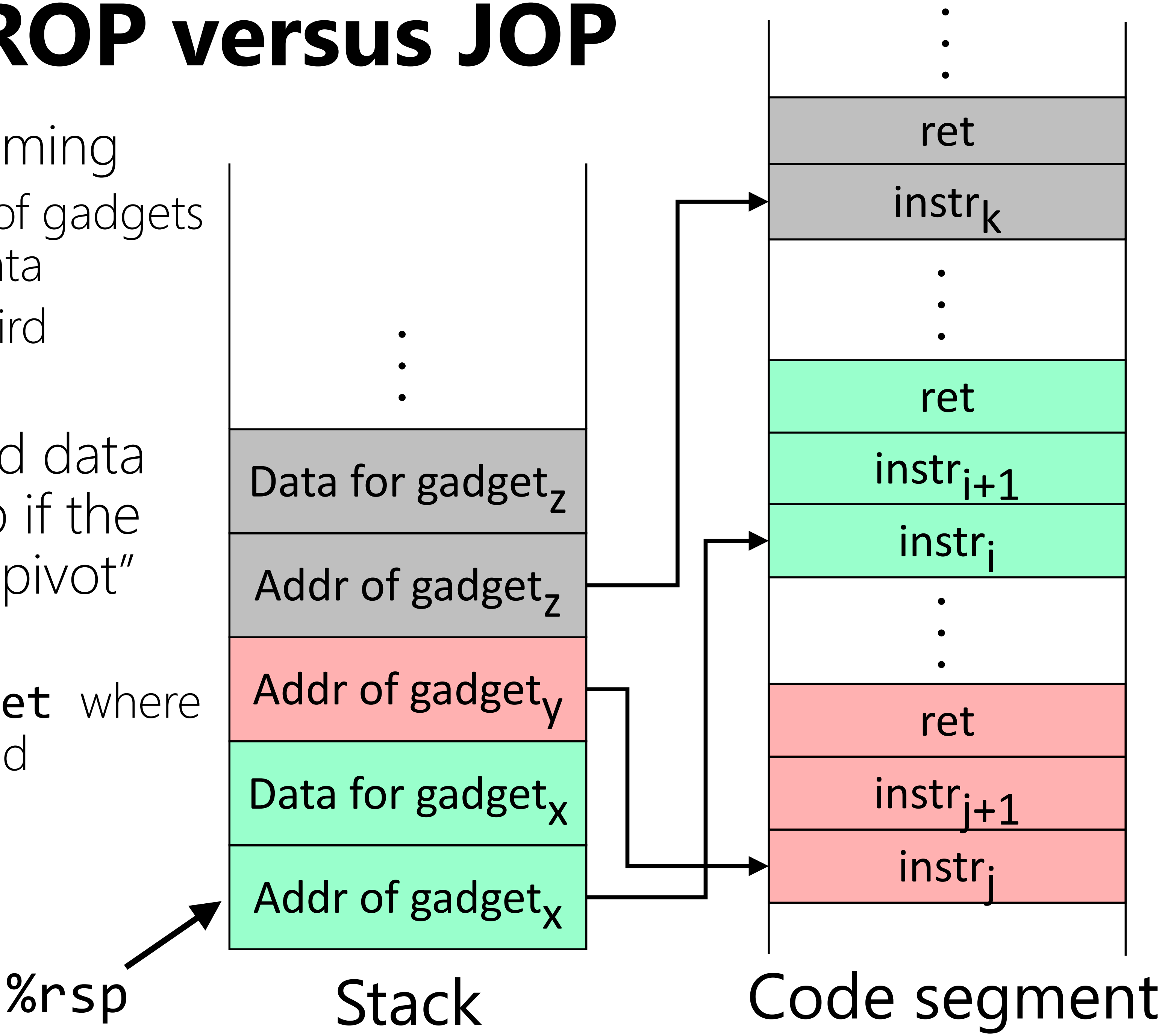
- Approach 1: Non-contiguous writes
  - A non-contiguous write vulnerability allows the attacker to only write locations of interest (but the attacker must know where to write . . .)
  - Type confusions are a common source of non-contiguous writes
- Approach 2: Stack pivoting
  - The attacker corrupts a register that is later used by the program to modify the stack
  - Ex: `mov %rax, %rsp` where `%rax` is attacker controlled

Defeating ASLR

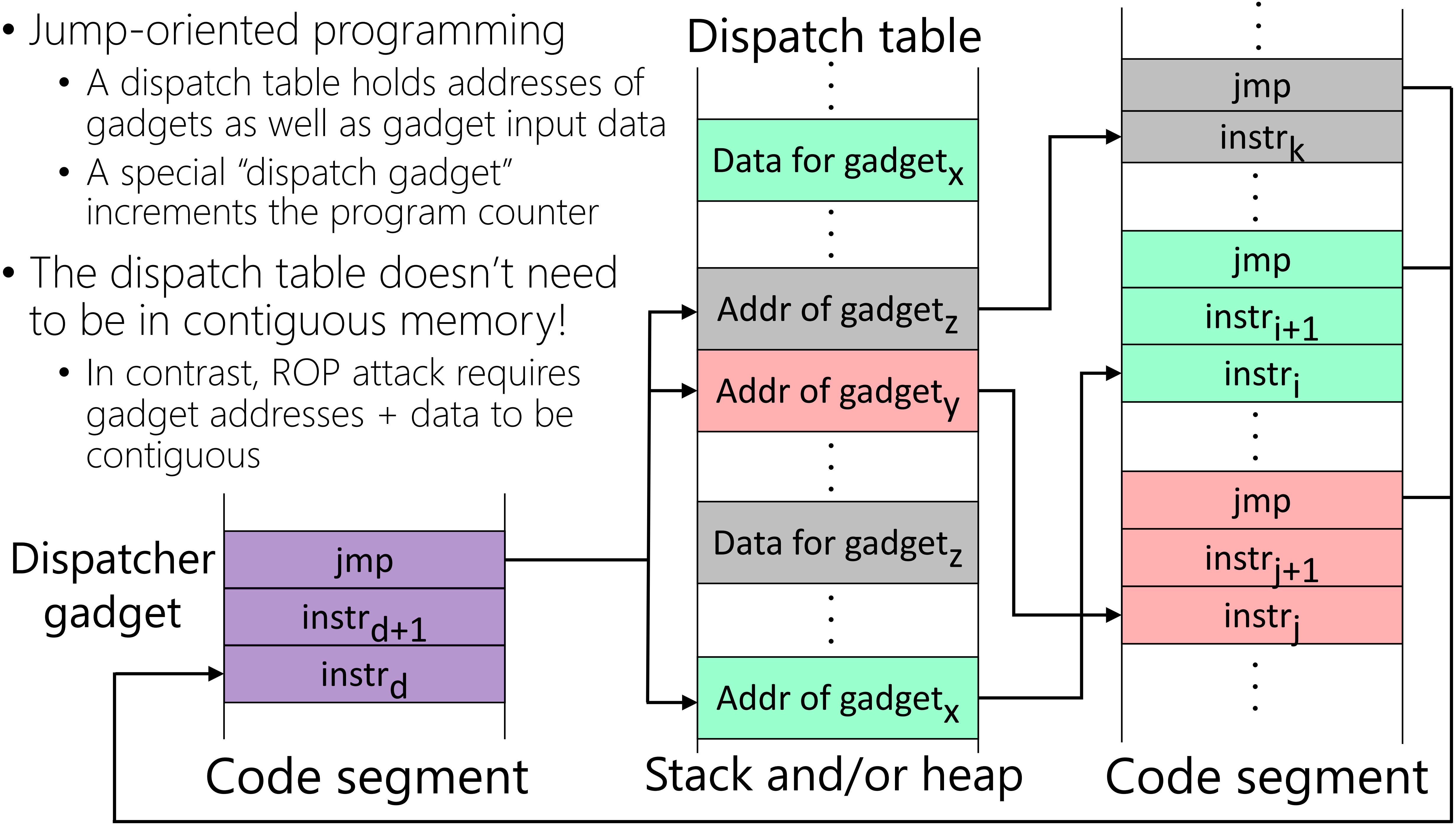
- Usually attackers try to use memory read vulnerabilities to find a single concrete memory address that enables the discovery of other concrete addresses
  - Ex: Discovering a single code pointer to a function in `libc` lets the attacker determine the location of any function in `libc` (since those relative offsets aren't changed by ASLR!)
  - Ex: An over-read of a buffer on the stack can disclose return addresses, narrowing the location of the stack
- Attackers can also leverage program quirks
  - Ex: A program might load non-relocatable libraries—look for gadgets there!
  - Ex: ASLR doesn't change after `fork()`, so the attacker can repeatedly guess locations and reuse work if crashes occur and new processes are respawned

ROP versus JOP

- Return-oriented programming
  - Stack hold the addresses of gadgets as well as gadget input data
  - Stack pointer acts as a weird program counter
- The gadget addresses and data can be stored in the heap if the attacker can find a “stack pivot” gadget
  - Ex: `mov %rax, %rsp; ret` where `%rax` is attacker-controlled

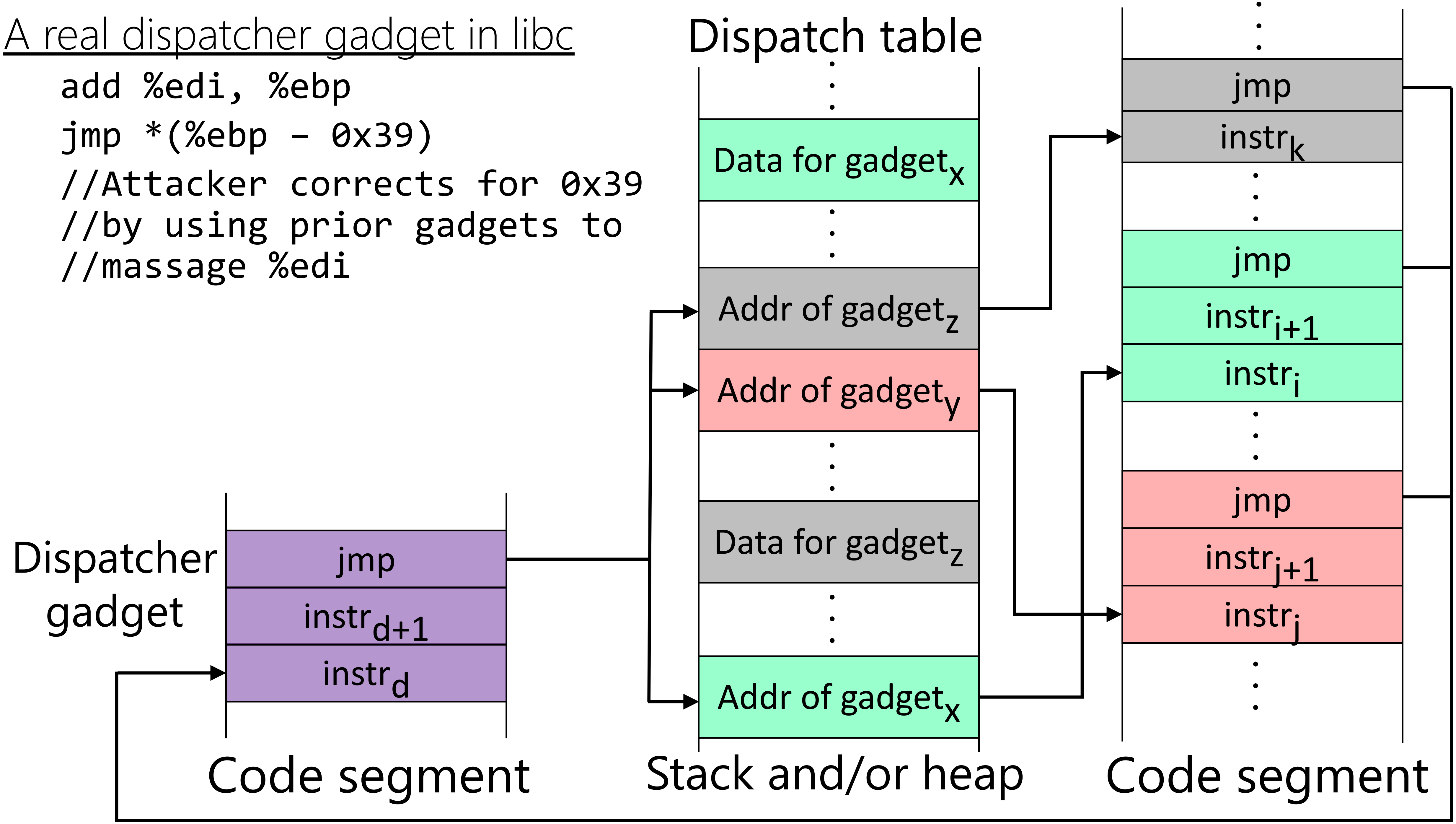


- Jump-oriented programming
  - A dispatch table holds addresses of gadgets as well as gadget input data
  - A special "dispatch gadget" increments the program counter
- The dispatch table doesn't need to be in contiguous memory!
  - In contrast, ROP attack requires gadget addresses + data to be contiguous



A real dispatcher gadget in libc

```
add %edi, %ebp
jmp *(%ebp – 0x39)
//Attacker corrects for 0x39
//by using prior gadgets to
//massage %edi
```



A real dispatcher gadget in libc

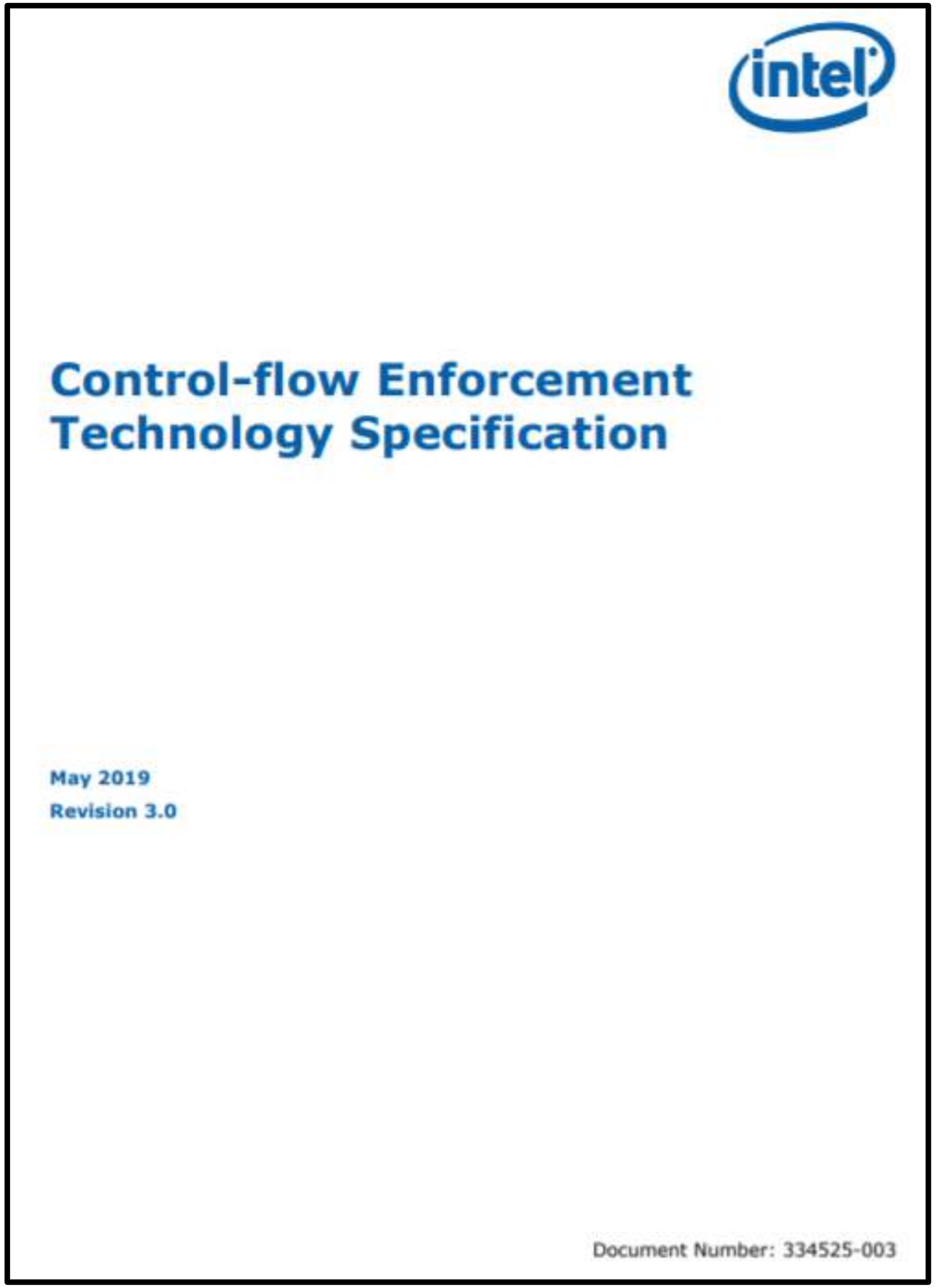
```
add %edi, %ebp
jmp *(%ebp – 0x39)
//Attacker corrects for 0x39
//by using prior gadgets to
//massage %edi
```

A real JOP attack using libc gadgets

```
popa            //Load all registers from
                //attacker-controlled stack
cmc                  //No practical effect
jmp *(%ecx)          //Go to dispatcher

xchg %ecx, %eax      //Swap reg values
fdiv st3, st0        //No practical effect
jmp *(%esi – 0xf)    //Go to dispatcher

mov *(%esi + 0xc), %eax //Set %eax
mov %eax, %esp       //No practical effect
call *(%esi + 0x4)   //Go to dispatcher

sysenter             //Invoke a system call,
                     //with the argument
                     //registers %eax (the,
                     //syscall num), %ebx,
                     //%ecx, %edx set using
                     //popa and gadgets
```

THIS ALL SOUNDS TERRIBLE

BUT IT’S NOT AS BAD AS IT SEEMS



Control-flow Enforcement Technology Specification

May 2019

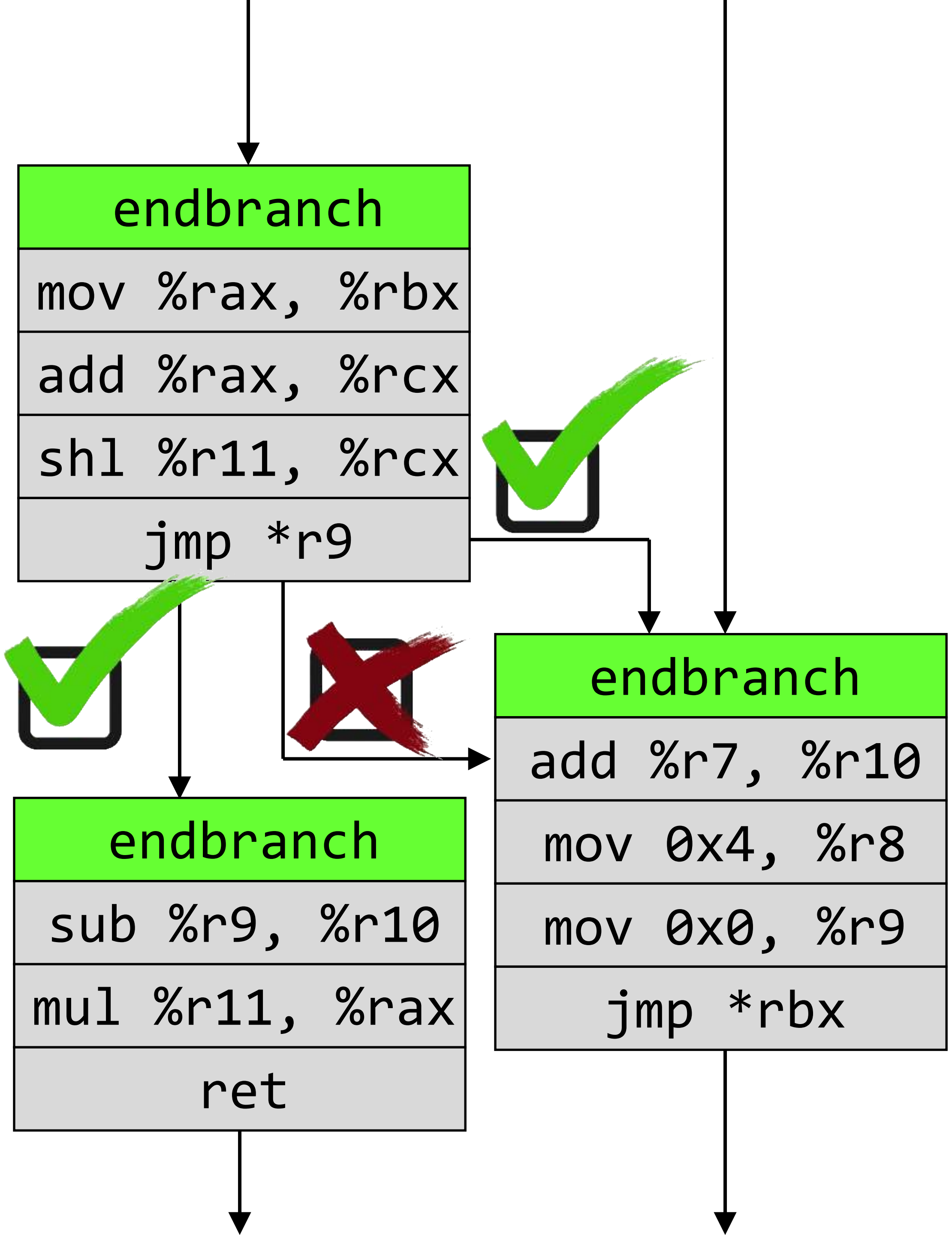
Revision 3.0

Document Number: 334525-003

- Intel’s CET spec introduces two technologies
  - Shadow stacks (as discussed last lecture!)
  - Indirect branch tracing (which we’ll discuss next)
- CET ships with:
  - Tiger Lake chips that arrive this year
  - Upcoming Xeon chips
- AMD says that it’s working on equivalent technology

Indirect Branch Tracking (IBT)

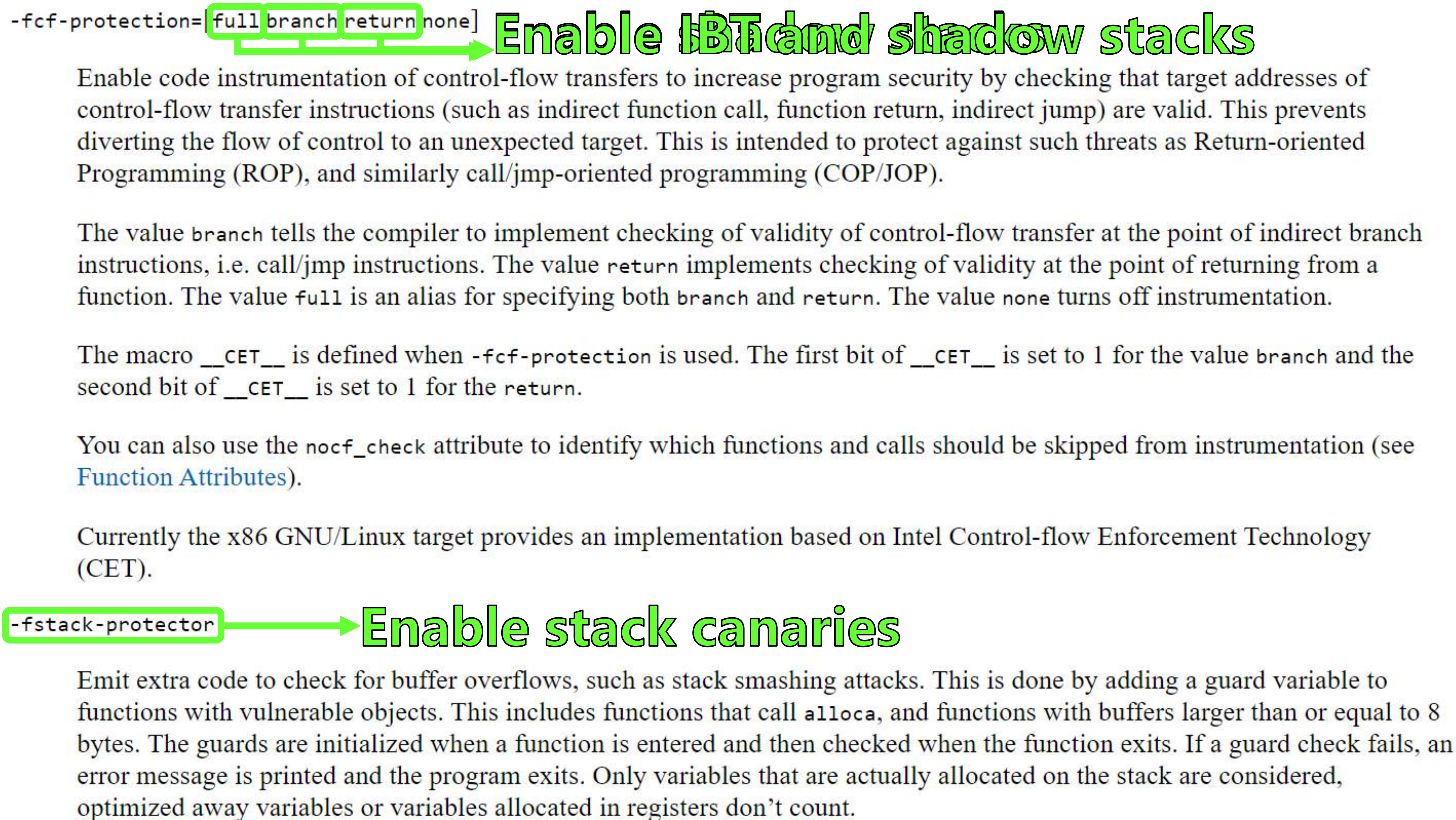
- IBT introduces **endbranch** (which evaluates to **nop** on older CPUs)
  - Hardware ensures that any **jmp**/**call** target is an **endbranch** . . .
  - . . . unless the **jmp**/**call** instruction is prefixed with **3EH** (“no-track”)
- To use IBT:
  - Set configuration registers like:
    - “Master enable bit” **%cr4.cet**
    - “User-mode IBT enable bit” **%ia32_u_cet.endbr_en**
  - Use a compiler that inserts **endbranch** at each potential indirect jump target



```
endbranch
mov %rax, %rbx
add %rax, %rcx
shl %r11, %rcx
jmp *r9
```

```
endbranch
sub %r9, %r10
mul %r11, %rax
ret
```

```
endbranch
add %r7, %r10
mov 0x4, %r8
mov 0x0, %r9
jmp *rbx
```



```
-fcf-protection=[full|branch|return|none]
```

Enable IBT and shadow stacks

Enable code instrumentation of control-flow transfers to increase program security by checking that target addresses of control-flow transfer instructions (such as indirect function call, function return, indirect jump) are valid. This prevents diverting the flow of control to an unexpected target. This is intended to protect against such threats as Return-oriented Programming (ROP), and similarly call/jmp-oriented programming (COP/JOP).

The value `branch` tells the compiler to implement checking of validity of control-flow transfer at the point of indirect branch instructions, i.e. call/jmp instructions. The value `return` implements checking of validity at the point of returning from a function. The value `full` is an alias for specifying both `branch` and `return`. The value `none` turns off instrumentation.

The macro `__CET__` is defined when `-fcf-protection` is used. The first bit of `__CET__` is set to 1 for the value `branch` and the second bit of `__CET__` is set to 1 for the `return`.

You can also use the `nocf_check` attribute to identify which functions and calls should be skipped from instrumentation (see Function Attributes).

Currently the x86 GNU/Linux target provides an implementation based on Intel Control-flow Enforcement Technology (CET).

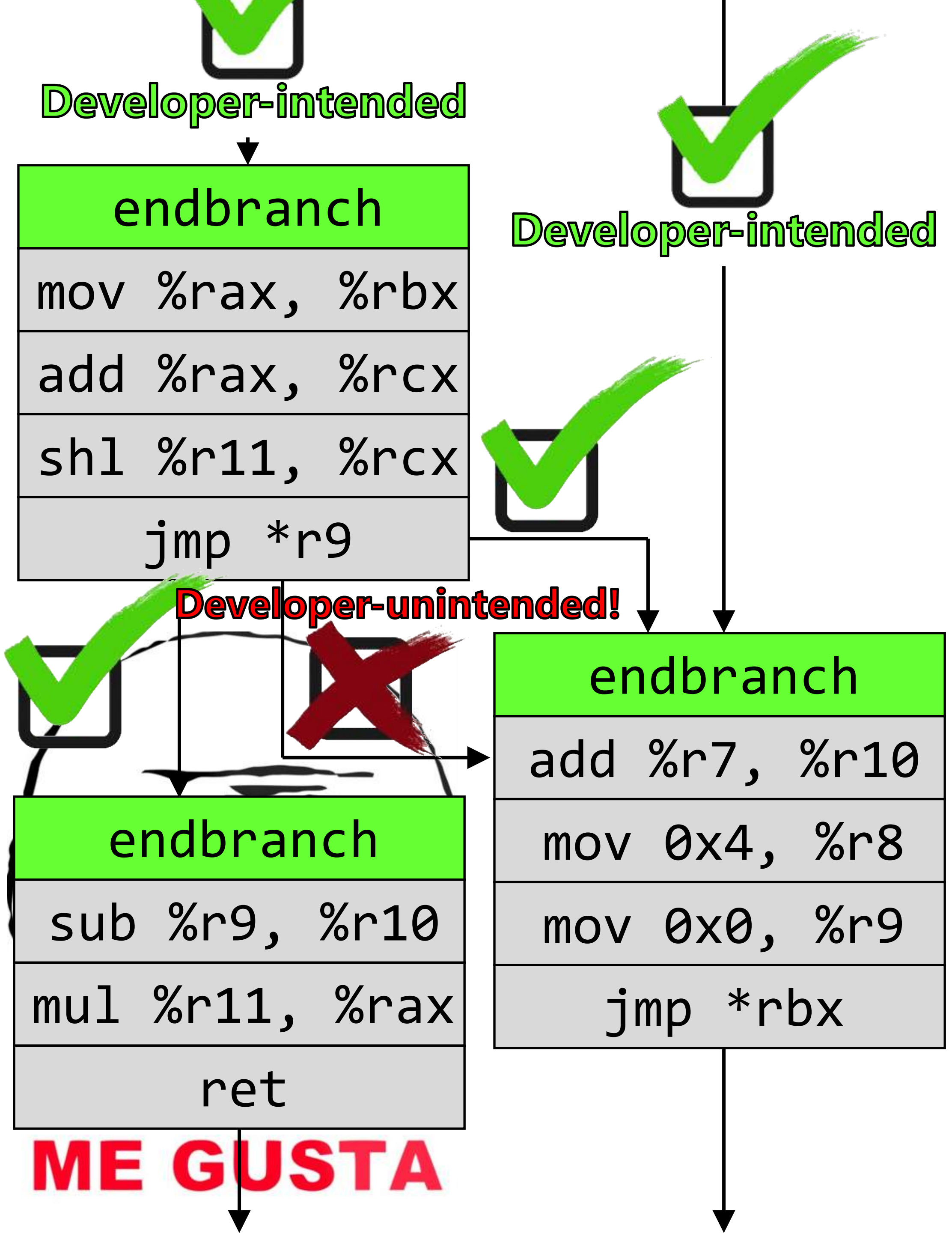
```
-fstack-protector
```

Enable stack canaries

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than or equal to 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits. Only variables that are actually allocated on the stack are considered, optimized away variables or variables allocated in registers don't count.

Indirect Branch Tracking (IBT)

- IBT provides relaxed CFI: a given **`jmp`**/ **`call`** can go to any valid **`jmp`**/**`call`** target
- ROP and JOP are still possible if the attacker can find enough **`endbranch`**-preceded gadgets
- However, IBT in concert with shadow stacks, ASLR, NX bits, and stack canaries make it much harder for attackers to divert control flow!



```
endbranch
mov %rax, %rbx
add %rax, %rcx
shl %r11, %rcx
jmp *r9
```

```
endbranch
sub %r9, %r10
mul %r11, %rax
ret
```

```
endbranch
add %r7, %r10
mov 0x4, %r8
mov 0x0, %r9
jmp *rbx
```

What Do Threats Look Like Today?

Microsoft

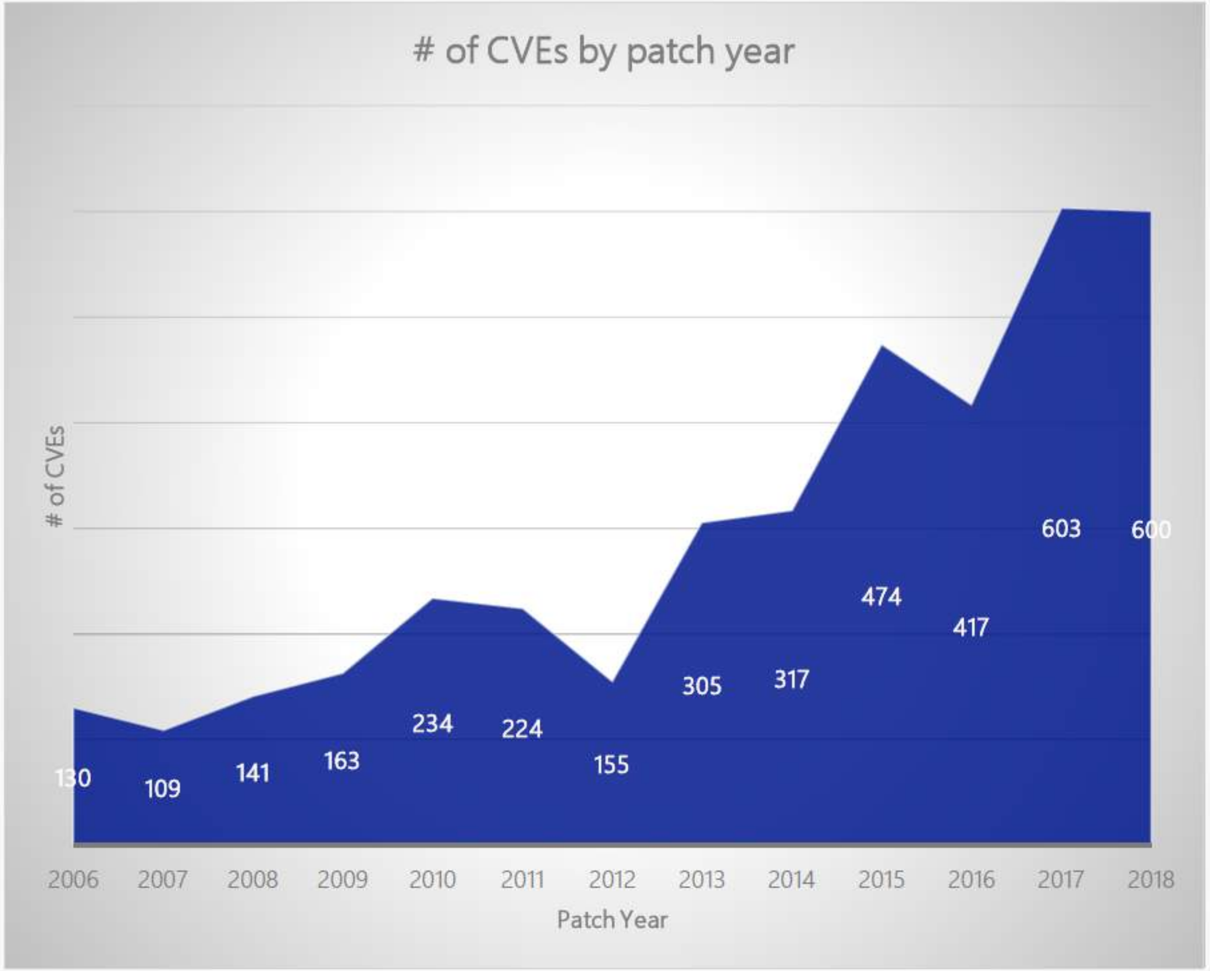
Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape

Matt Miller (@epakskape)
Microsoft Security Response Center (MSRC)

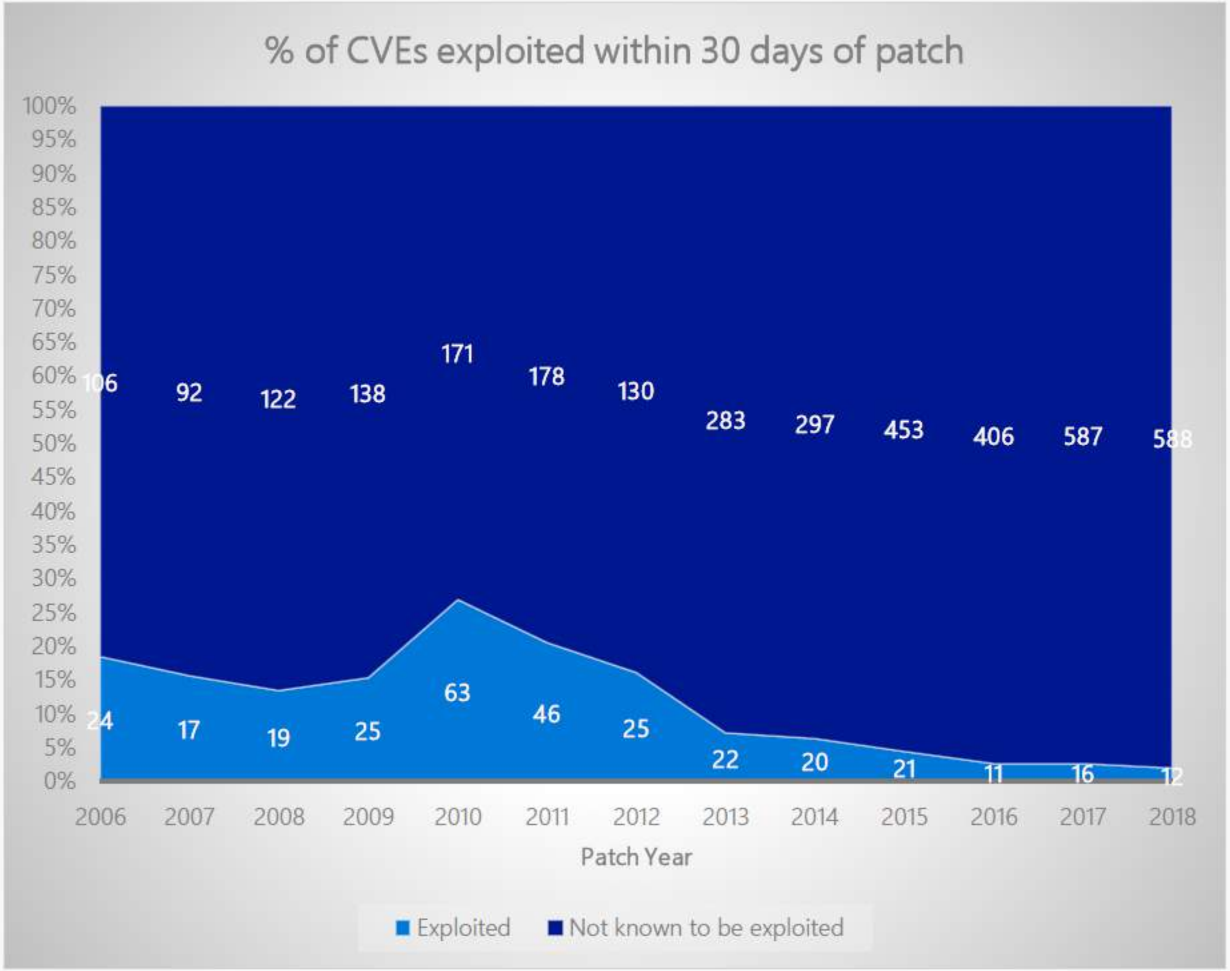
BlueHat IL
February 7th, 2019

This presentation is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

More vulnerabilities fixed, fewer known exploits



# of CVEs by patch year



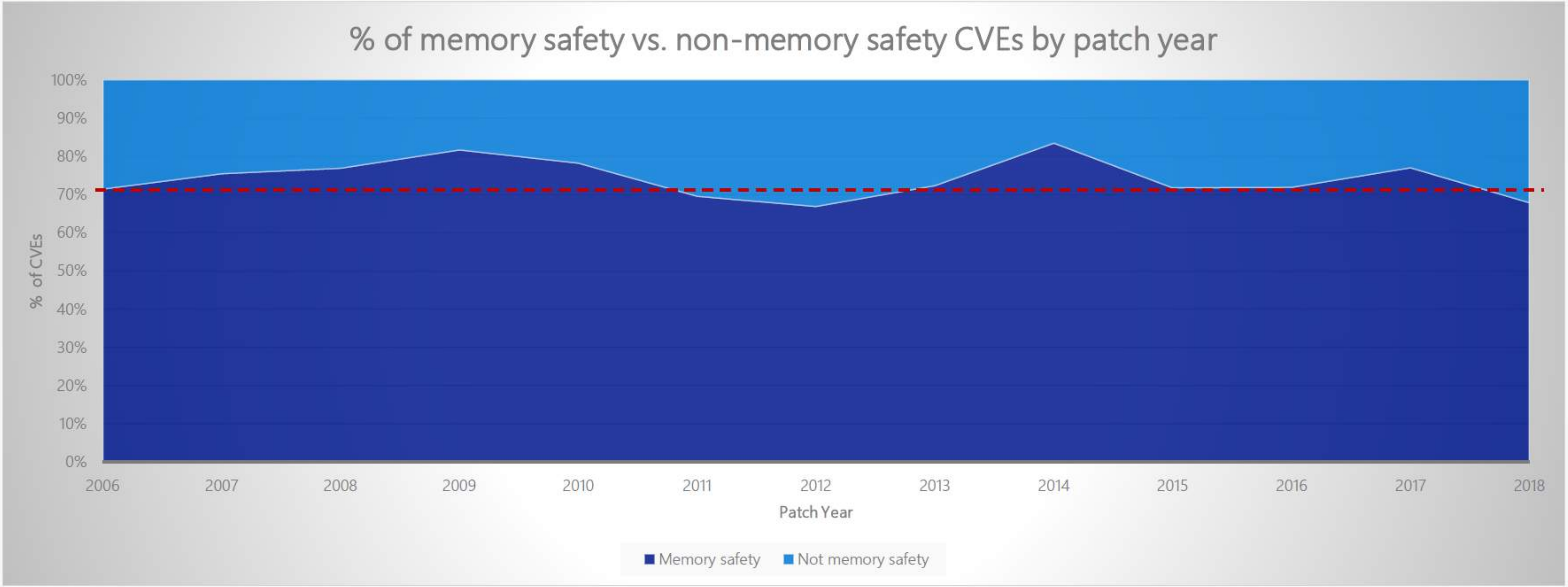
% of CVEs exploited within 30 days of patch

On the surface, risk appears to be *increasing*

But known actualized risk appears to be *decreasing*

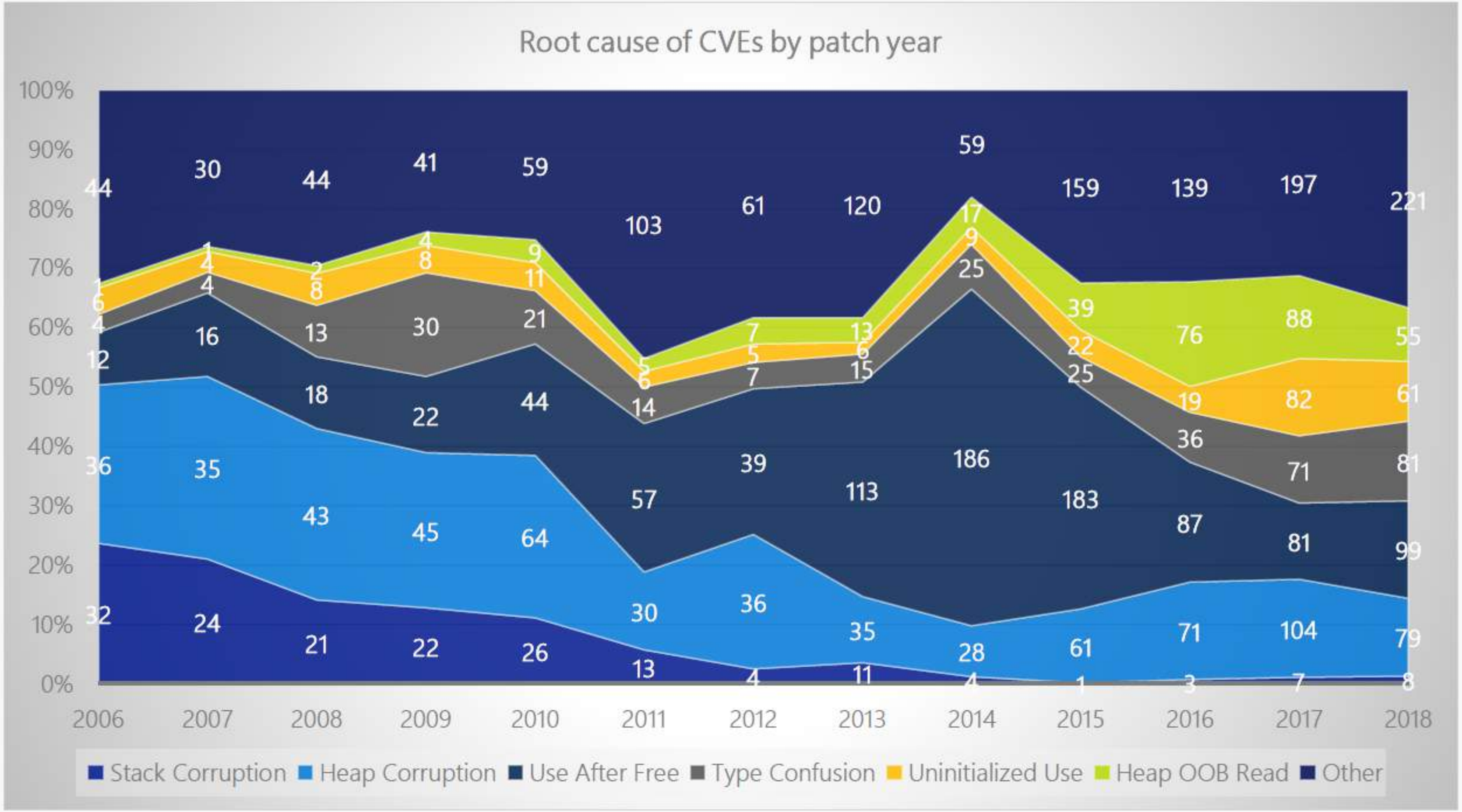
Memory safety issues remain dominant

We closely study the root cause trends of vulnerabilities & search for patterns



~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

#1: heap out-of-bounds

#2: use after free

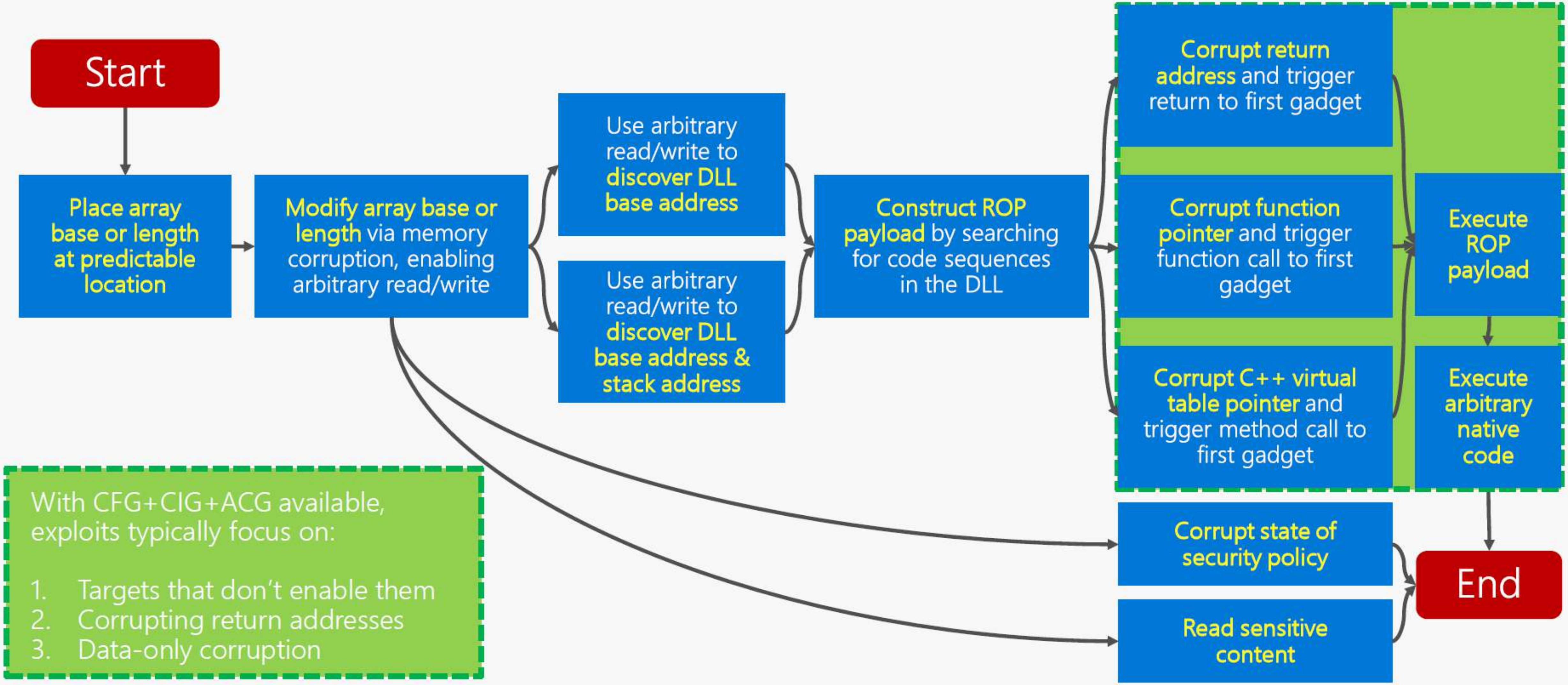
#3: type confusion

#4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Challenges with breaking exploitation techniques [2/4]

Most exploits have followed the same general steps since ~2016



Start

Place array base or length at predictable location

Modify array base or length via memory corruption, enabling arbitrary read/write

Use arbitrary read/write to **discover DLL base address**

Use arbitrary read/write to **discover DLL base address & stack address**

Construct ROP payload by searching for code sequences in the DLL

Corrupt return address and trigger return to first gadget

Corrupt function pointer and trigger function call to first gadget

Corrupt C++ virtual table pointer and trigger method call to first gadget

Execute ROP payload

Execute arbitrary native code

Corrupt state of security policy

Read sensitive content

End

With CFG+CIG+ACG available, exploits typically focus on:

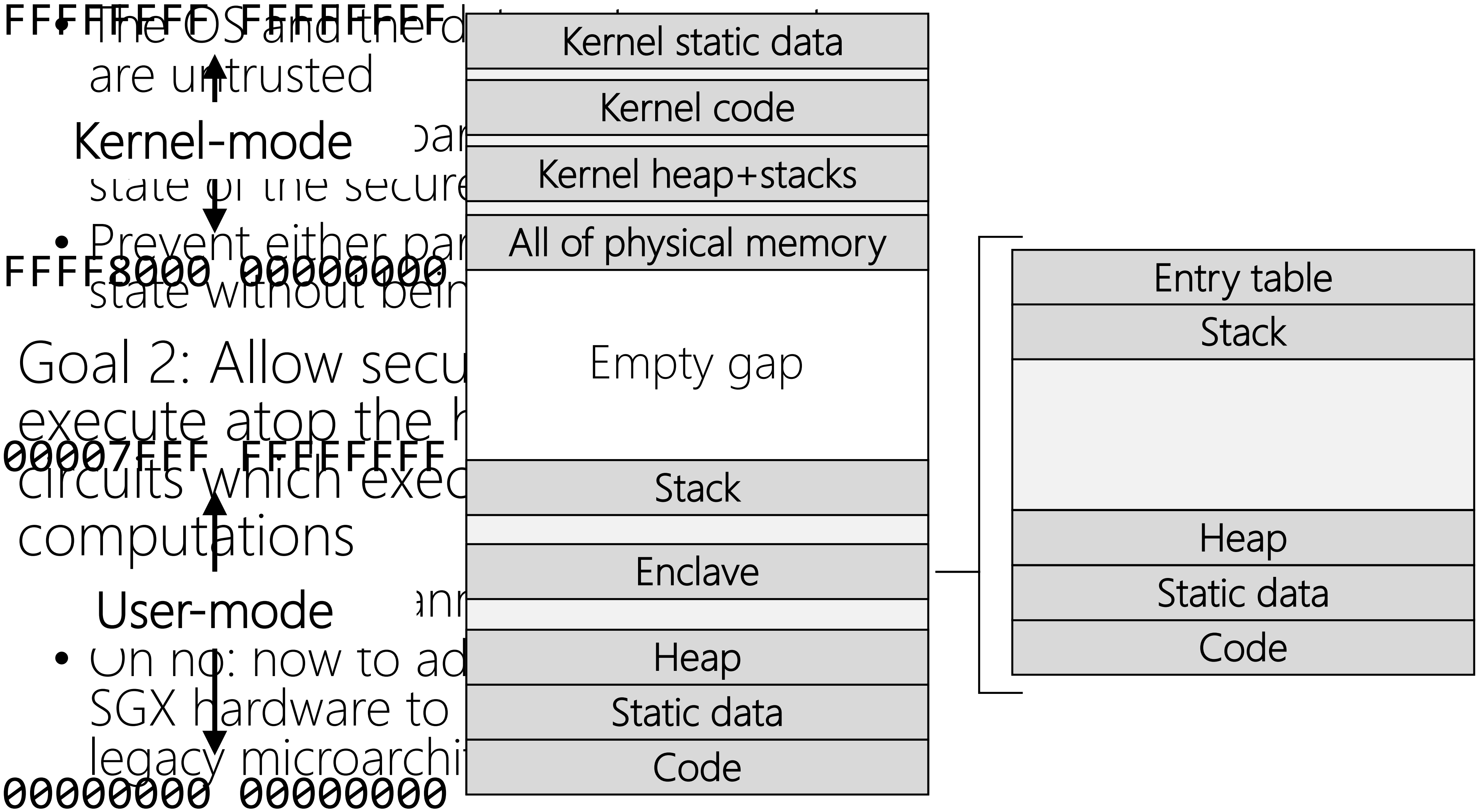
1. Targets that don't enable them
2. Corrupting return addresses
3. Data-only corruption

SGX

Overview of SGX

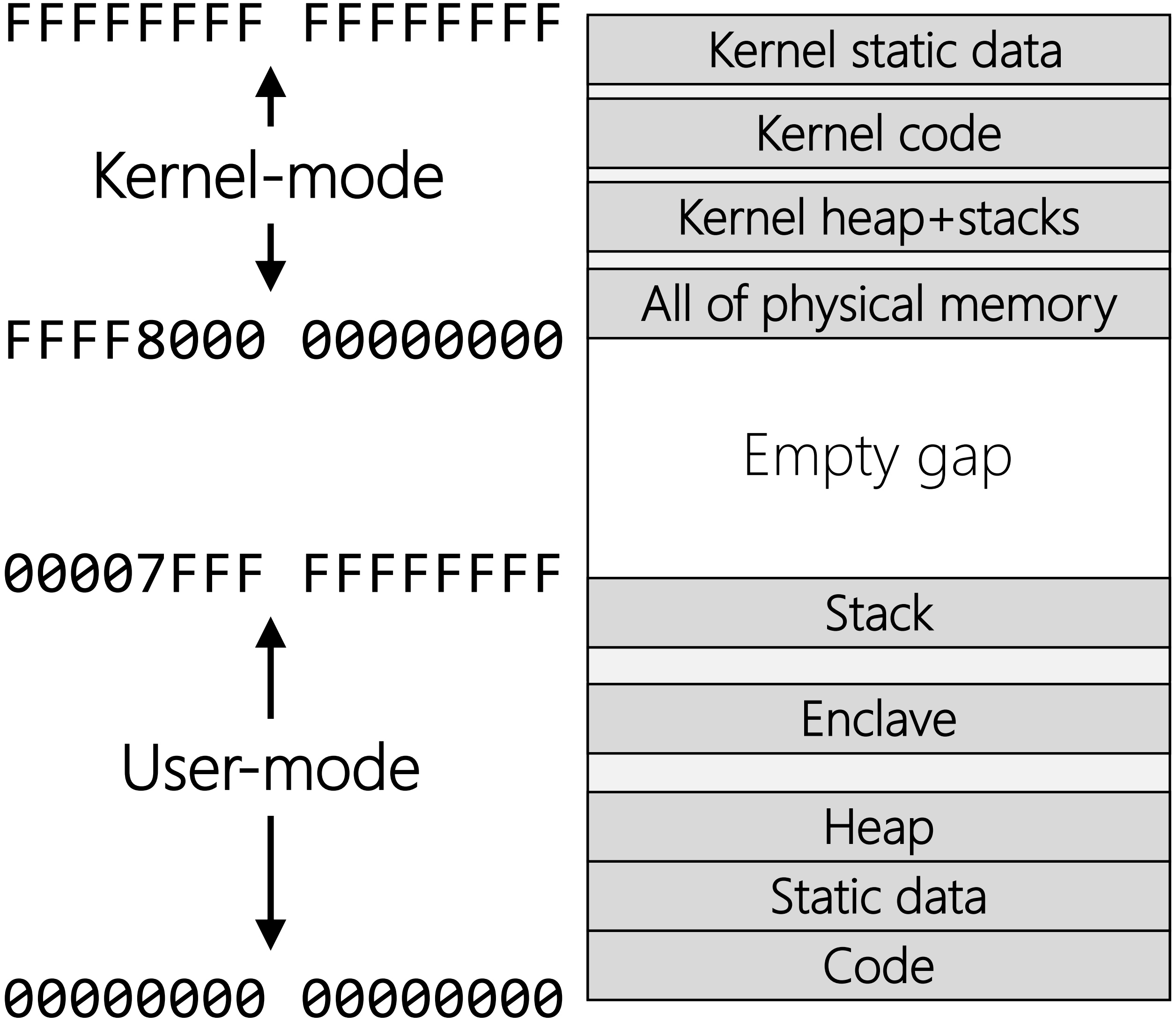
- Goal 1: Allow a client to interact with a secure server-side computation
  - The OS and the d are untrusted
  - par state of the secure
  - Prevent either par state without bein
- Goal 2: Allow secu execute atop the h circuits which exec computations
  - ann
  - Oh no: how to ad SGX hardware to legacy microarchi

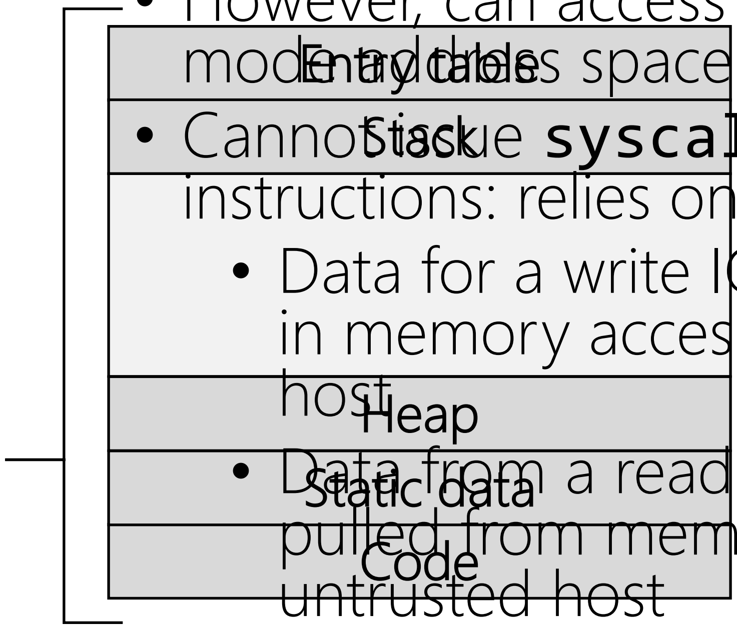
x86-64: Virtual address space of untrusted host



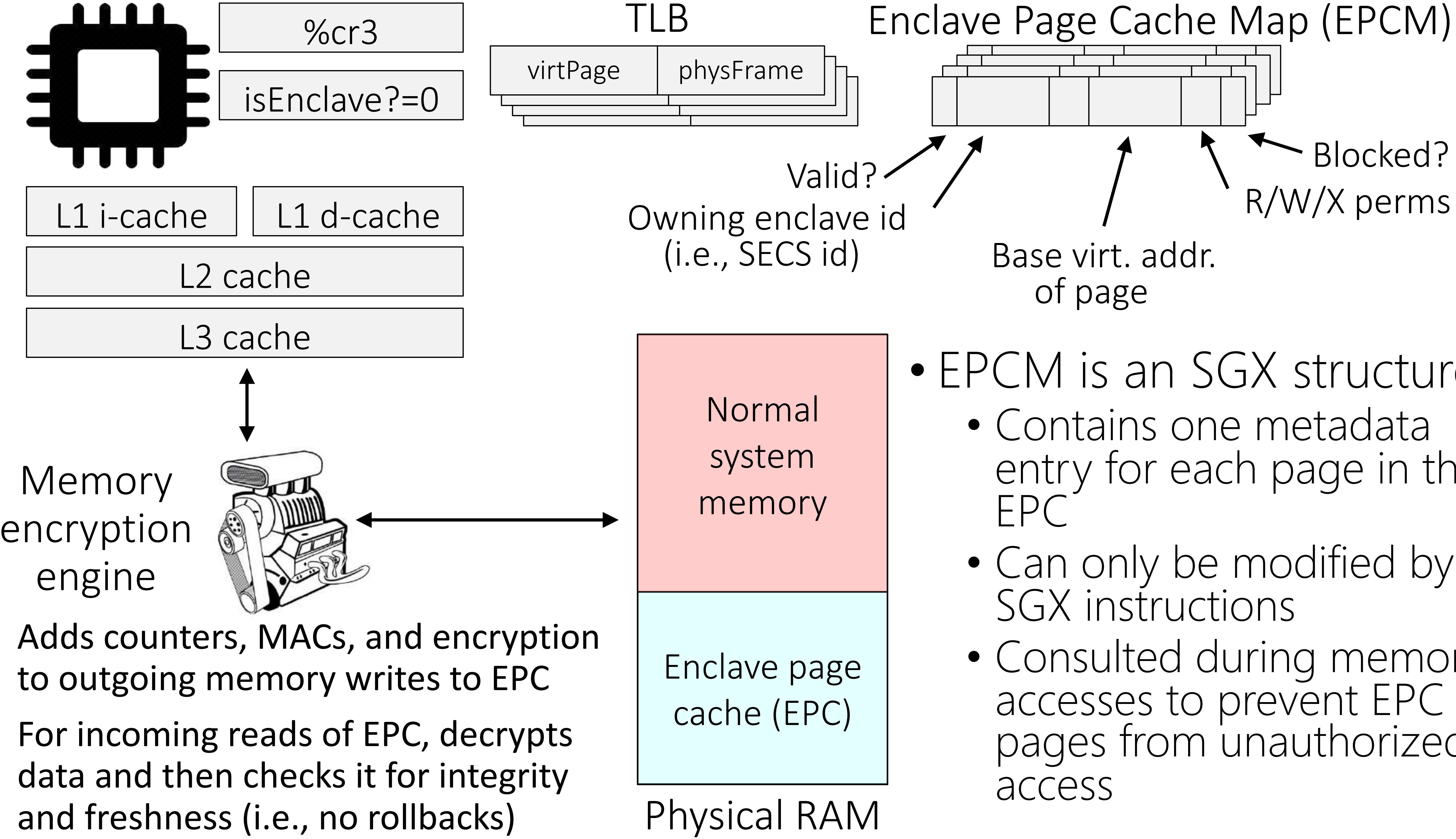
Overview of SGX

x86-64: Virtual address space of untrusted host





- Untrusted host process embeds an enclave
  - Cannot access enclave pages: reads return -1, writes are ignored
  - Jumps to enclave code using `EENTER`
- Enclave code runs at Ring 3 (i.e., the least-privileged level)
  - However, can access the entire user-mode address space of host
  - Cannot issue `syscall` or `int` instructions: relies on host for IO!
    - Data for a write IO must be placed in memory accessible to untrusted host
    - Data from a read IO must be pulled from memory accessible to untrusted host
  - Returns to untrusted host using `EEXIT` instruction



%cr3

isEnclave?=0

TLB

virtPage | physFrame

Enclave Page Cache Map (EPCM)

Valid?

Owning enclave id
(i.e., SECS id)

Base virt. addr.
of page

R/W/X perms

Blocked?

L1 i-cache

L1 d-cache

L2 cache

L3 cache

Memory
encryption
engine

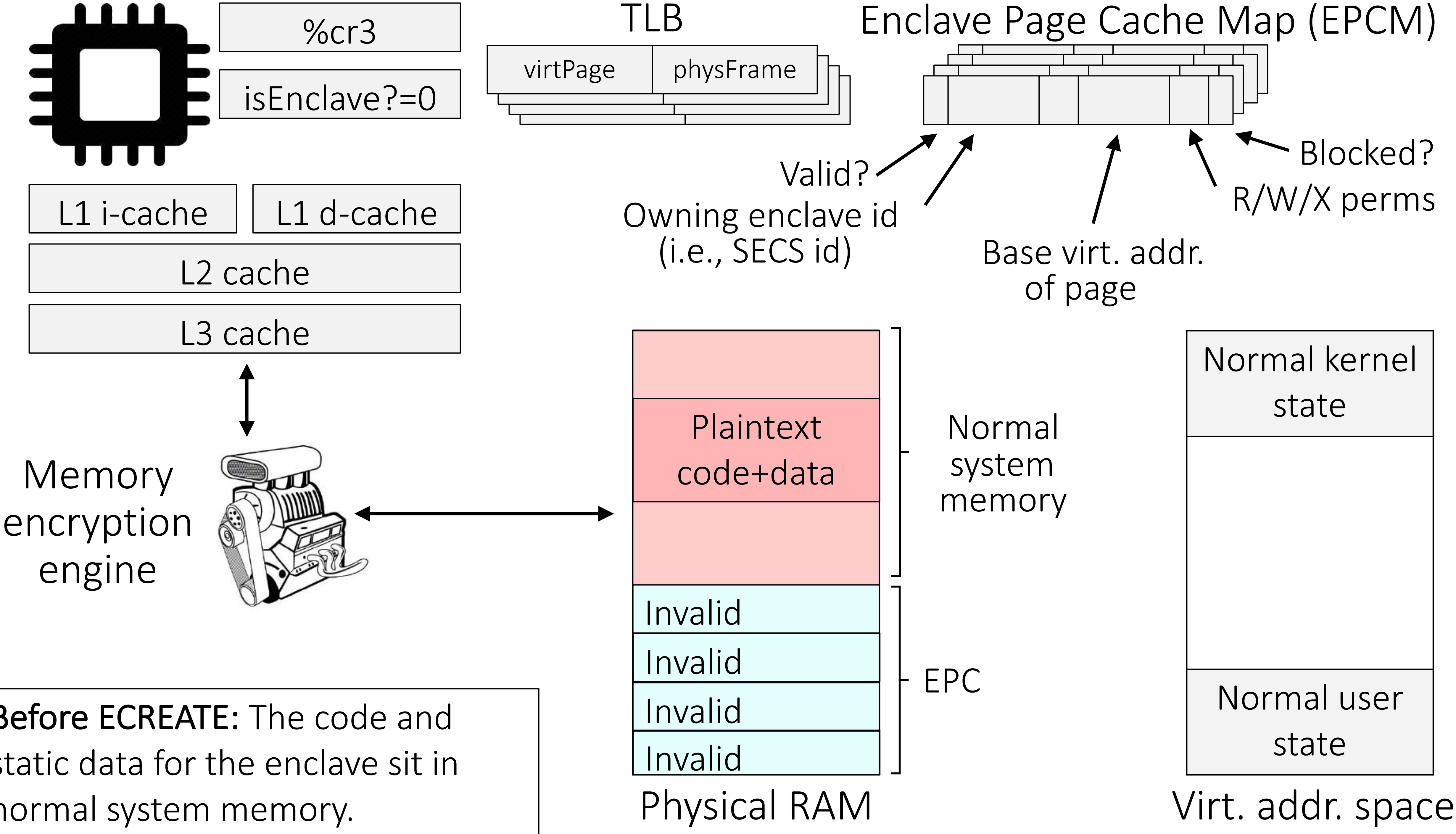
Normal
system
memory

Enclave page
cache (EPC)

Physical RAM

- **Adds counters, MACs, and encryption to outgoing memory writes to EPC**
- **For incoming reads of EPC, decrypts data and then checks it for integrity and freshness (i.e., no rollbacks)**

- EPCM is an SGX structure
  - Contains one metadata entry for each page in the EPC
  - Can only be modified by SGX instructions
  - Consulted during memory accesses to prevent EPC pages from unauthorized access



%cr3

isEnclave?=0

TLB

virtPage

physFrame

Enclave Page Cache Map (EPCM)

Valid?

Owning enclave id (i.e., SECS id)

Base virt. addr. of page

R/W/X perms

Blocked?

L1 i-cache

L1 d-cache

L2 cache

L3 cache

Memory encryption engine

Plaintext code+data

Normal system memory

Invalid

Invalid

Invalid

Invalid

EPC

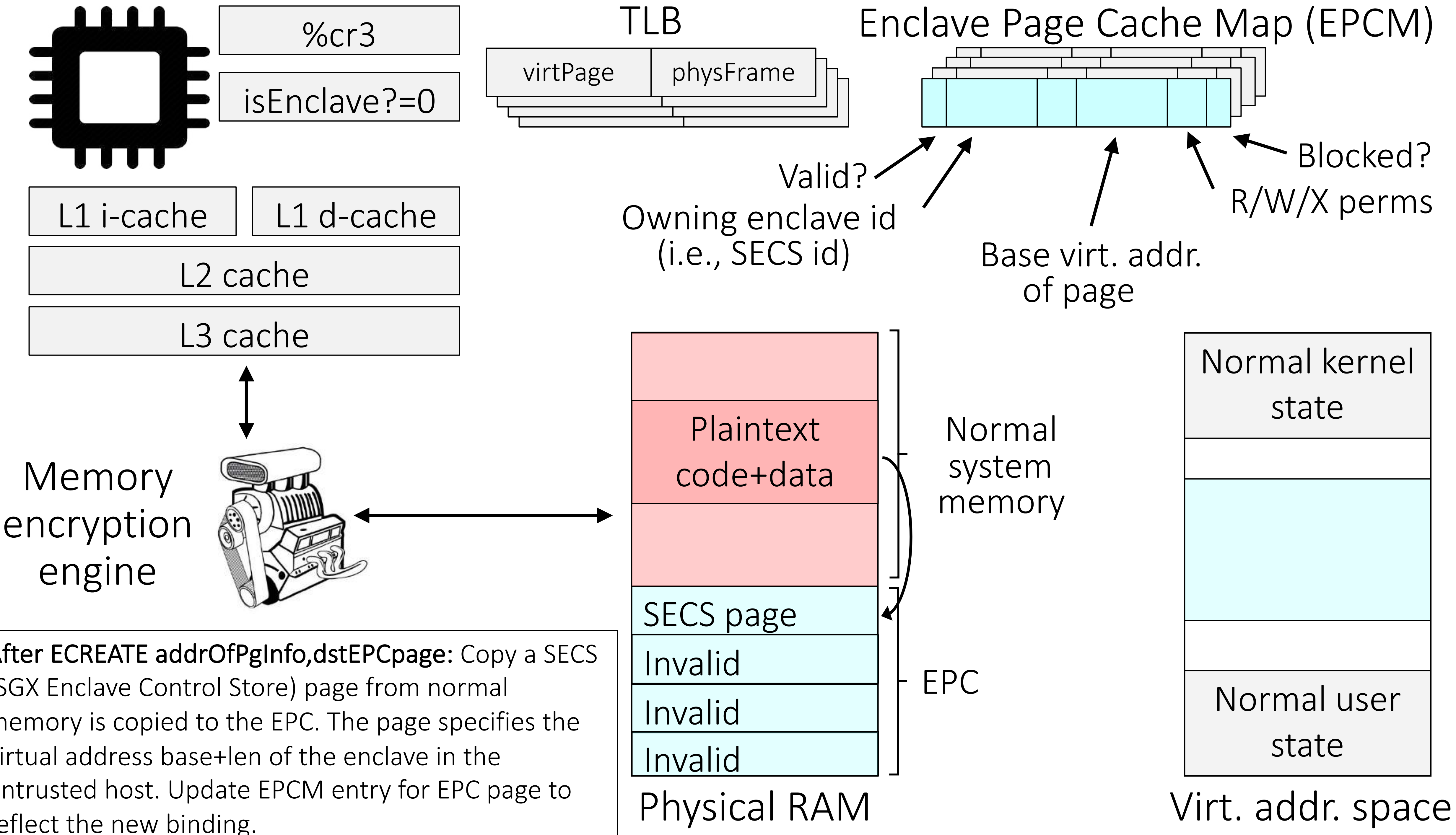
Physical RAM

Normal kernel state

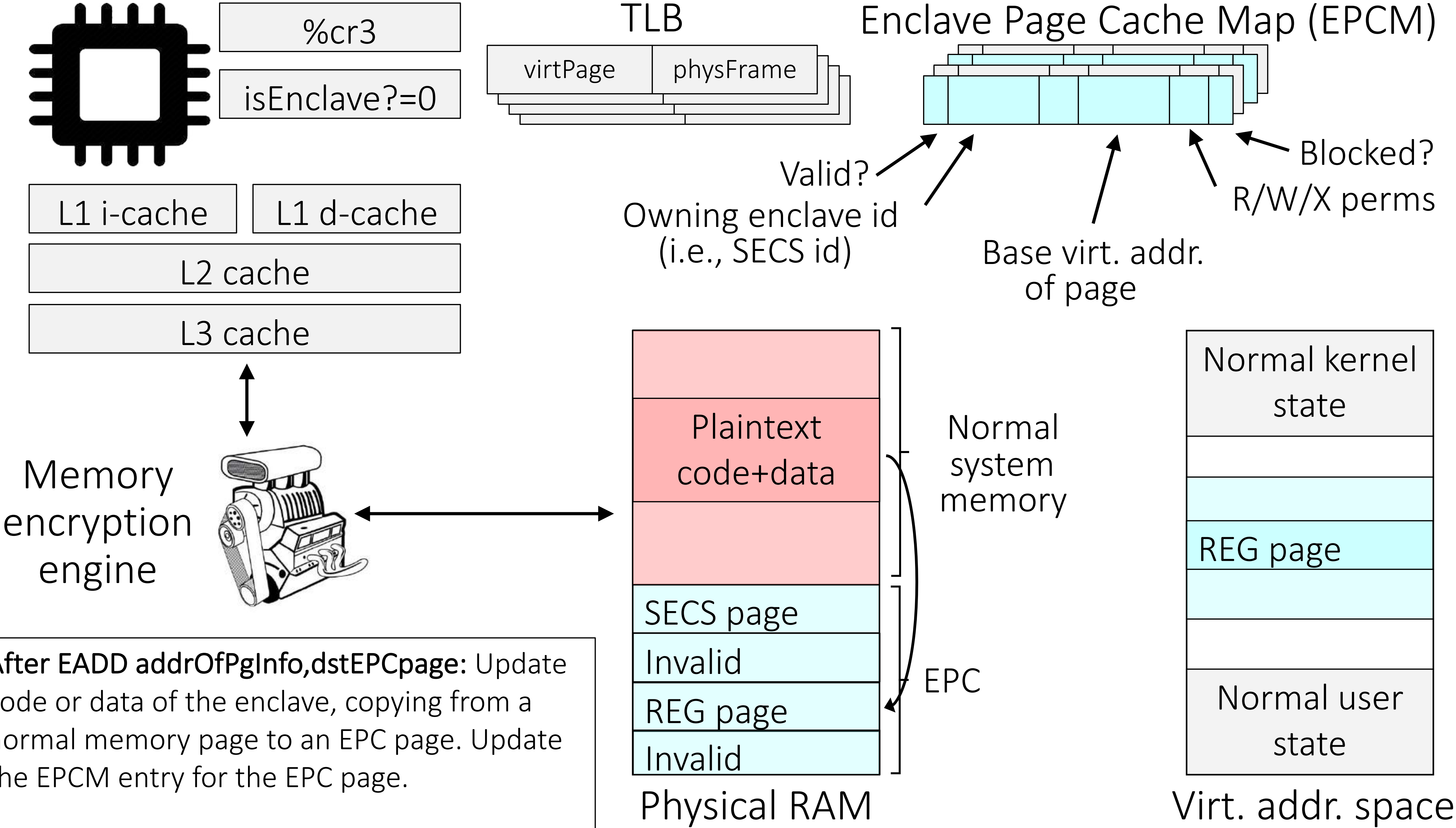
Normal user state

Virt. addr. space

Before ECREATE: The code and static data for the enclave sit in normal system memory.



After ECREATE addrOfPgInfo,dstEPCpage: Copy a SECS (SGX Enclave Control Store) page from normal memory is copied to the EPC. The page specifies the virtual address base+len of the enclave in the untrusted host. Update EPCM entry for EPC page to reflect the new binding.



%cr3

isEnclave?=0

TLB

virtPage

physFrame

Enclave Page Cache Map (EPCM)

Valid?

Owning enclave id (i.e., SECS id)

Base virt. addr. of page

R/W/X perms

Blocked?

L1 i-cache

L1 d-cache

L2 cache

L3 cache

Memory encryption engine

Plaintext code+data

Normal system memory

SECS page

Invalid

REG page

Invalid

EPC

Physical RAM

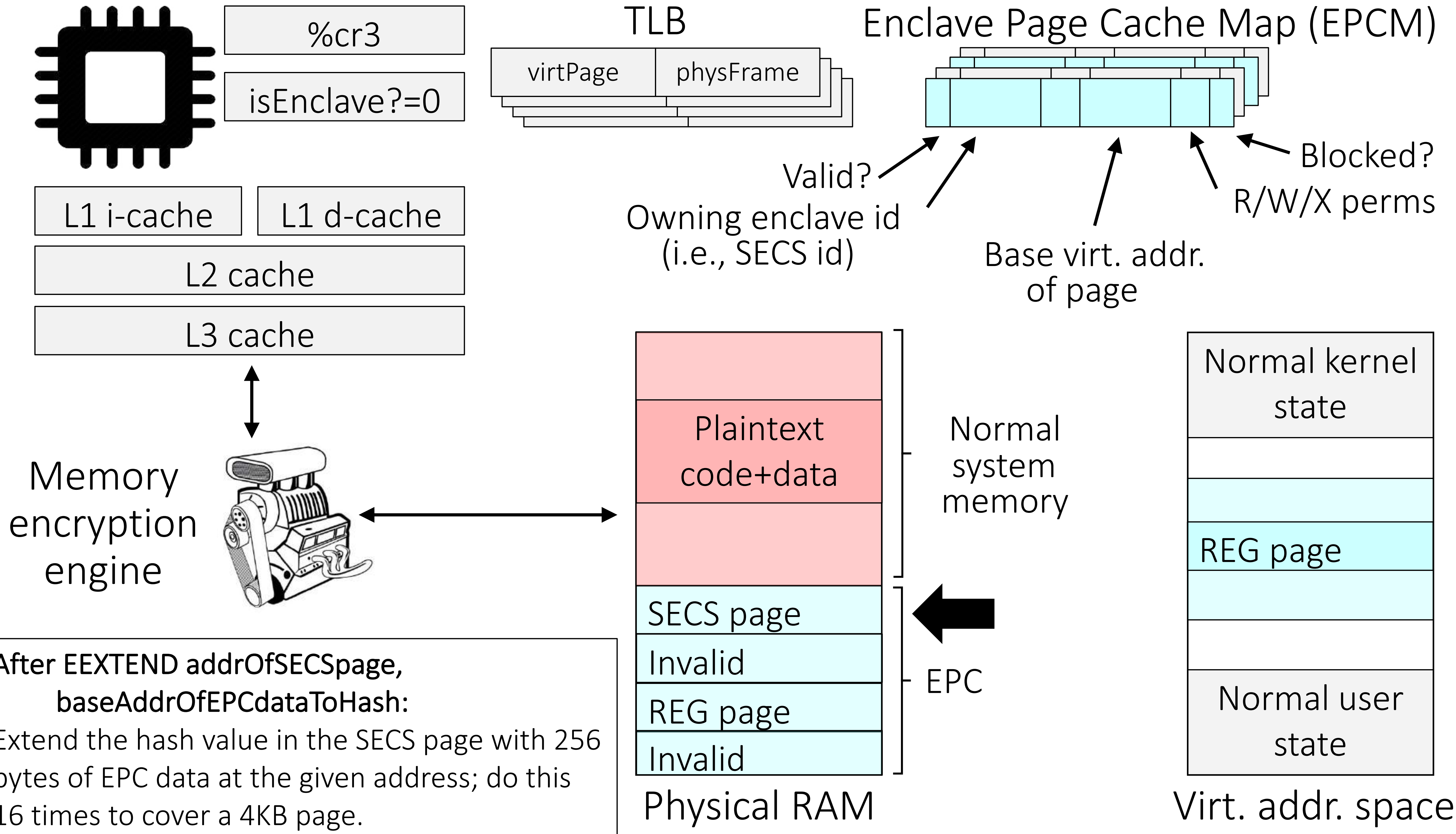
Normal kernel state

REG page

Normal user state

Virt. addr. space

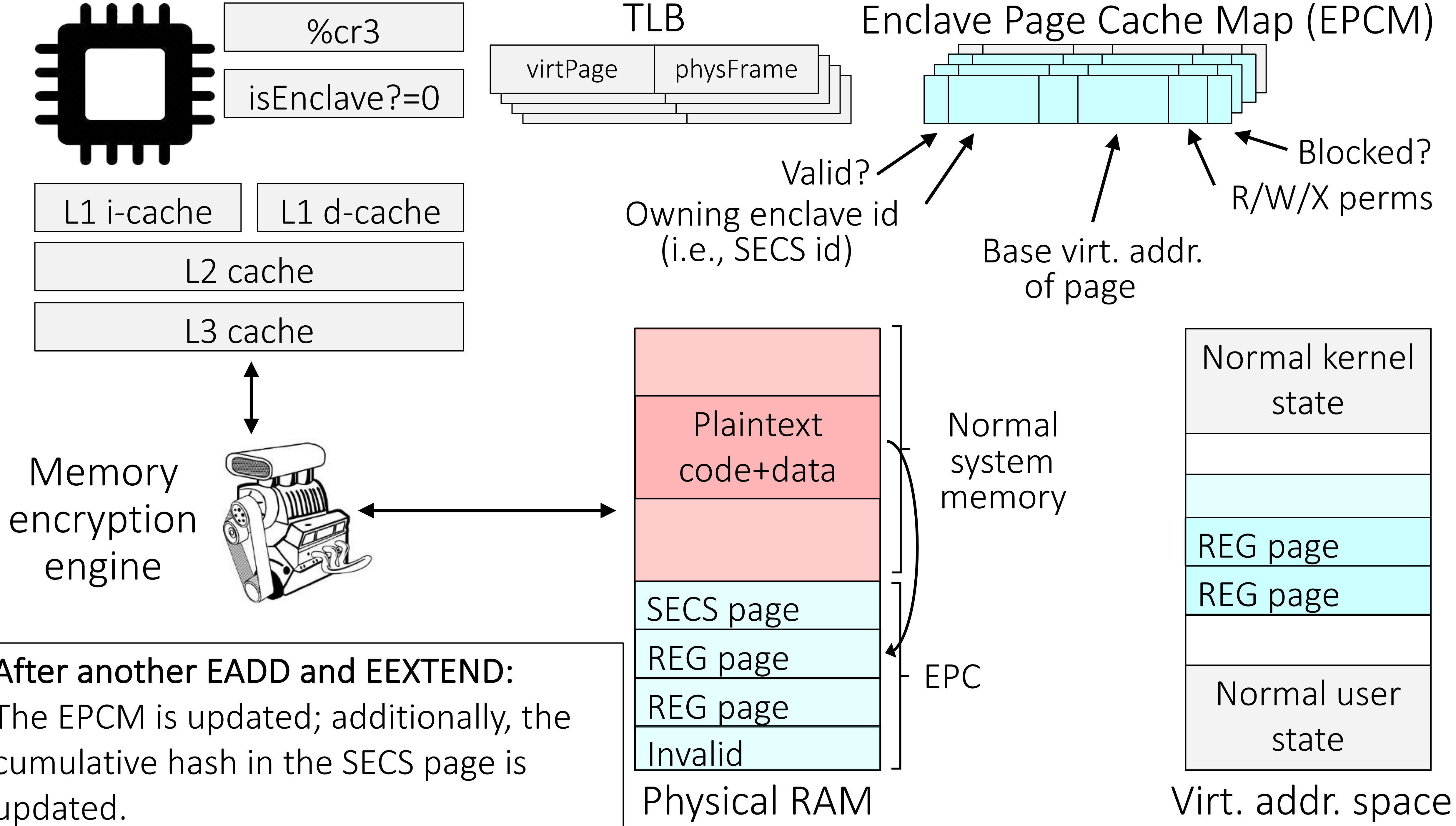
After EADD addrOfPgInfo,dstEPCpage: Update code or data of the enclave, copying from a normal memory page to an EPC page. Update the EPCM entry for the EPC page.



After EEXTEND addrOfSECSpage,
baseAddrOfEPCdataToHash:
Extend the hash value in the SECS page with 256 bytes of EPC data at the given address; do this 16 times to cover a 4KB page.

SGX: EEXTEND

- By invoking EEXTEND, we build up a cumulative hash that can later be used to attest the state used to initialize an enclave
- Why does EEXTEND only cover 256 bytes instead of a full 4KB page?
  - Hashing an entire page would take longer!
  - A long-running instruction either has to:
    - Disable interrupts (which means that the system becomes less response to IO events, timers, etc.)
    - Allow interrupts but fail with an error code if interrupted (meaning that the instruction must be retried by the program that invoked it)
- Note that ECREATE, EADD, EEXTEND, and EINIT can only be called by privileged code (i.e., Ring 0 code)
  - OS can launch denial-of-service by refusing to load an enclave, or by loading the enclave improperly (e.g., omitting an EEXTEND)



%cr3

isEnclave?=0

L1 i-cache

L1 d-cache

L2 cache

L3 cache

Memory encryption engine

TLB

virtPage

physFrame

Enclave Page Cache Map (EPCM)

Valid?

Owning enclave id (i.e., SECS id)

Base virt. addr. of page

R/W/X perms

Blocked?

Plaintext code+data

SECS page

REG page

REG page

Invalid

Normal system memory

EPC

Physical RAM

Normal kernel state

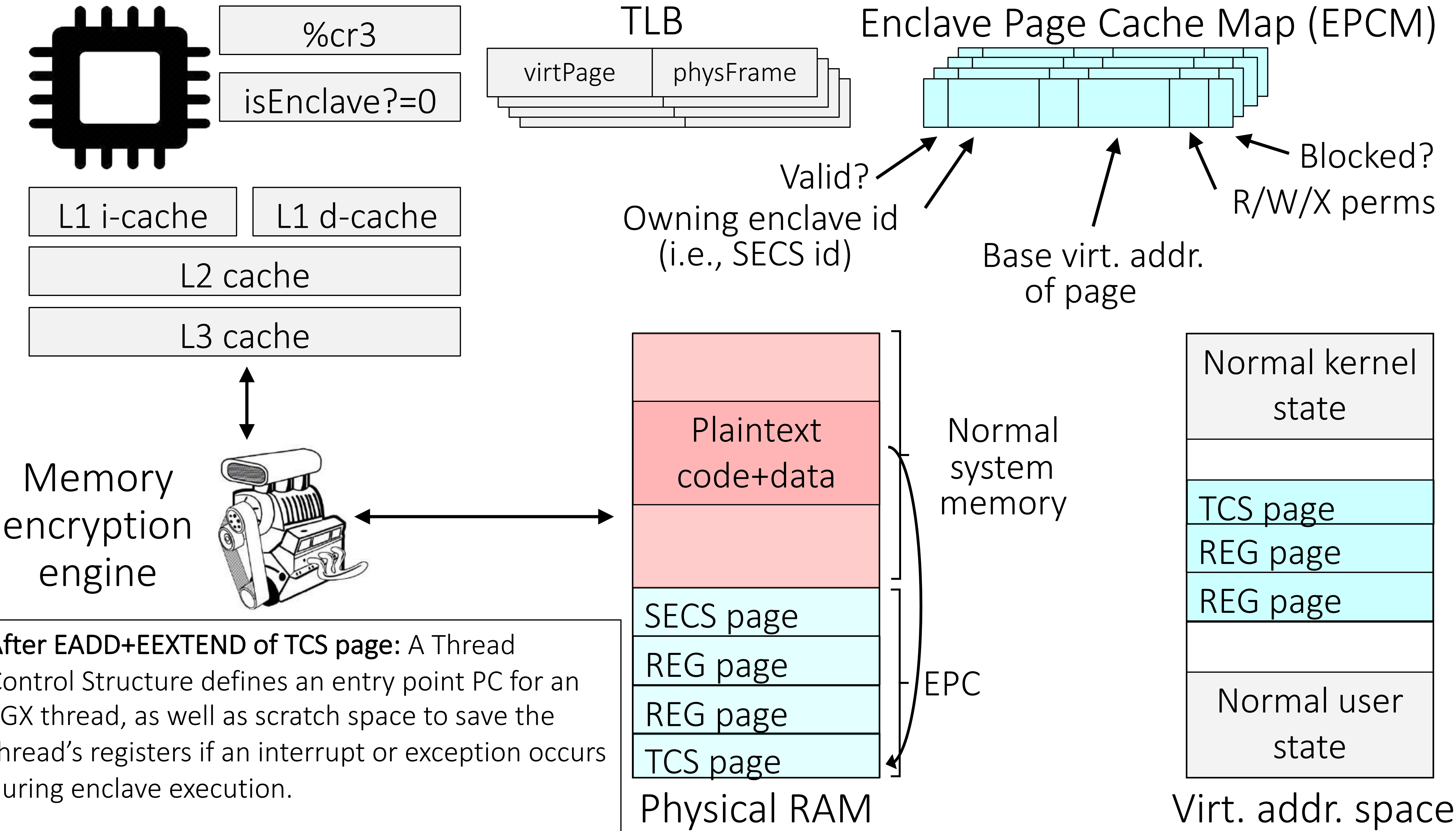
REG page

REG page

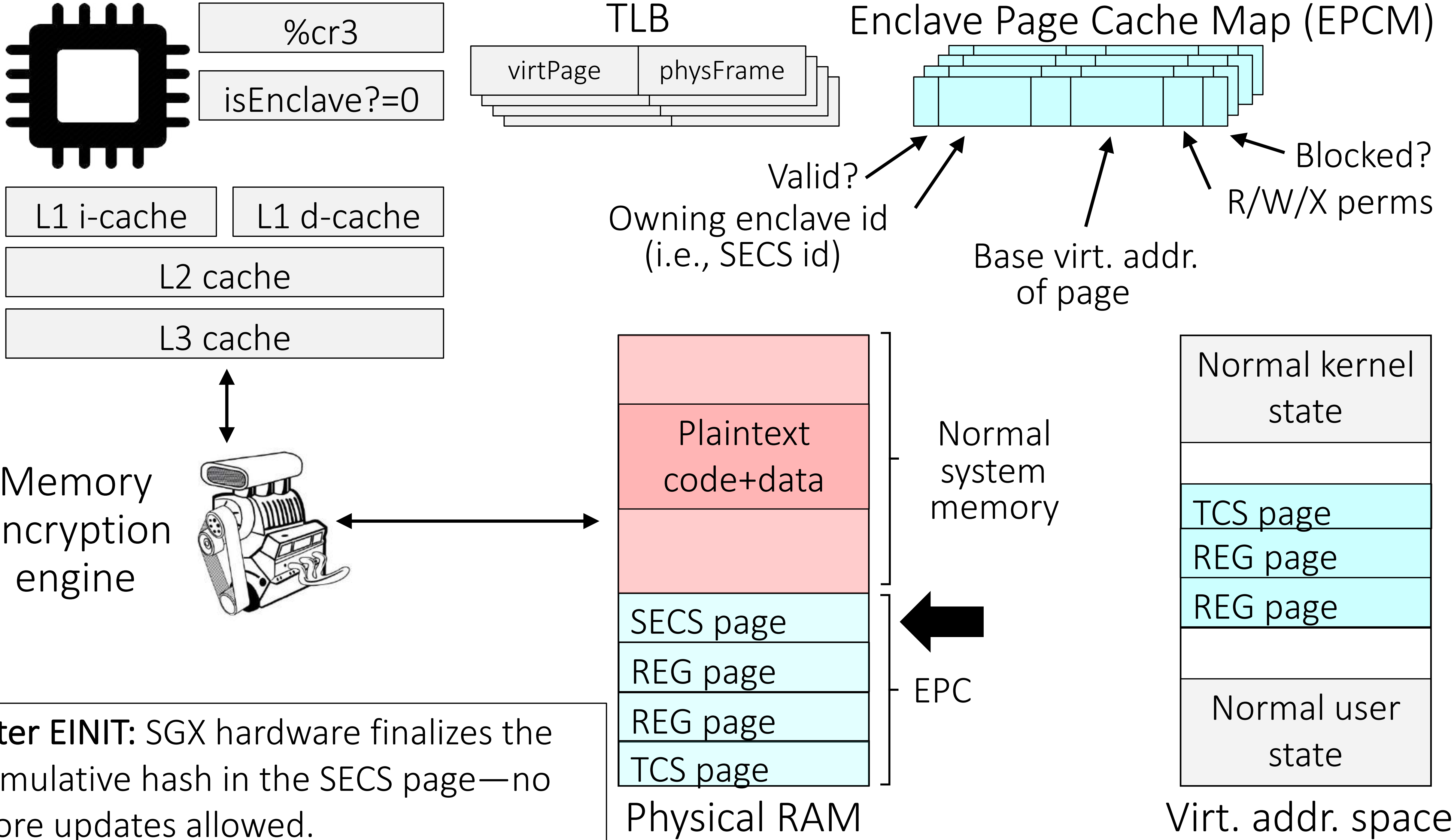
Normal user state

Virt. addr. space

After another EADD and EEXTEND:
The EPCM is updated; additionally, the cumulative hash in the SECS page is updated.



After EADD+EEXTEND of TCS page: A Thread Control Structure defines an entry point PC for an SGX thread, as well as scratch space to save the thread’s registers if an interrupt or exception occurs during enclave execution.



After EINIT: SGX hardware finalizes the cumulative hash in the SECS page—no more updates allowed.

EENTER and EEXIT

Neither instruction **changes `%cr3`!**

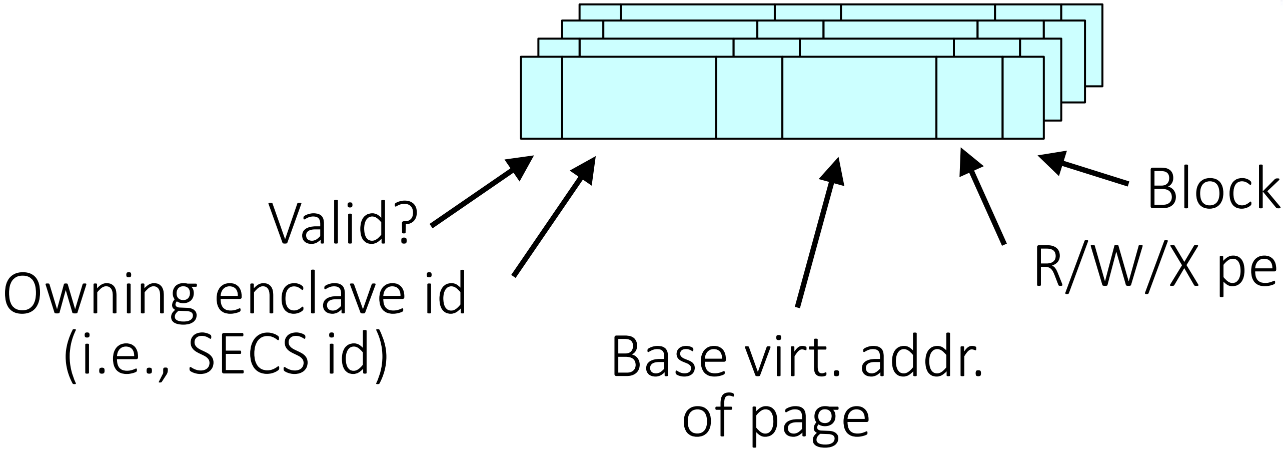
- `EENTER addrOfTCS, AEP`
  - Check `TCS.isBusy` to ensure that the untrusted host is not currently executing this SGX thread
  - Flush TLB entries
  - Flip `isEnclave` bit to 1
  - Save `%rip`, `%rsp`, and `%rbp` to scratch area inside the enclave's address space
  - Save `AEP` ("Asynchronous Exit Pointer" that points to a location in the untrusted host) to scratch area inside the enclave's address space
  - Write the address of the instruction after `EENTER` to `%rcx`; this value should be saved by the enclave code
  - Jump to the enclave start address described by TCS
- `EEXIT jmpTarget`
  - Flip `isEnclave` bit to 0
  - Flush TLB entries
  - Clear `TCS.isBusy`
  - Jump to `jmpTarget` in the untrusted host; should be address saved by `EENTER`

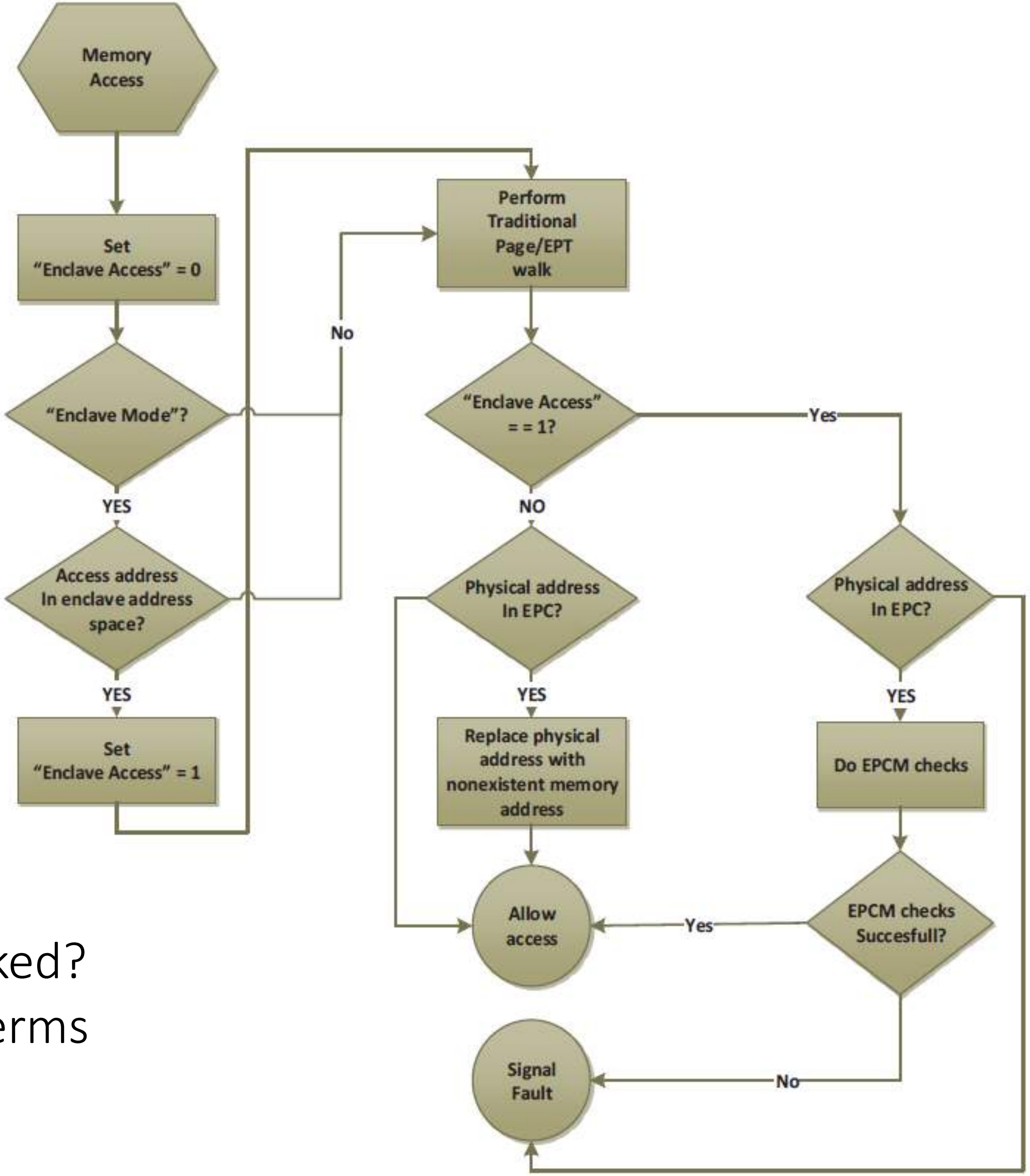
EENTER only callable by Ring 3+isEnclave=0
EEXIT only callable by Ring 3+isEnclave=1

Memory Access Checks in SGX

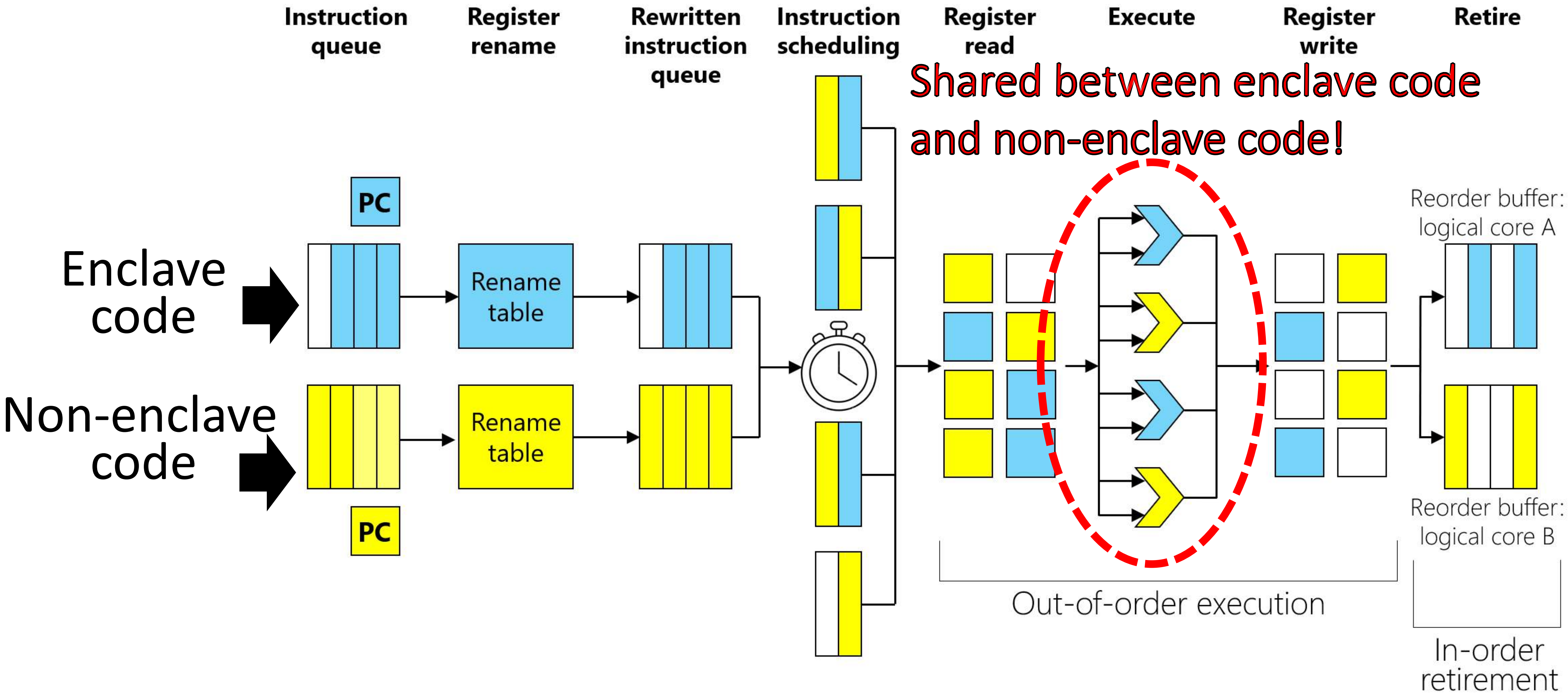
- SGX adds extra hardware to the memory controller to ensure that EPC pages can only be accessed by enclave code
- Extra hardware relies on EPCM

Enclave Page Cache Map (EPCM)

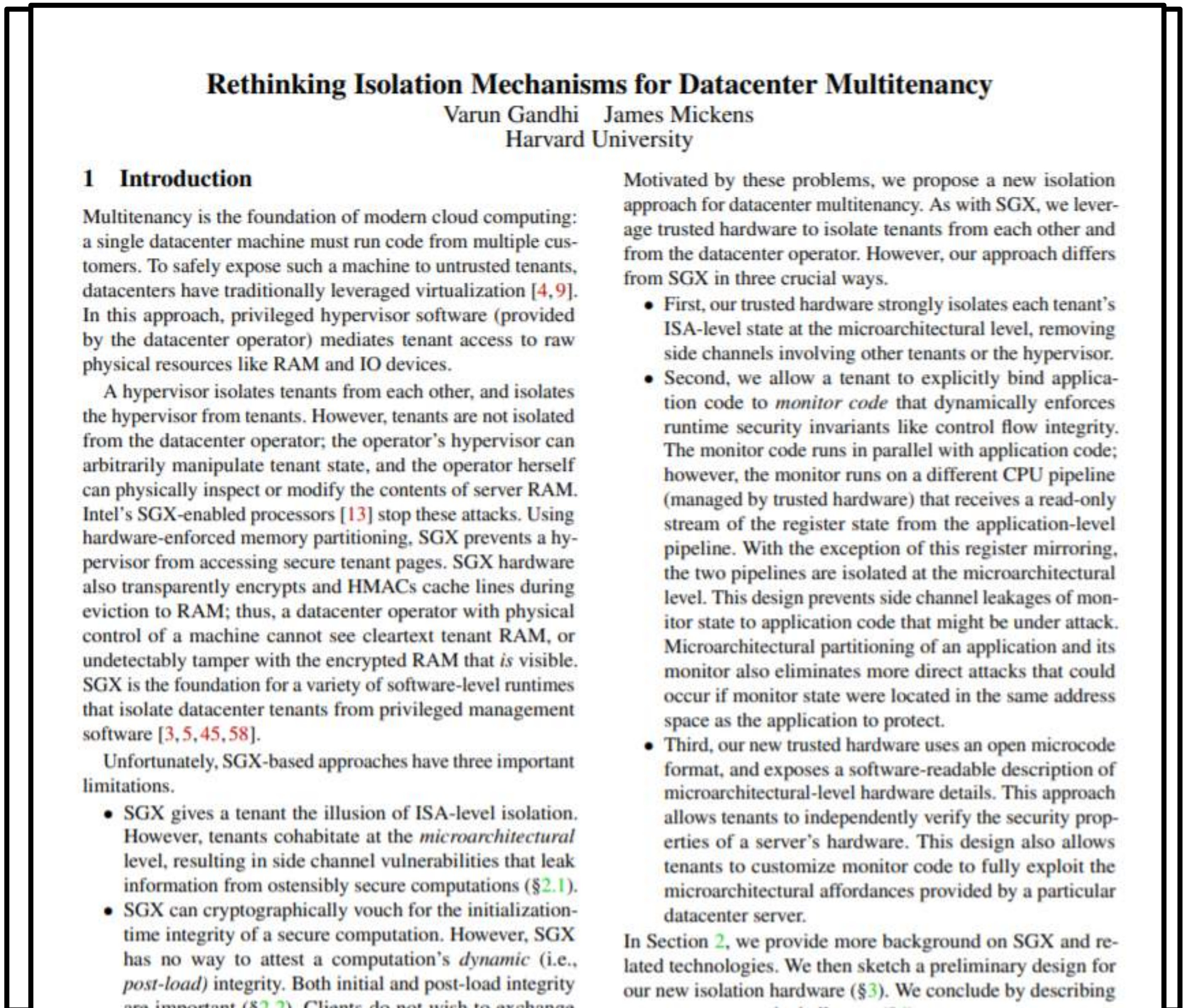




SGX is Vulnerable to Hyperthreading Side Channels!



SGX is Vulnerable to Cache Side Channels!



Rethinking Isolation Mechanisms for Datacenter Multitenancy

Varun Gandhi James Mickens
Harvard University

1 Introduction

Multitenancy is the foundation of modern cloud computing: a single datacenter machine must run code from multiple customers. To safely expose such a machine to untrusted tenants, datacenters have traditionally leveraged virtualization [4,9]. In this approach, privileged hypervisor software (provided by the datacenter operator) mediates tenant access to raw physical resources like RAM and IO devices.

A hypervisor isolates tenants from each other, and isolates the hypervisor from tenants. However, tenants are not isolated from the datacenter operator; the operator's hypervisor can arbitrarily manipulate tenant state, and the operator herself can physically inspect or modify the contents of server RAM. Intel's SGX-enabled processors [13] stop these attacks. Using hardware-enforced memory partitioning, SGX prevents a hypervisor from accessing secure tenant pages. SGX hardware also transparently encrypts and HMACs cache lines during eviction to RAM; thus, a datacenter operator with physical control of a machine cannot see cleartext tenant RAM, or undetectably tamper with the encrypted RAM that *is* visible. SGX is the foundation for a variety of software-level runtimes that isolate datacenter tenants from privileged management software [3,5,45,58].

Unfortunately, SGX-based approaches have three important limitations.

- SGX gives a tenant the illusion of ISA-level isolation. However, tenants cohabitate at the *microarchitectural* level, resulting in side channel vulnerabilities that leak information from ostensibly secure computations (§2.1).
- SGX can cryptographically vouch for the initialization-time integrity of a secure computation. However, SGX has no way to attest a computation's *dynamic* (i.e., *post-load)* integrity. Both initial and post-load integrity are important (§2.2). Clients do not wish to exchange

Motivated by these problems, we propose a new isolation approach for datacenter multitenancy. As with SGX, we leverage trusted hardware to isolate tenants from each other and from the datacenter operator. However, our approach differs from SGX in three crucial ways.

- First, our trusted hardware strongly isolates each tenant's ISA-level state at the microarchitectural level, removing side channels involving other tenants or the hypervisor.
- Second, we allow a tenant to explicitly bind application code to *monitor code* that dynamically enforces runtime security invariants like control flow integrity. The monitor code runs in parallel with application code; however, the monitor runs on a different CPU pipeline (managed by trusted hardware) that receives a read-only stream of the register state from the application-level pipeline. With the exception of this register mirroring, the two pipelines are isolated at the microarchitectural level. This design prevents side channel leakages of monitor state to application code that might be under attack. Microarchitectural partitioning of an application and its monitor also eliminates more direct attacks that could occur if monitor state were located in the same address space as the application to protect.
- Third, our new trusted hardware uses an open microcode format, and exposes a software-readable description of microarchitectural-level hardware details. This approach allows tenants to independently verify the security properties of a server's hardware. This design also allows tenants to customize monitor code to fully exploit the microarchitectural affordances provided by a particular datacenter server.

In Section 2, we provide more background on SGX and related technologies. We then sketch a preliminary design for our new isolation hardware (§3). We conclude by describing

- The enclave and the untrusted host share the same L1 cache!
- Untrusted host can:
  - Load the enclave and evict an oracle array's cache lines
  - Wait for the enclave to bring a cleartext secret value into the L1 cache, then do:

    ```
    uint8_t v = *secret_ptr;
    uint64_t o = oracle[v*4096];
    ```
  - The read will fail at the ISA level, but at the microarchitectural level, a speculative load will read a cache line from the oracle array!
  - The untrusted host can then time how long it takes to read each cache line in the oracle array, determining the secret byte!

What Did We Learn?

Day I: Introduction to Control Flow Integrity

- Overview of virtual address spaces
- Attacks: Buffer overflows, return-to-libc
- Defenses: NX bits, Intel shadow stacks, ASLR

Day II: Advanced Attacks and Defenses

- Attacks: Memory vulnerabilities via type confusion, ROP attacks, JOP attacks
- Defenses: Compiler-enforced CFI, Intel indirect branch tracing
- SGX: Design and vulnerabilities

Software-level Attacks on Architectural and Microarchitectural State

ACACES SUMMER SCHOOL

July 9th - 10th, 2020

~~Fiuggi, Rome~~ Zoom World

James Mickens

Harvard University