Hacking in Darkness: Return-oriented Programming against Secure Enclaves

Jaehyuk Lee° Jinsoo Jang° Yeongjin Jang⋆
Nohyun Kwak° Yeseul Choi° Changho Choi°

Taesoo Kim⋆ Marcus Peinado† Brent Byunghoon Kang °

Microsoft Research †
Georgia Institute of Technology ⋆
KAIST °

Big Idea: Cloud Computing

Big Hurdle: “Security”

62 Percent of Companies Store Sensitive Customer Data in the Public Cloud

And almost 40 percent of cloud services are commissioned without the involvement of IT, a recent survey found.

By **Jeff Goldman** | Posted February 21, 2017 Share G+1

Network Worl NEWS

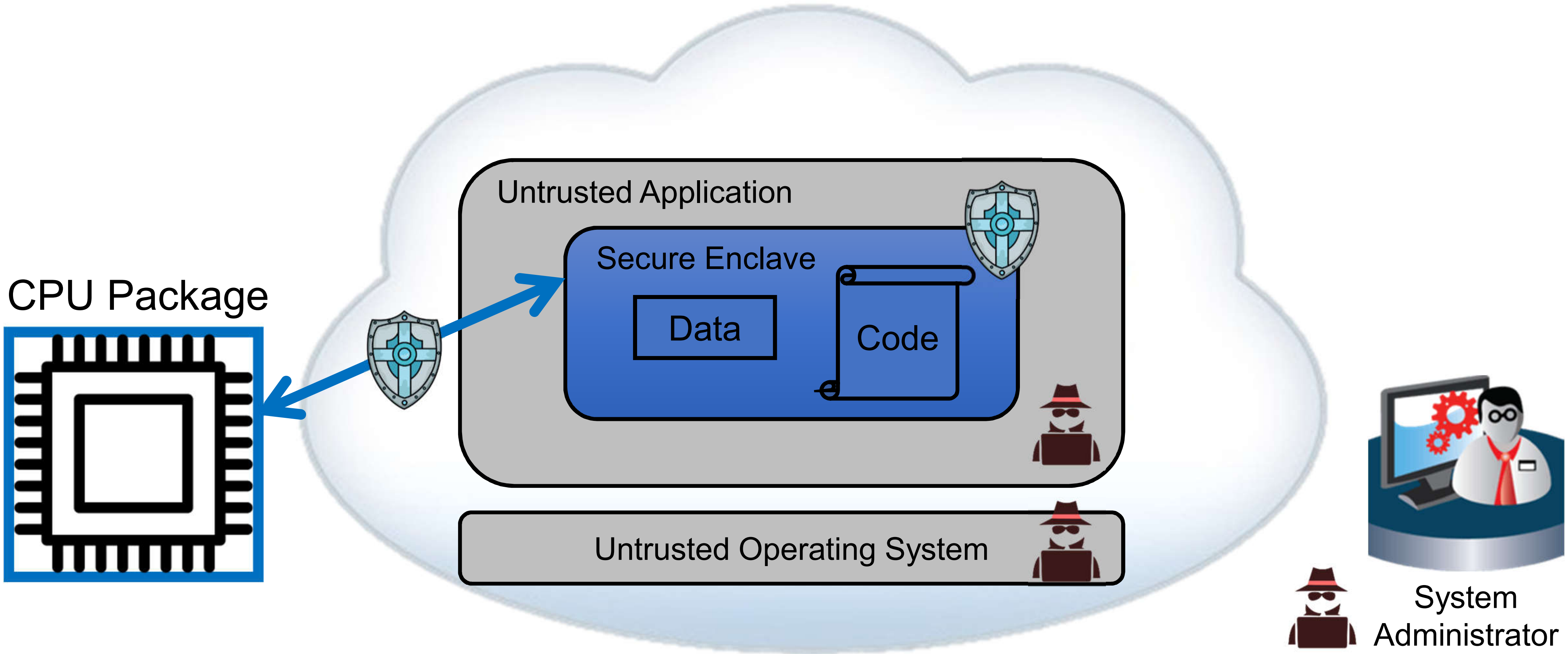
IT leaders say it's hard to keep the cloud safe

Shadow IT causing cloud trouble by illicitly working behind the scenes

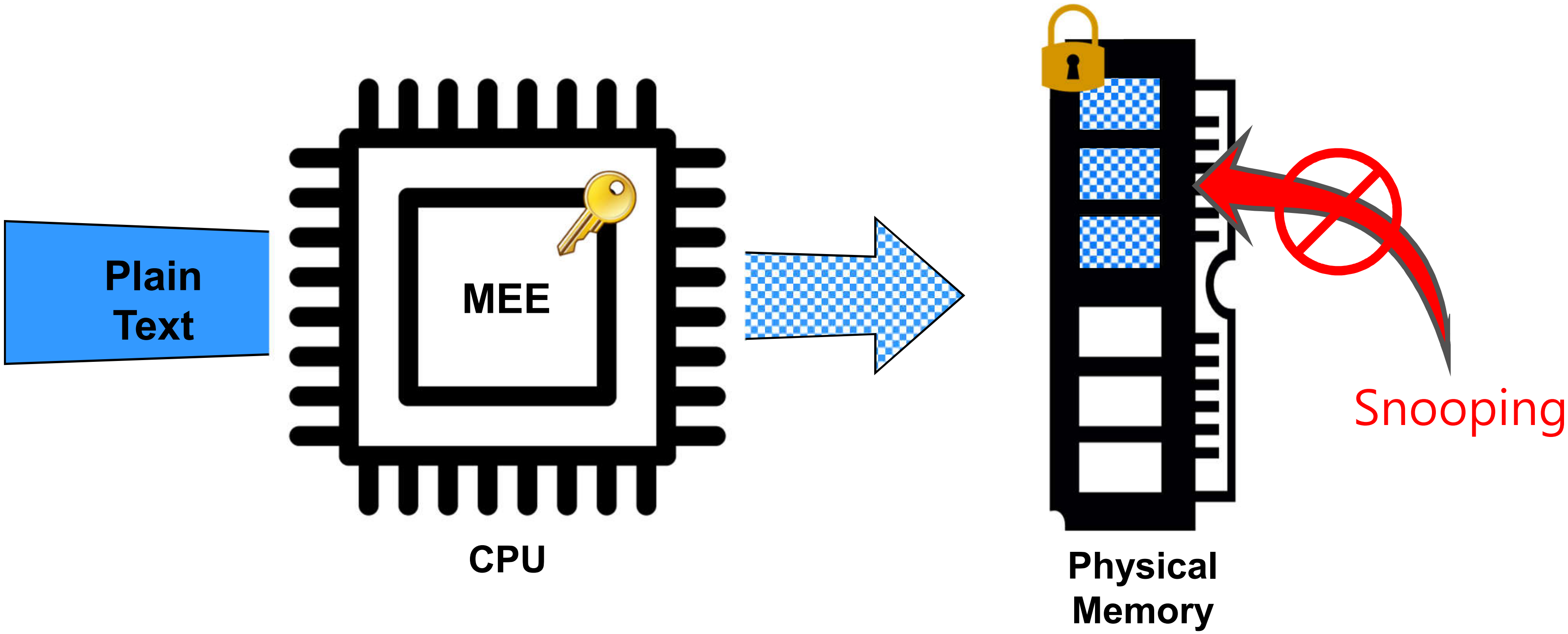
By Sharon Gaudin | Follow

Senior Writer, Computerworld | FEB 15, 2017 12:17 PM PT

SGX protects enclave from outside

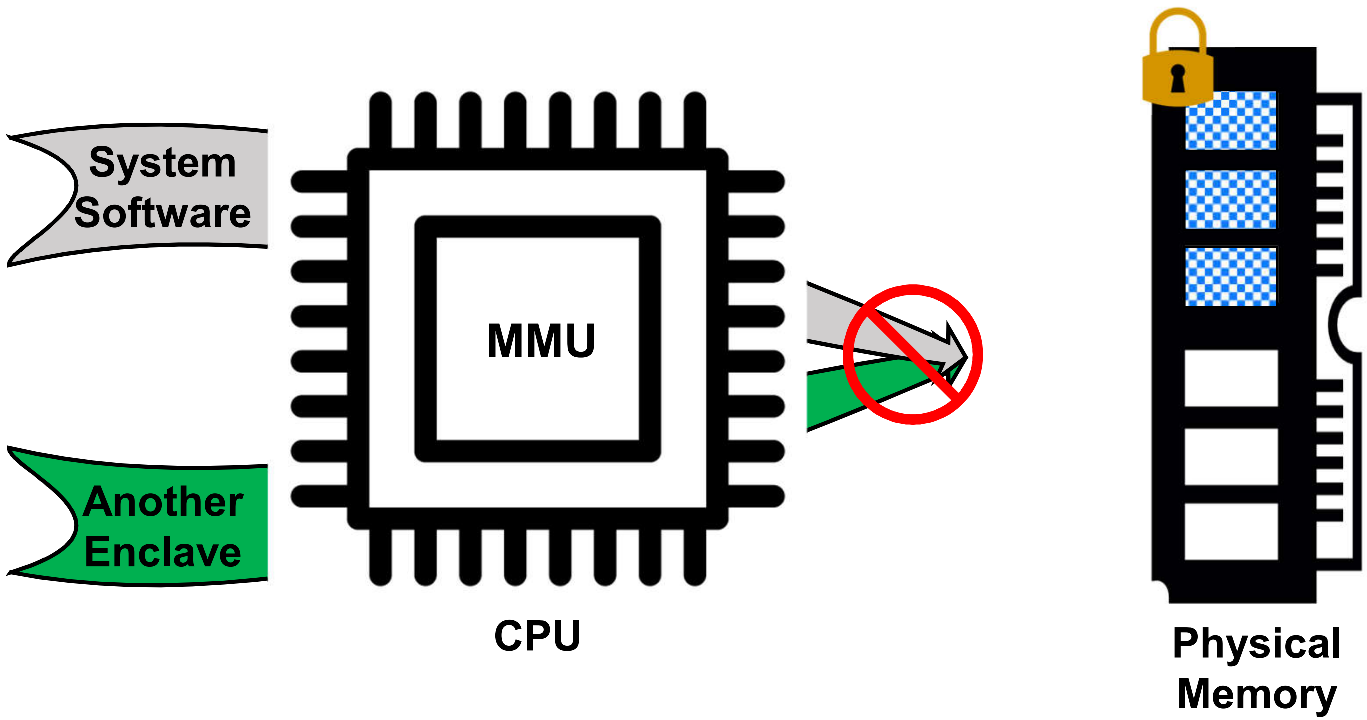


Memory encryption in SGX



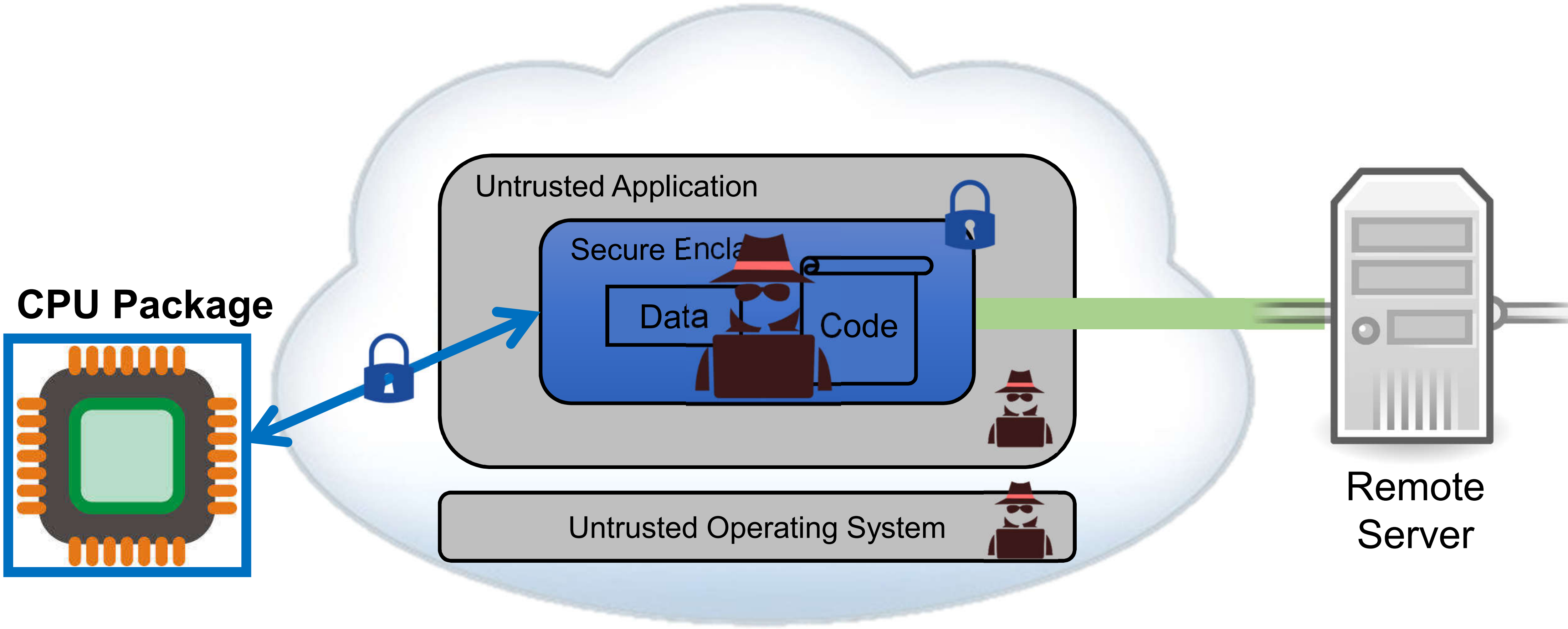
- MEE encrypts all contents of the enclave memory
- Decrypts using the hardware provided key
- Cold boot attacks & Snooping is impossible

Memory protection in SGX



- MMU keeps system software from accessing Enclaves
- Allows the accessibility of the enclave to its own contents

Now, can we say all software is secure ?



Software vulnerabilities are prevalent

OSS-Fuzz: Five months later, and rewarding projects

Monday, May 8, 2017

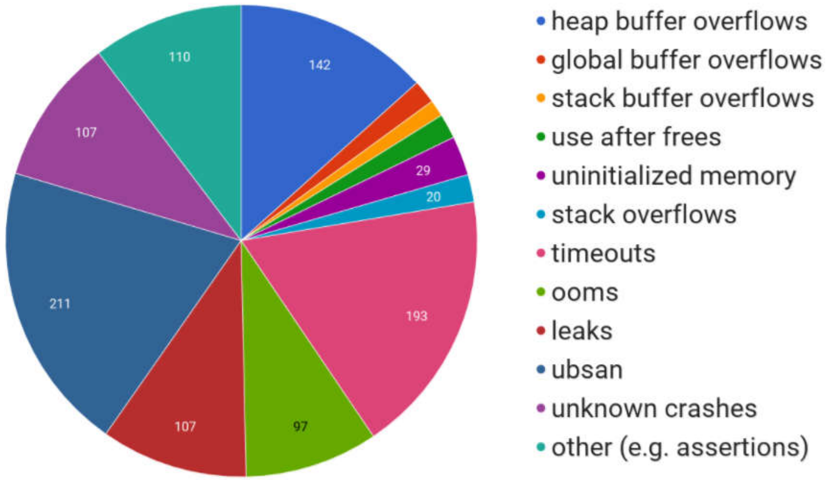
Five months ago, we announced OSS-Fuzz, Google's effort to help make open source software more secure and stable. Since then, our robot army has been working hard at fuzzing, processing 10 trillion test inputs a day. Thanks to the efforts of the open source community who have integrated a total of 47 projects, we've found over 1,000 bugs (264 of which are potential security vulnerabilities).

CVE

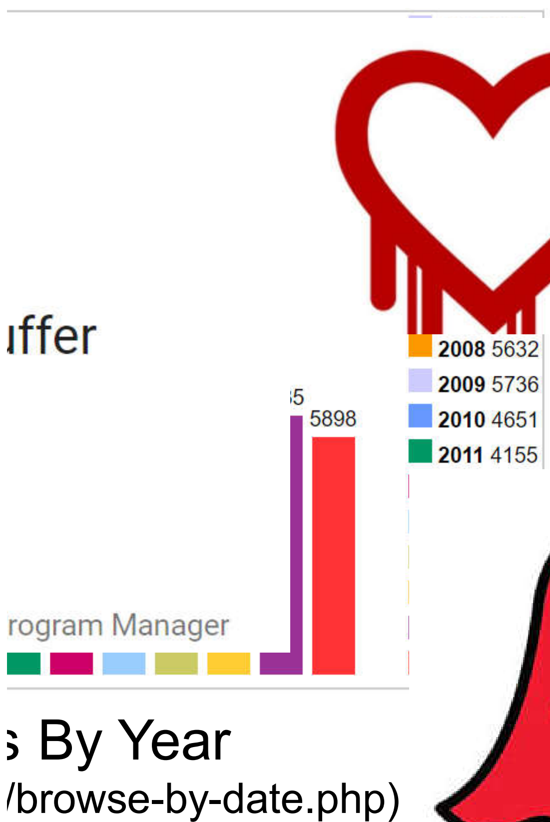
over

Februa

Posted

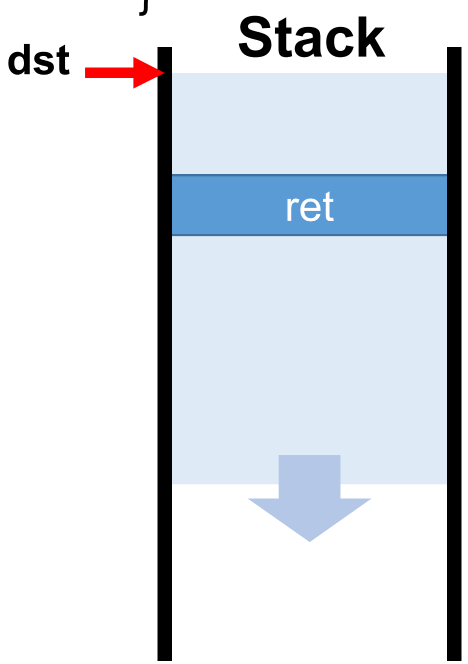


Breakdown of the types of bugs we're finding.



Return-oriented programming (ROP) attack

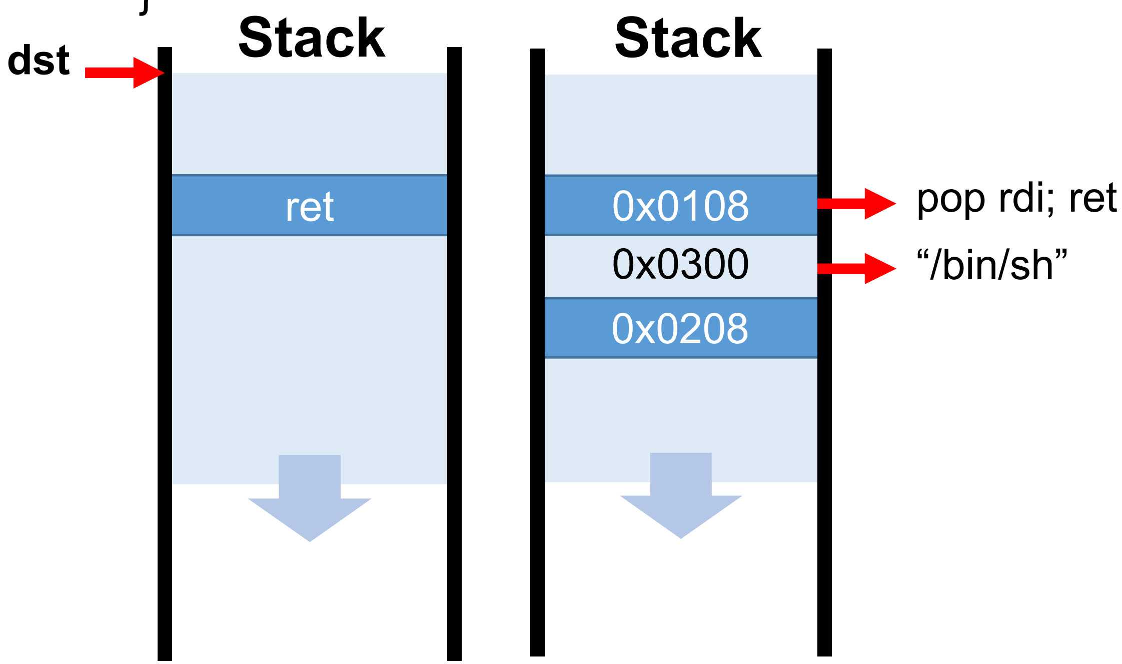
```
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```



Return-oriented programming (ROP) attack

```
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```

e.g., system("/bin/sh")



Return-oriented programming (ROP) attack

```
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```

e.g., system("/bin/sh")

Stack Stack Stack

dst

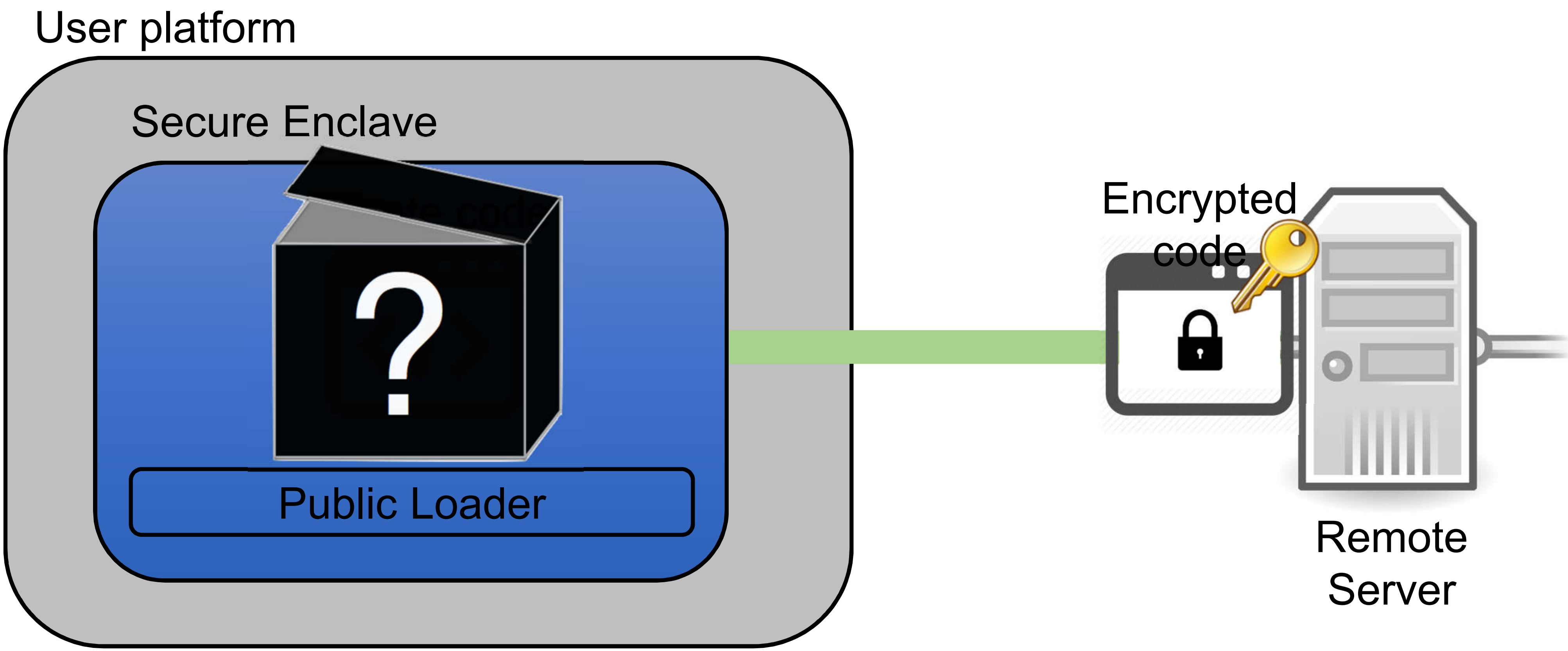
Assumption:
Addresses of the pop gadget & function are known (e.g., reverse engineering)

Deploying an encrypted binary in SGX

- Operating System loads the enclave pages to memory.
  - Malicious OS can see the content of enclave binaries.

- Software vendor can make use of full encryption over enclave binaries.
  - Prevent the reverse engineering.
  - VC3 first showed private code can be loaded to enclaves.

Deploying an encrypted binary in SGX



- Encryption over the binary to prevent reverse engineering

ROP inside an **enclave**

```
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```

Code is not visible
(i.e., loaded in an encrypted form)
- 0x100: ????
- 0x200: ????

For the enclave binaries
Addresses of the pop gadget & function
are unknown

Threat model of Dark-ROP

- **The attacker has full control of all software of the system**
  - including the operating system and the untrusted app.
- **The attacker can make the enclave program crash multiple times.**
  - Inspecting the program behavior from the crash.
- **The application is built with a standard compiler with Intel SDK**
  - (e.g. Visual Studio for SGX, or gcc)
- **Enclave application is distributed in an encrypted format**
  - All the runtime information of the enclaves are hidden

Contribution of Dark-ROP

- We devise a new way to launch a code-reuse attack against encrypted enclave binaries
  - Finding POP gadgets to control registers in enclaves
  - Finding memcpy function to copy data from enclaves
- The Dark-ROP attack can completely disarm the security guarantees of SGX
  - Decrypting and generating the correctly sealed data.
  - Bypassing local and remote attestation.

Dark ROP: ROP in darkness

- Step 1. Finding the locations of pop gadgets
  - Pop gadget: bunch of pops followed by ret instruction.
    - pop r??; ret
    - pop r??; pop r??; ret
  - Enabling load value into the registers in enclave context
- Step 2. Locating ENCLU + pop rax (i.e., EEXIT)
  - ENCLU instruction is used to
    - Decipher pop gadgets
    - Retrieve the hardware provided key for unsealing
    - Generate the malicious report data to bypass remote attestation

Dark ROP: ROP in darkness

- Step 3. Deciphering all pop gadgets
  - ENCLU instruction is used to decipher pop gadgets found at first step.
  - Discerning which gadget loads value to which register.
    - pop r??; ret -> pop rax; ret;

- Step 4. Locating memcpy()
  - Copying secret data from the enclaves
  - Injecting malicious data to the enclaves

Step 1. Looking for pop gadgets

Enclave Memory map

	Address	Access Permission	
ENCLAVE	0xF7500000 - 0xF752b000	r-x	Code
	……		
	0xF7741000 - 0xF7841000	rw-	Heap
	0xF7842000 - 0xF7882000	rw-	Stack

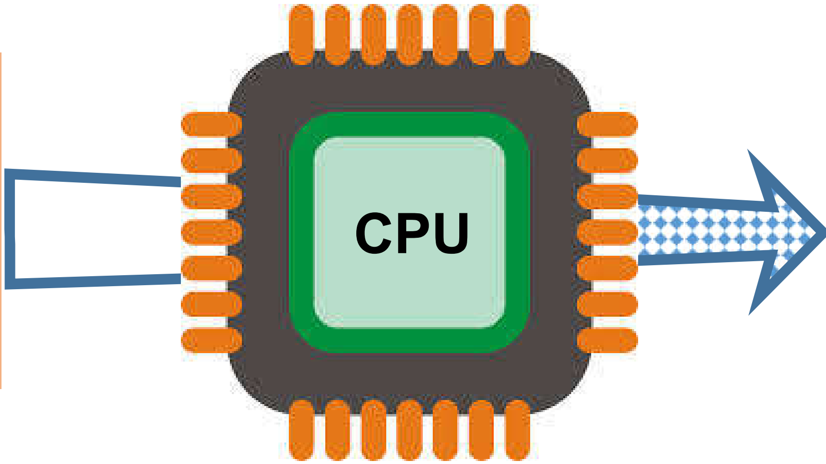
Attackers have a full control over the layout of the enclave

Step 1. Looking for pop gadgets

Asynchronous Enclave Exit (AEX)

Enclave Context

rax = 0x00000001
rdx = 0x00000002
….



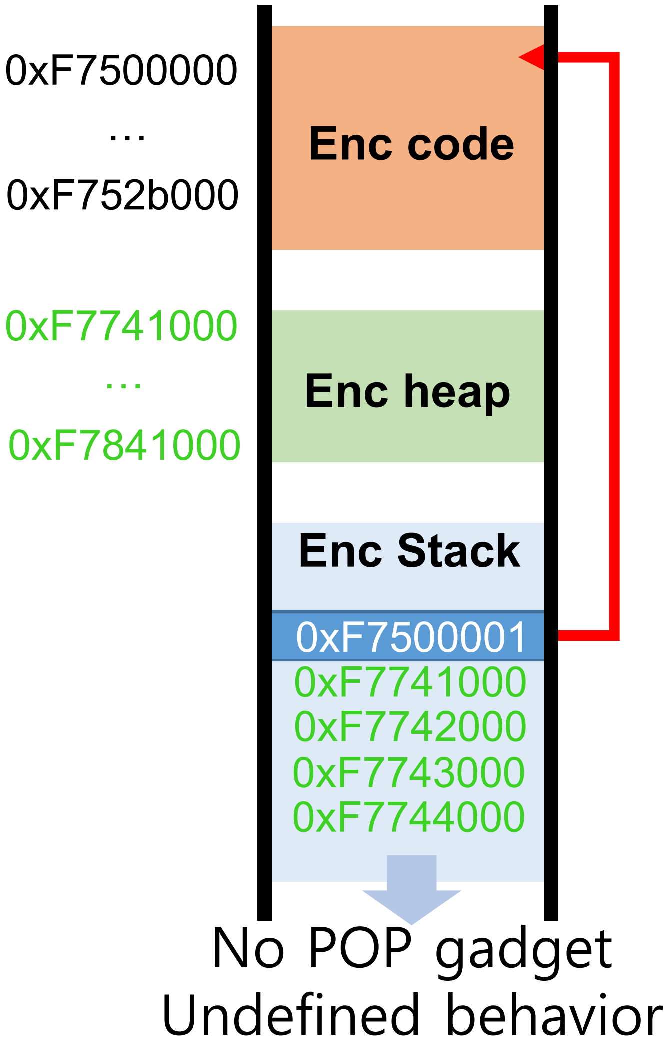
Exception Context

rax = 0x00000003
rdx = 0x00000000
….
cr2 = 0x7f181000

When Exception happens inside enclave

CPU fills synthetic state
But, we know which page incurs fault

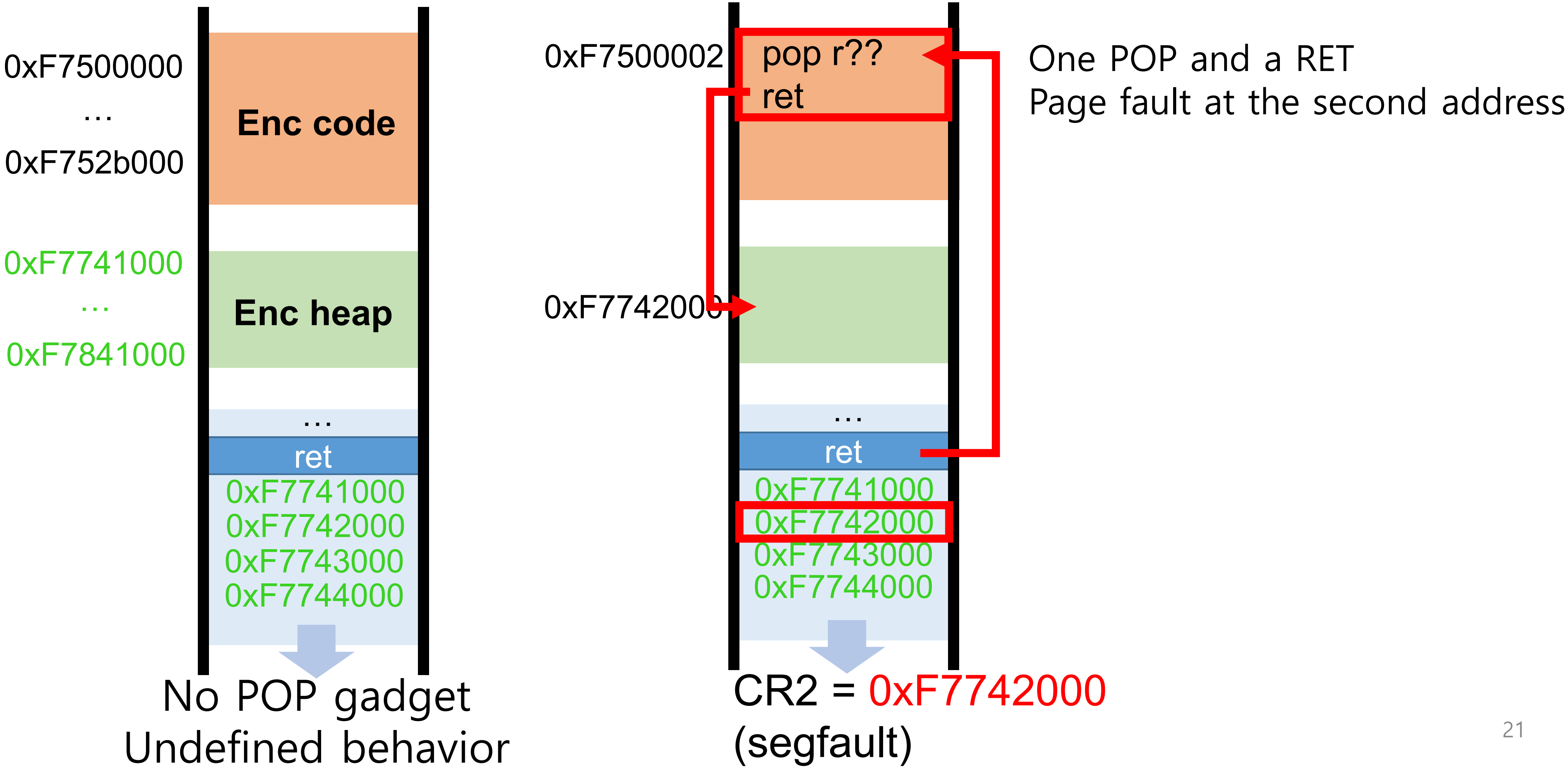
Step 1. Looking for pop gadgets



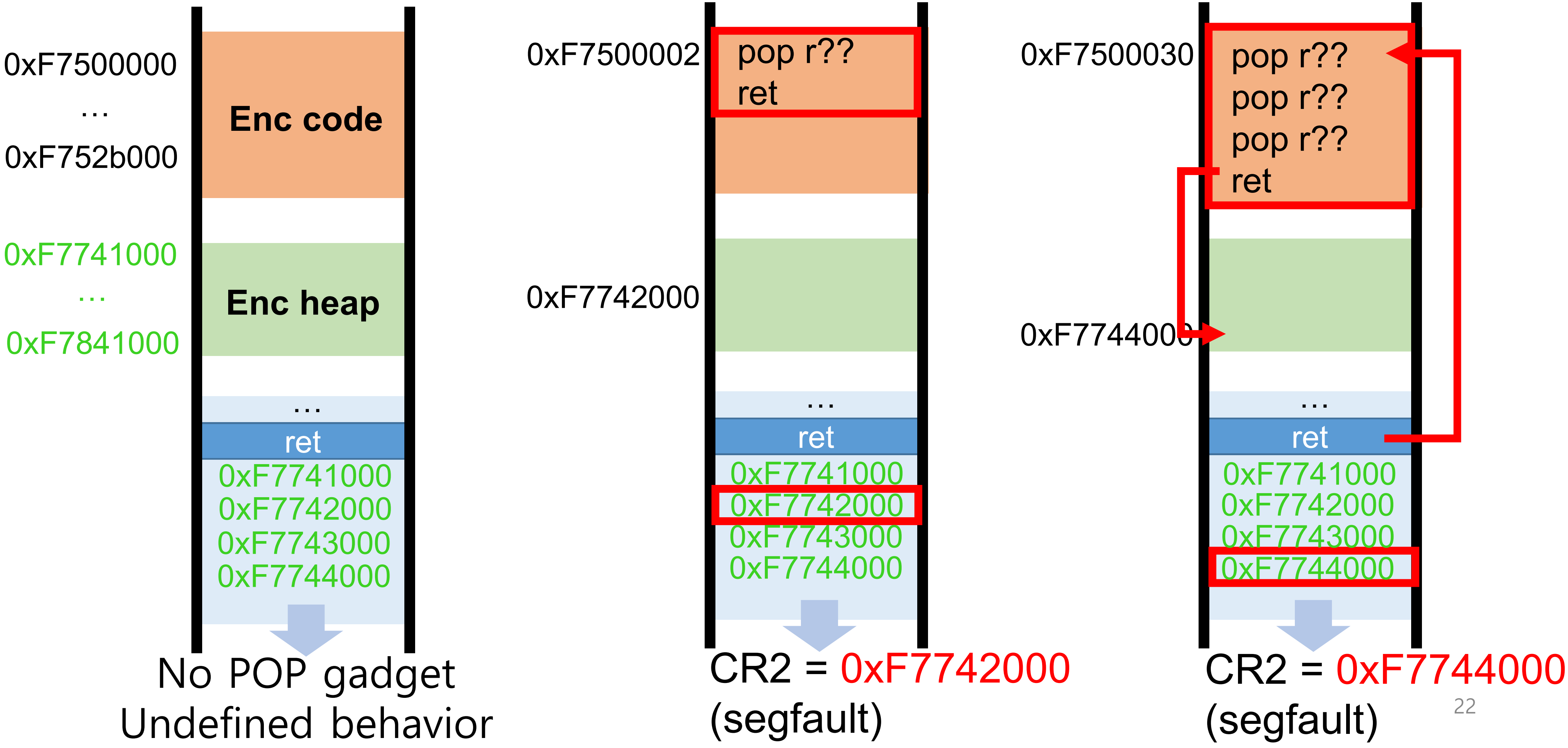
Key idea

- **Write addresses of non-executable pages on the stack**
- **RET to a non-executable address produces a page fault and an AEX**
  - **This is how we find RET instructions.**
- **The page incurring the fault is known (CR2 register)**
- **The faulting page tells us how many POPs happened before the RET**

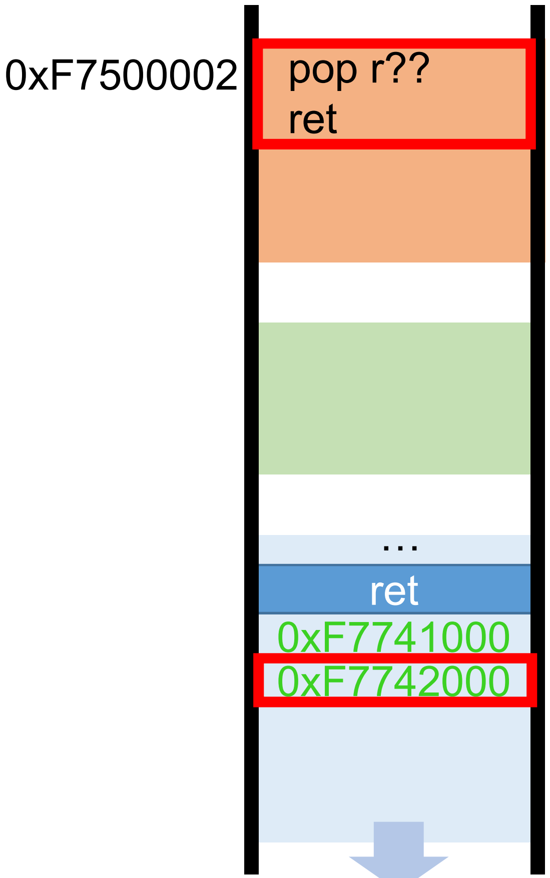
Step 1. Looking for pop gadgets



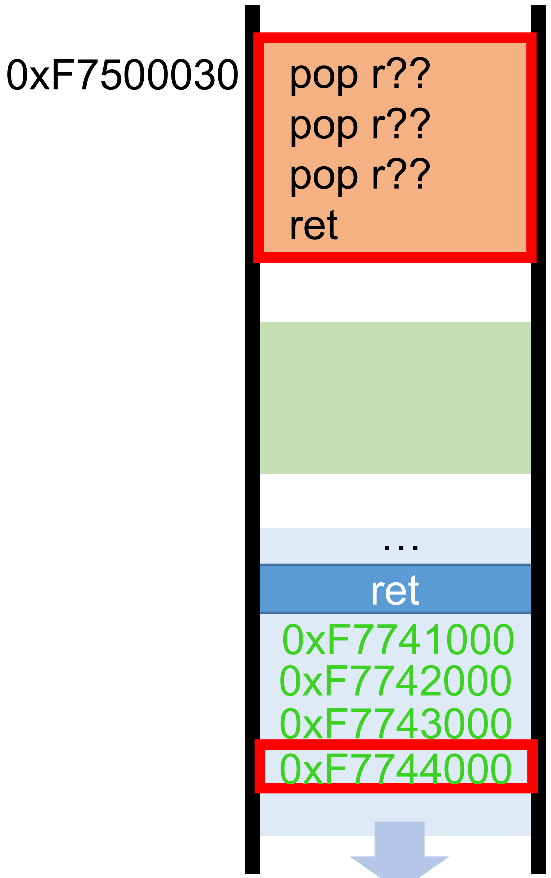
Step 1. Looking for pop gadgets



Step 1. Looking for pop gadgets



CR2 = 0xF7742000
(segfault)



CR2 = 0xF7744000
(segfault)

Search entire enclave code

→ Catalog of pop gadgets
(unknown args)

0xF7500002 → **pop** r??;
ret

0xF7500030 → **pop** r??;
pop r??;
pop r??;
ret

…

We still need to find the target registers

Catalog of pop gadgets
(unknown args)

0xF7500002 → **pop** r??;
ret

0xF7500030 → **pop** r??;
pop r??;
pop r??;
ret

…

Catalog of pop gadgets
(known args)

0xF7500002 → pop **rax**;
ret

0xF7500030 → pop **rbx**;
pop **rcx**;
pop **rdx**;
ret

…

Step 2. Looking for ENCLU: One opcode represents multiple functionalities

ENCLU

Rax value	Leaf function	Description
0x0	EREPORT	Create a cryptographic report
0x1	EGETKEY	Retrieve a cryptographic key
…		
0x4	EEXIT	Synchronously exit an enclave
…		
0x6	EMODPE	Extend an access permission of EPC

- ENCLU instruction handles all user level enclave operations.
- ENCLU behavior depends on RAX value.
- RAX = 4 -> Enclave exit.
- EEXIT does not erase enclave register values.

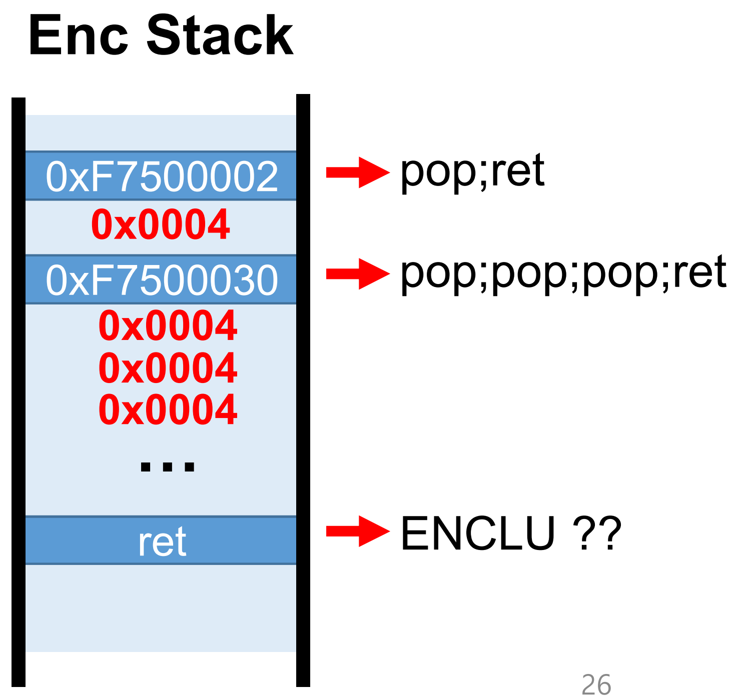
Step 2. Looking for ENCLU instruction

- It's "required" to have a ENCLU (to exit) for proper functioning.

- Chain multiple pop gadgets we found in step 1 with a probing address.

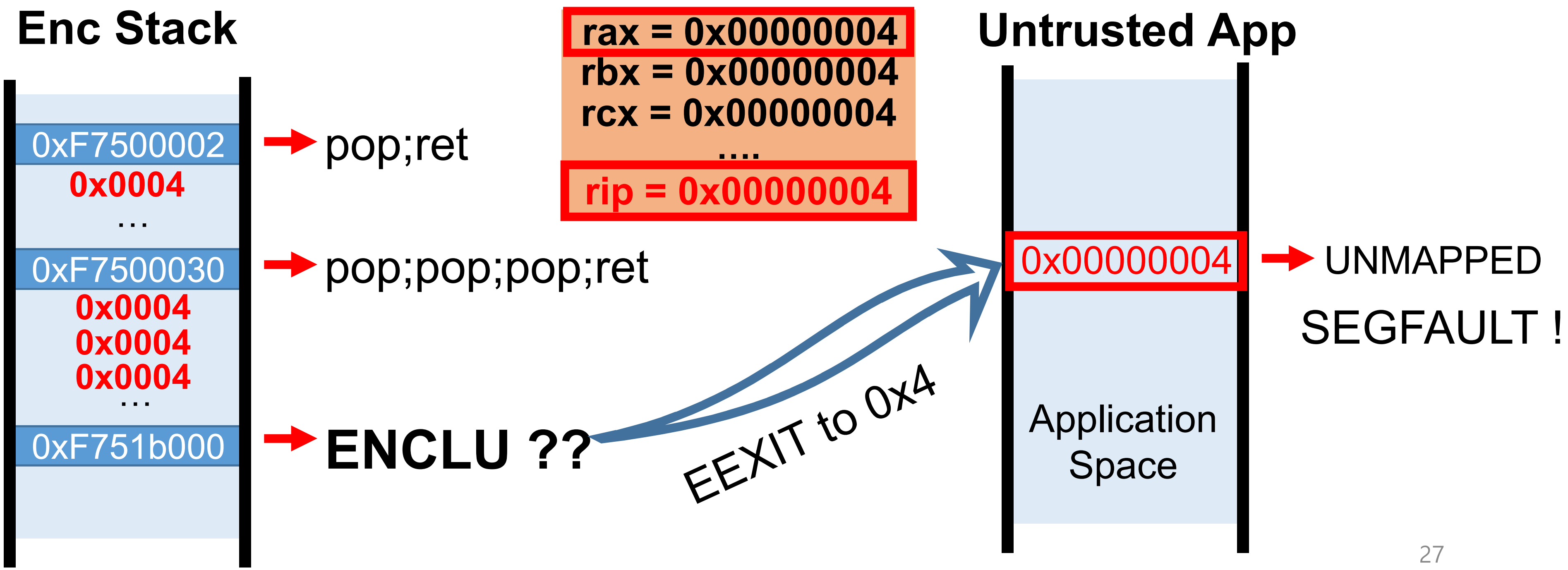
- IF POP gadget loads RAX = 4 and ENCLU at probing address then EEXIT happens

rax = 0x00000004
rbx = 0x00000004
rcx = 0x00000004
....



Step 2. Looking for ENCLU instruction

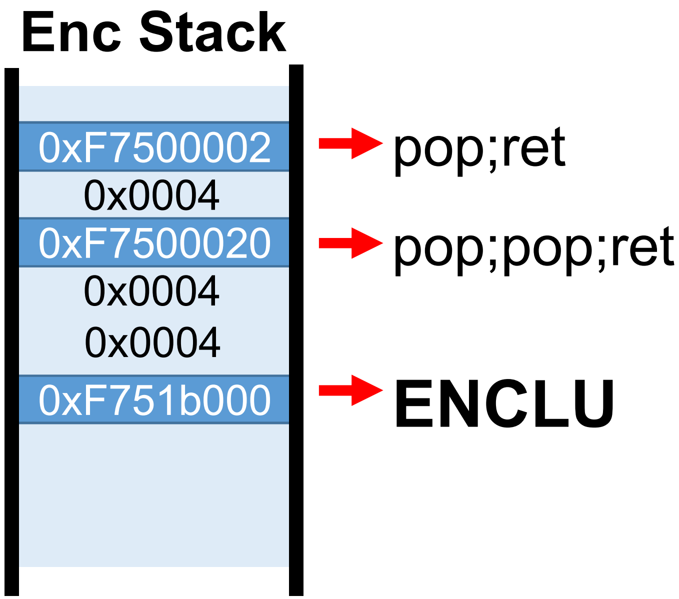
- How do we know whether eexit is invoked ?
- If EEXIT happens, it will jump to address loaded in RBX register.
- If pop rax; ret & pop rbx; ret gadget was chained, enclave exits to 0x4



Step 2. Looking for pop rax; ret

- Now, locate pop rax; ret; gadget

→ EEXIT (RAX == 0x4) / AEX (RAX == 0x3)
→ Chain gadgets one by one and checks EEXIT happens



SEGFAULT
&
RAX == 0x3

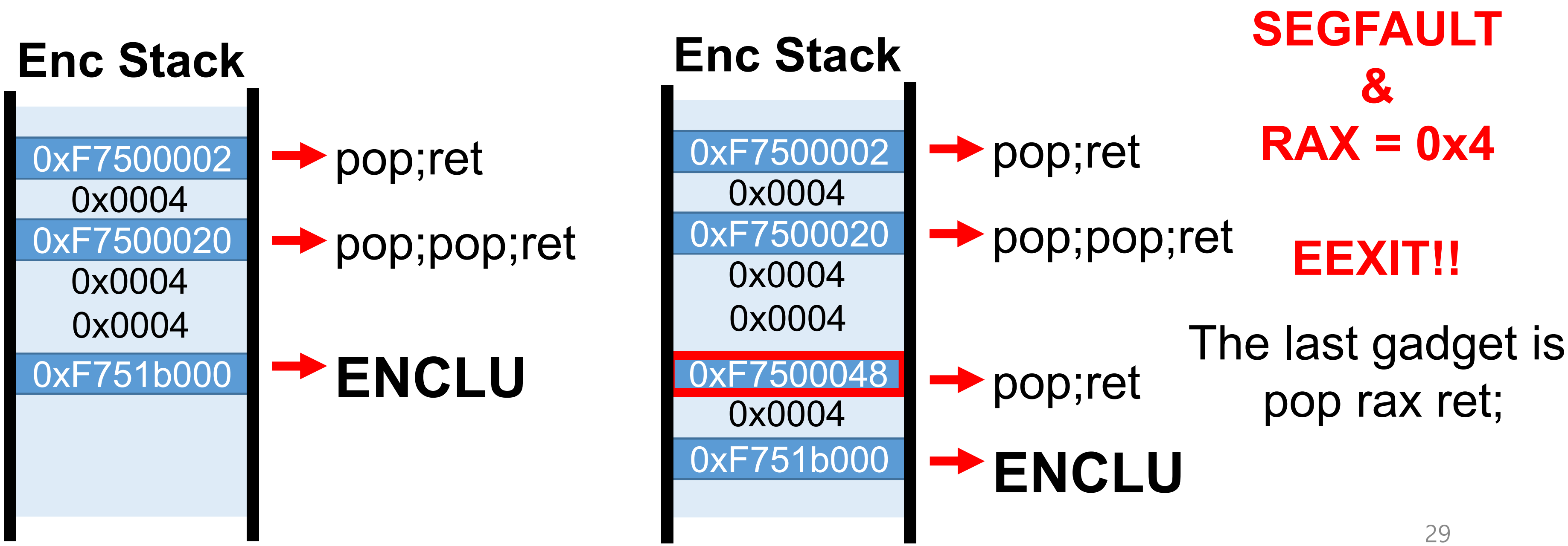
AEX!!

Step 2. Looking for pop rax; ret

- Now, locate pop rax; ret; gadget

→ EEXIT (RAX == 0x4) / AEX (RAX == 0x3)

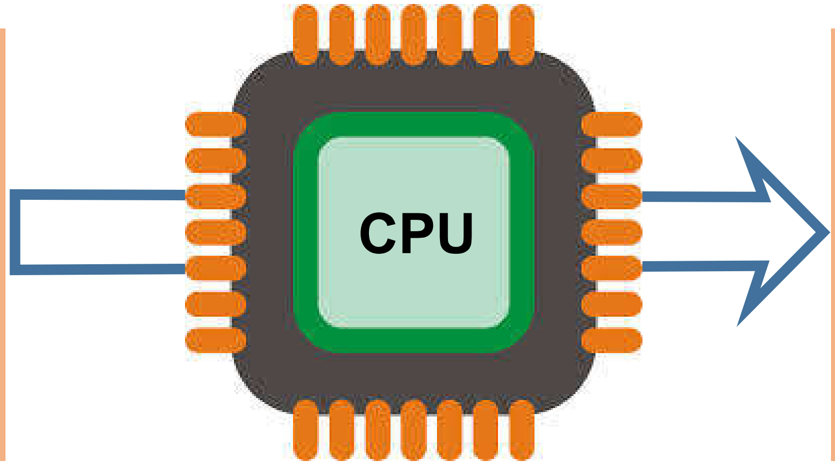
→ Chain gadgets one by one and checks EEXIT happens



Step 3. Deciphering pop gadgets: in search of **r?? registers**

Enclave Context

rax = 0x00000004
rbx = 0x00000004
rcx = 0x00000002
....
rdi = 0x00000001



Outside Enclave

rax = 0x00000004
rbx = 0x00000004
rcx = 0x00000002
....
rdi = 0x00000001

When EEXIT is invoked

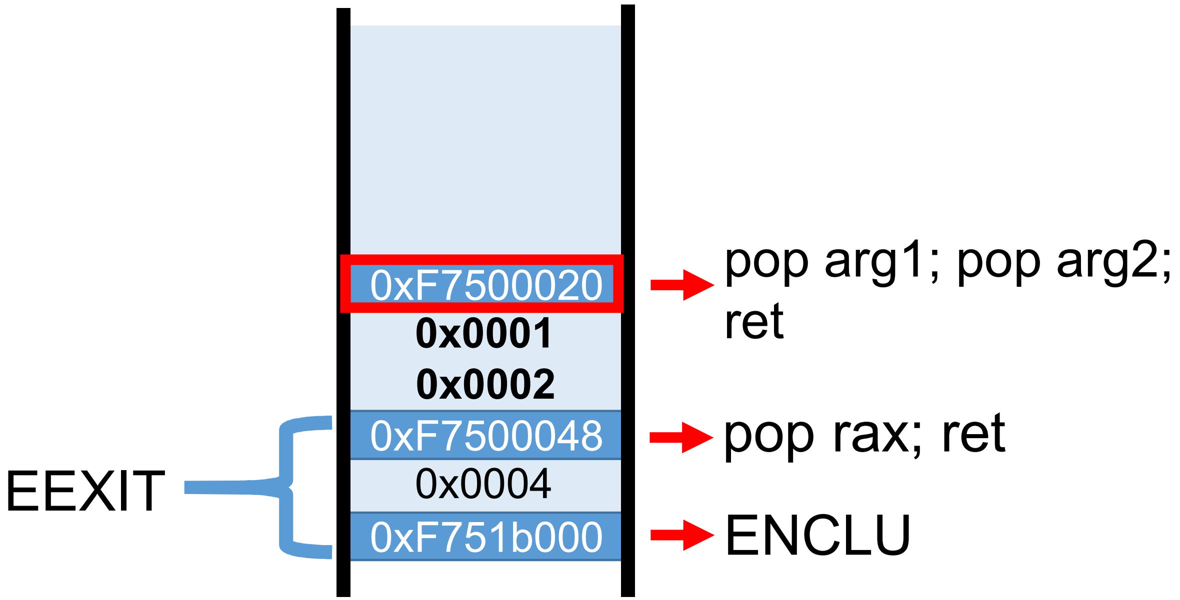
Enclave register values visible outside enclave

Step 3. Deciphering pop gadgets: in search of **r?? registers**

- EEXIT (ENCLU & rax=4) leaves register file uncleaned

  → Scan code for all pop gadgets

  → check arguments

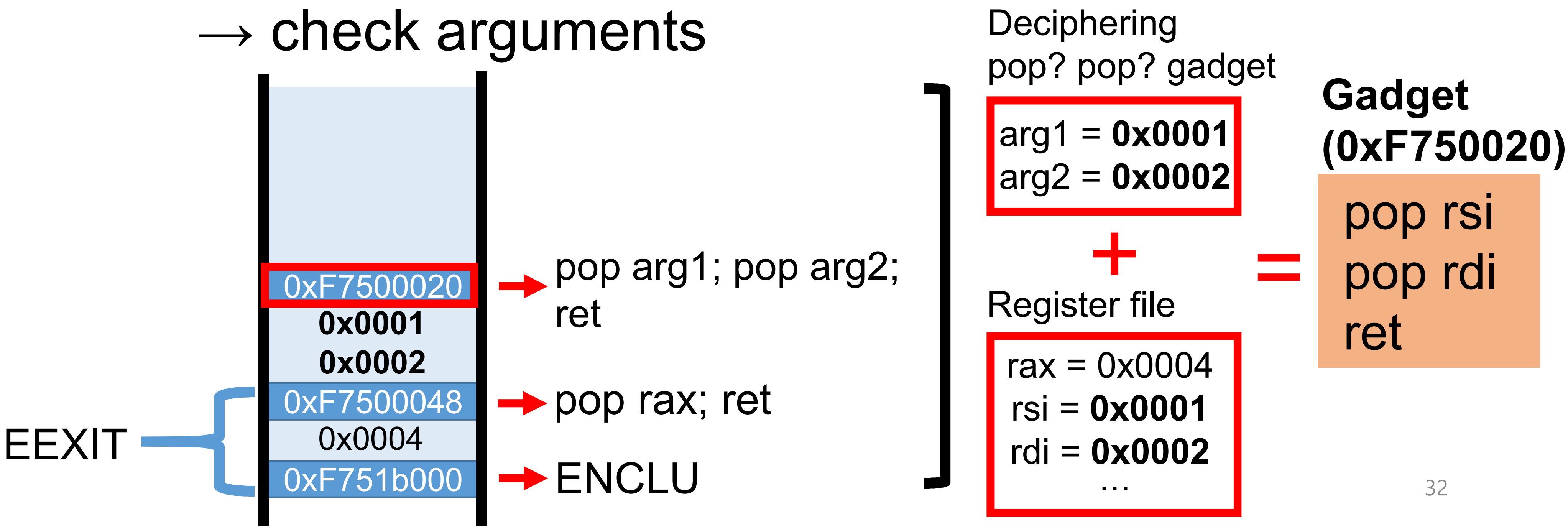


Step 3. Deciphering pop gadgets: in search of **r?? registers**

- EEXIT (ENCLU & rax=4) left a register file uncleaned

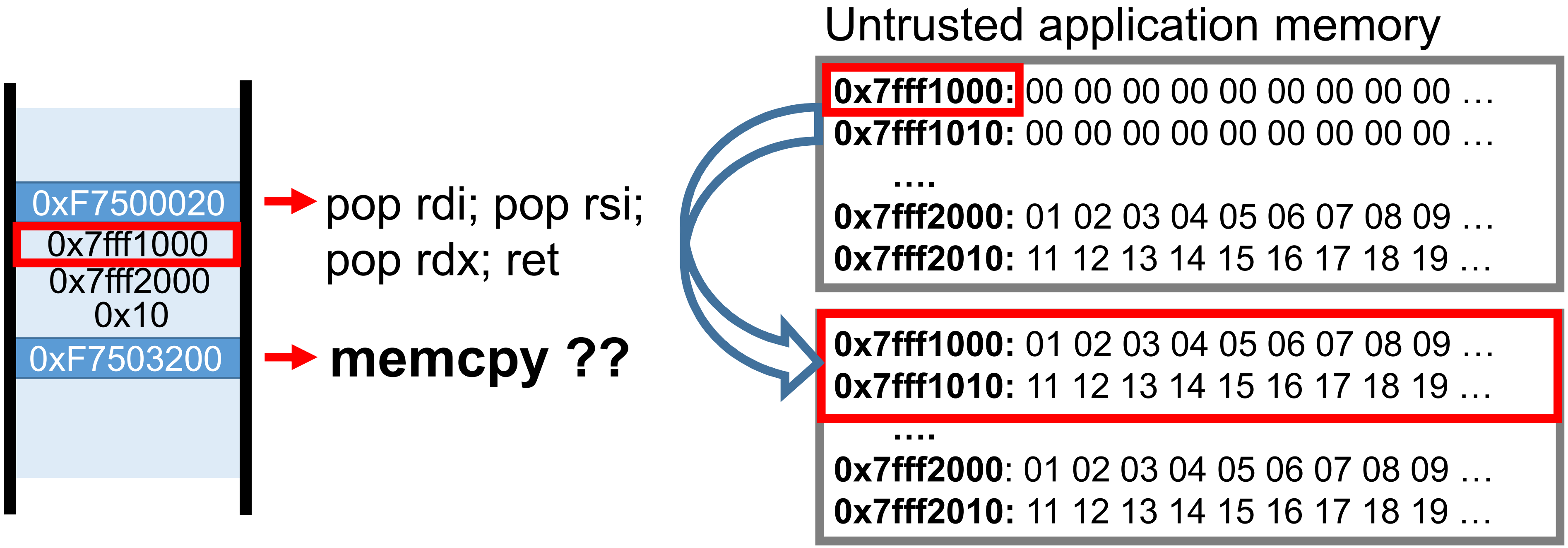
→ Scan code for all pop gadgets

→ check arguments



Step 4. Looking for memcpy()

- Identifying memcpy(**dst*,** some valid address, 0x10)



→ Check if "dst" contains data

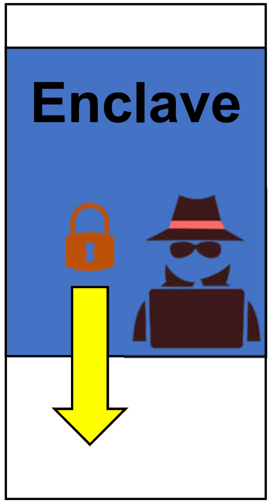
Gadgets everywhere (e.g., SDK)

Gadget	From
ENCLU Gadget	
`do_ereport:`	
`  enclu`	`libsgx_trts.a`
`  pop rdx`	
`  pop rcx`	
`  pop rbx`	
`  ret`	
`sgx_register_exception_h`	
`  mov rax, rbx`	`libsgx_trts.a`
`  pop rbx`	
`  pop rbp`	
`  pop r12`	
`  ret`	
`relocate_enclave:`	`libsgx_trts.a`
`  pop rsi`	
`  pop r15`	
`  ret`	
`  pop rdi`	
`  ret`	
Memcpy Gadget	
`memcpy:`	`libsgx_tstdc.a`

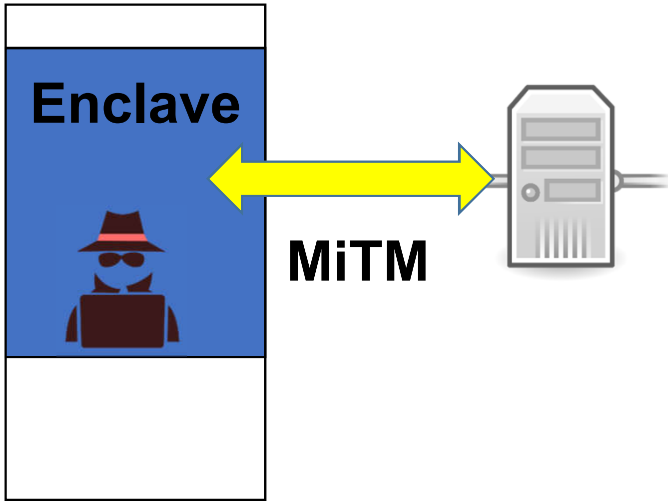
Gadget	From
GPR Modification Gadget	
`__intel_cpu_indicator_init:`	
`  pop r15`	`sgx_tstdc.lib`
`  pop r14`	
`  pop r13`	
`  pop r12`	
`  pop r9`	
`  pop r8`	
`  pop rbp`	
`  pop rsi`	
`  pop rdi`	
`  pop rbx`	
`  pop rcx`	
`  pop rdx`	
`  pop rax`	
`  ret`	
ENCLU Gadget	
`do_ereport:`	
`  enclu`	`sgx_trts.lib`
`  pop rax`	
`  ret`	

What can we do with all this?

Leak secrets

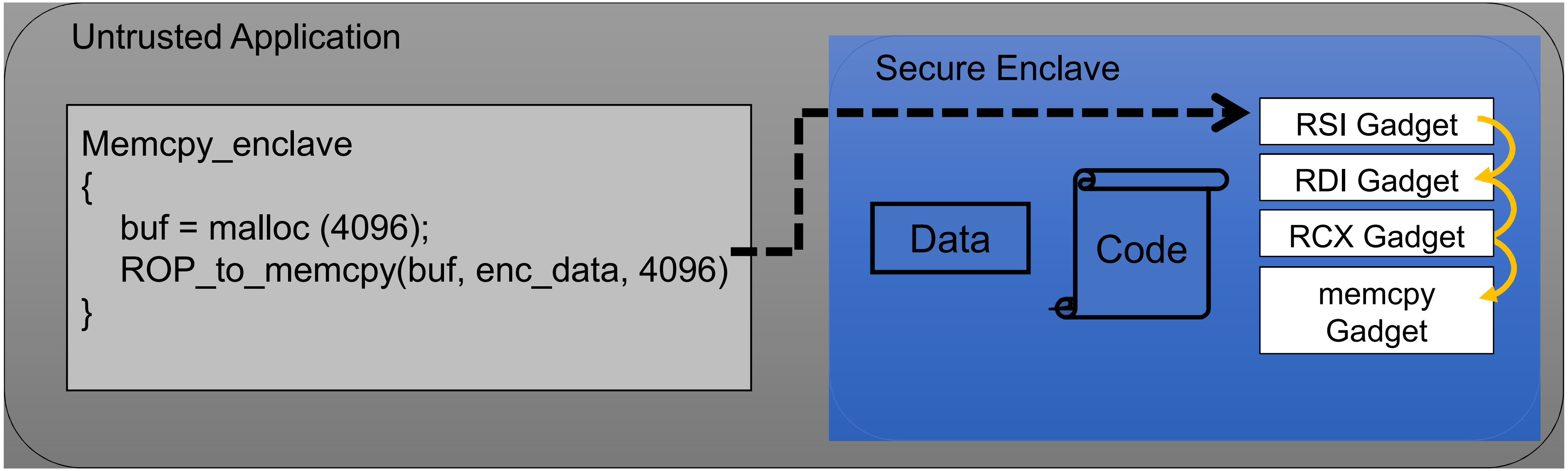


Emulated Enclave



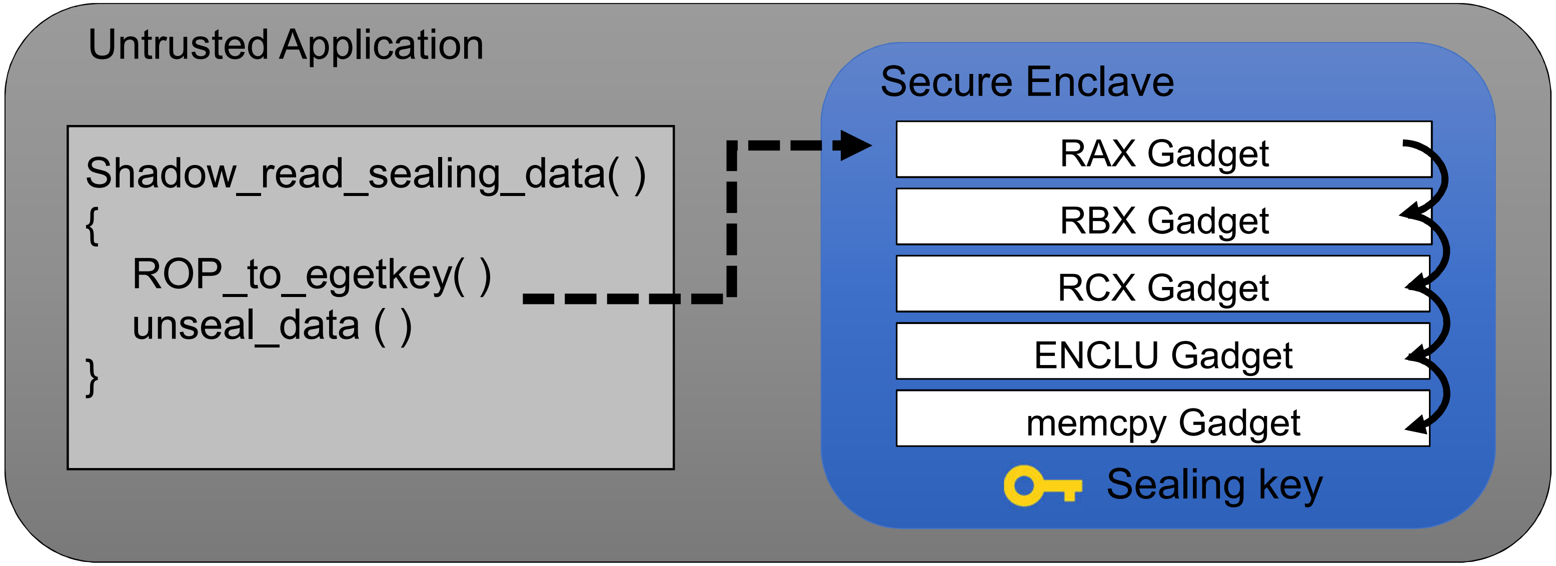
- Leak sensitive information
- Permanently parasite to the enclave program

Case study 0: Dumping confidential data



- Memcpy all enclave memory contents into untrusted memory
    - i.e., memcpy(non-enclave region, enclave, size)
- Complete breakdown in enclave confidentiality

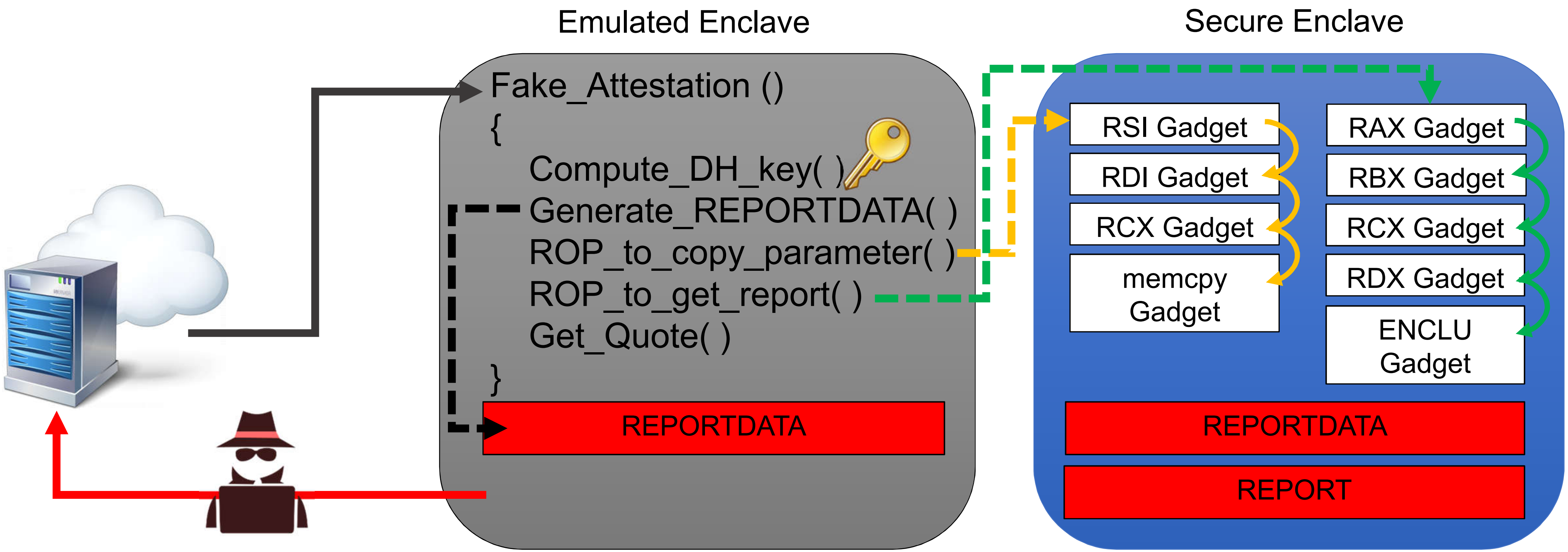
Case study 1: Compromising sealed data



Unsealing and leaking confidential data

i.e., EGETKEY retrieves the hardware key bound to specific enclave

Case study 2: Hijacking remote attestation

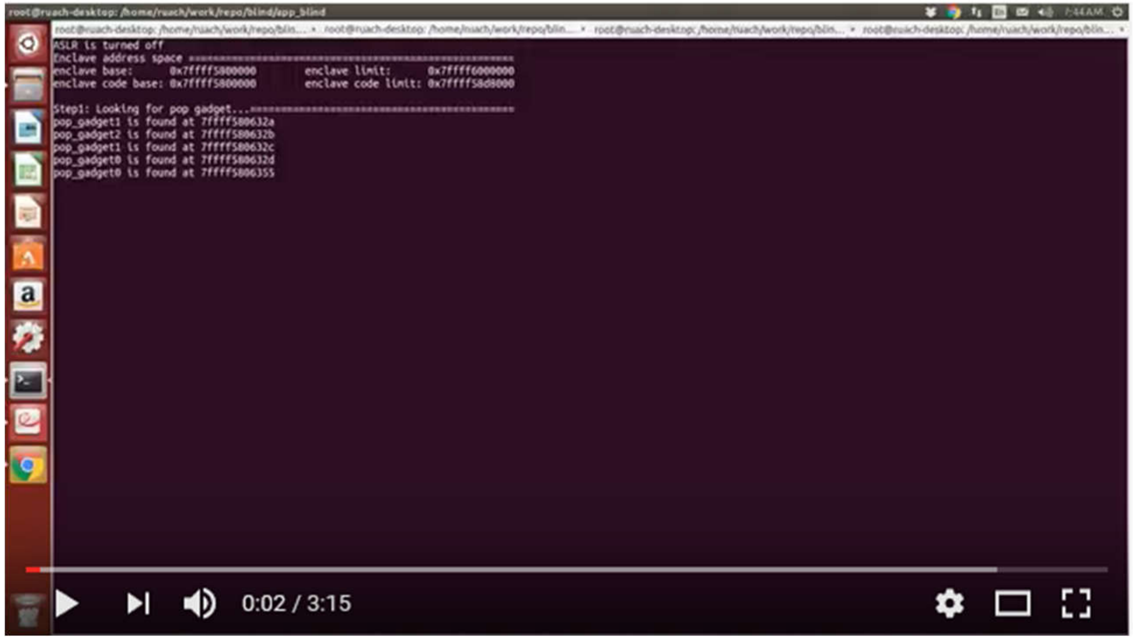


- Breaking the Integrity guarantees of SGX
  - MiTM between secure enclave and attestation server
  - Masquerading the enclave to deceive remote attestation server

Conclusion

- The first practical ROP attack on real SGX hardware
  - Exploits a memory-corruption vulnerability
- Demonstrates how the security of SGX can be disarmed.
  - Exfiltrate all memory contents from the enclave
  - Bypass the SGX attestation
  - Break the data-sealing properties
- Encourage the community
  - Explore the SGX characteristic-aware defense mechanisms
  - Develop an efficient way to reduce the TCB in the enclave.

DEMO: PoC Dark ROP



https://youtu.be/hyuZFf3QxvM

- Target binary: remote attestation example from Intel SDK
- Vulnerability: stack overflow

Q&A