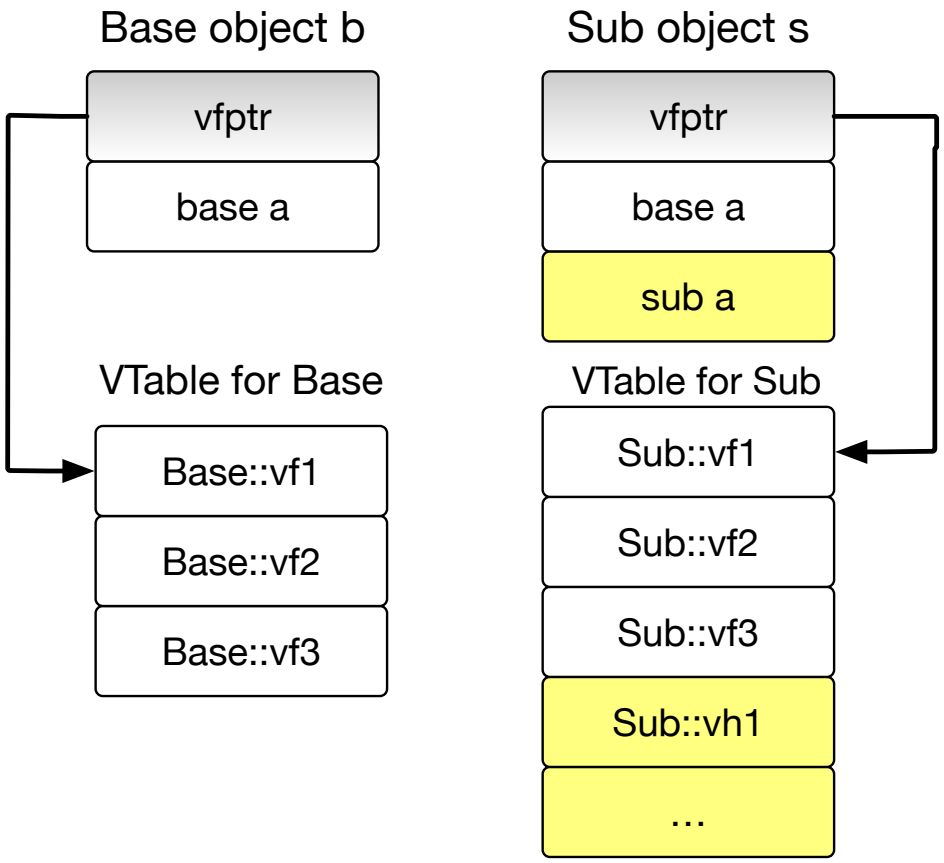
VScape: Assessing and Escaping Virtual Call Protections

Kaixiang Chen[1], Chao Zhang[123], Tingting Yin[1],
Xingman Chen[1], Lei Zhao[4]

1 Institute for Network Sciences and Cyberspace, Tsinghua University
2 Beijing National Research Center for Information Science and Technology
3 Tsinghua University-QI-ANXIN Group JCNS
4 School of Cyber Science and Engineering, Wuhan University

VTable for Dynamic Dispatch (C++)

class Sub: public Base {…};



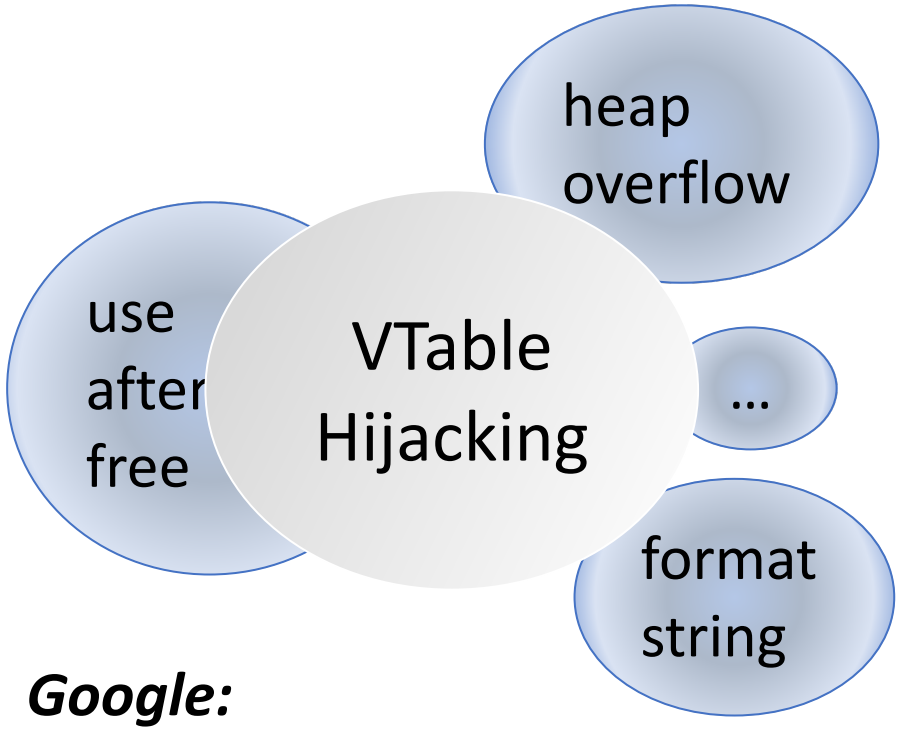

```
void foo(Base* obj){
  obj→vf3();
}

void main()
{
   Base* obj = new Sub();
   foo(obj);
}
```

```
mov rax, qword ptr [rcx] ;   load vptr
add rax, 16              ;   find vfptr
call [rax]               ;   invoke vf
```

Polymorphic functions are invoked via indirect call instructions.

VTable Hijacking in real world

- A common way to exploit



Google:

"80% attacks exploit use-after-free..."

Microsoft:

50% CVEs targeted Winows7 are UAF

Pwn2Own：

chrome 2014-2019

Firefox 2014-2019

Safari 2014-2020

.....

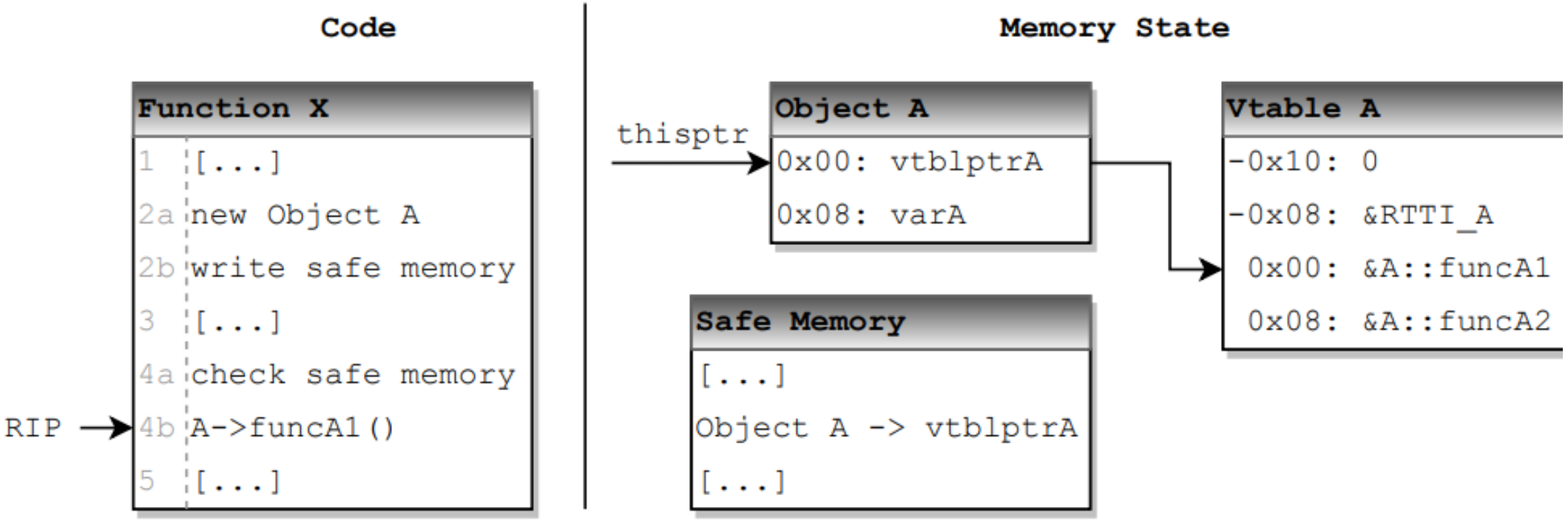
- written in C++
- BIG Targets in the Cloud

Virtual Call Protections

Type1: Protect integrity of vptr

Solution: DFI solutions, VPS

Con: high runtime overheads & hard to deploy



A VPS protected application

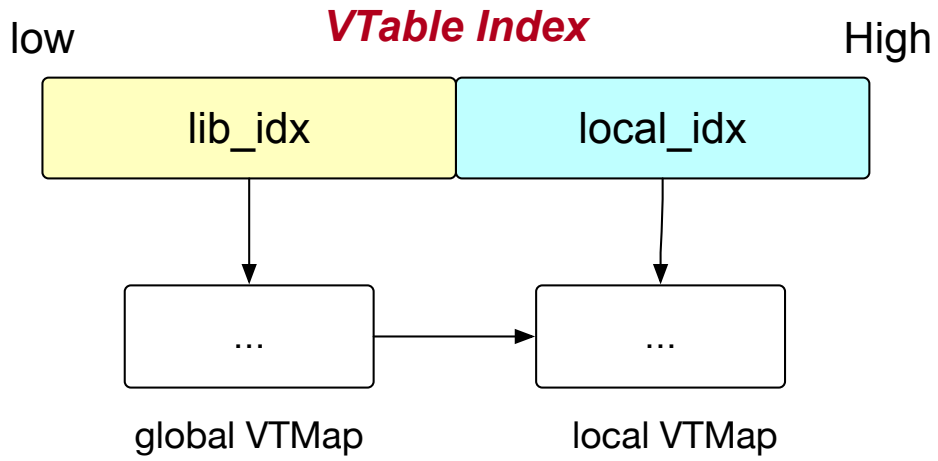
Virtual Call Protections

Type1: Protect integrity of vptr

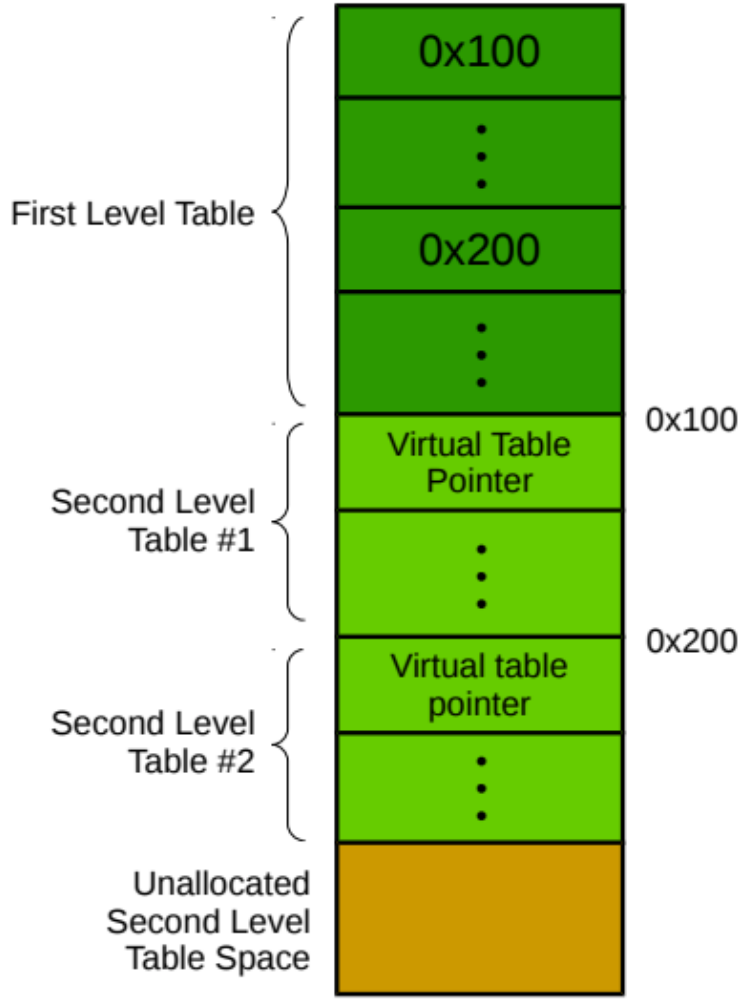
Type2: ABI incompatible methods

Solution: CFIXX, VTrust

Con: compatibility issue&hard to deploy



VTable pointer index in VTrust



CFIXX Metadata Memory Layout

Virtual Call Protections

Type1: Protect integrity of vptr

Type2: ABI incompatible methods

Type3: Validity check for virtual call targets

Solution(Coarse-grained): CCFIR, binCFI, LockDown

Con: not enough to stop PC hijack

Solution(Fine-grained):

MCFI, πCFI, CFI-LB, SafeDispatch,VTV&IFCC

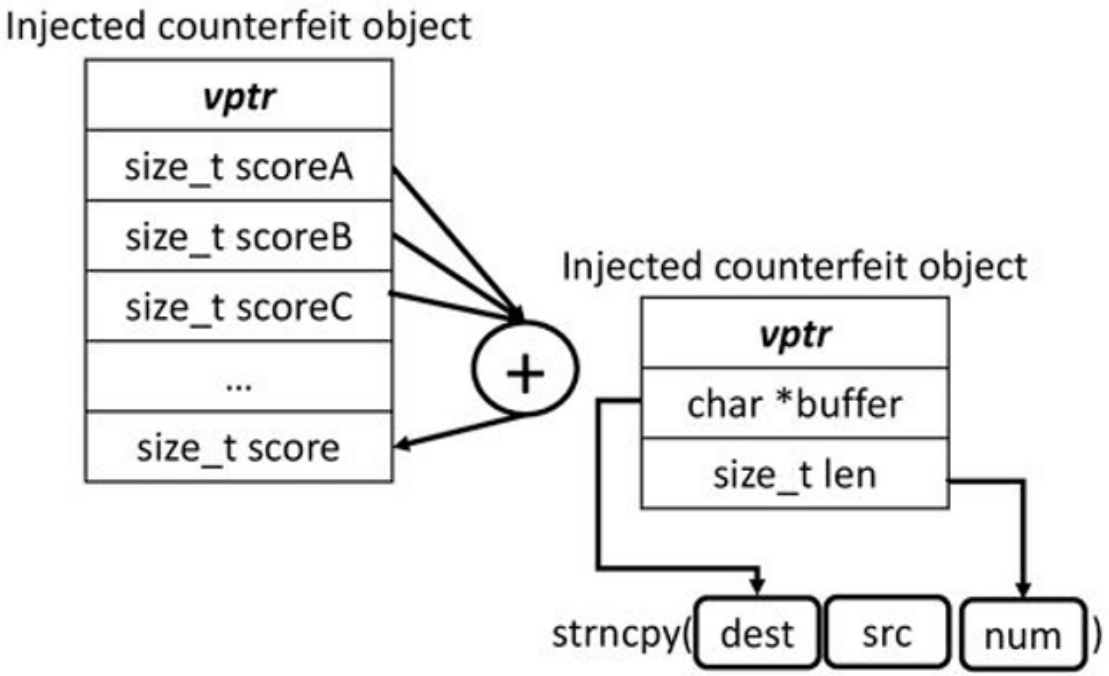
Con: corner cases of vcall and vfunc

```
movq    (%r12), %rbx  ; set rbx to the vptr
movq    %rbx, %rdi
callq   verify_vtable ; verify_vtable(vptr)
movq    %r12, %rdi
callq   *16(%rbx)     ; obj.foo()
movq    %r12, %rdi
callq   *24(%rbx)     ; obj.bar()
```

Implementation of VTV&IFCC

Existing attacks

- **C**ounterfeit **O**bject-**o**riented **P**rogramming (**COOP**): On the Difficulty of Preventing Code Reuse Attacks in C++ Applications (S&P'15)



Control-Flow Bending: On the Effectiveness of Control-Flow Integrity (Usenix'15)

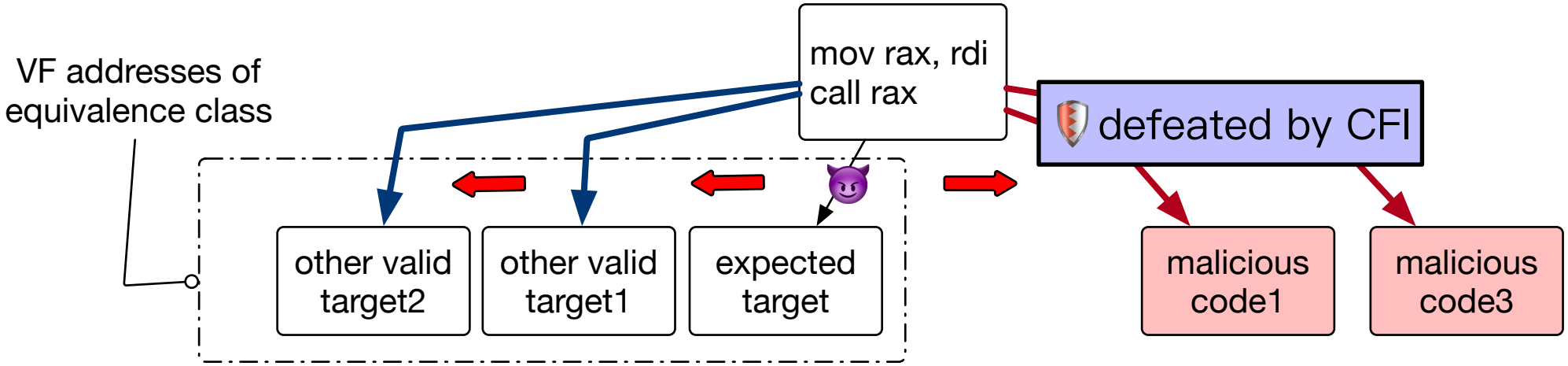
Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks (CCS'15)

Control jujutsu: On the weaknesses of fine-grained control flow integrity (CCS'15)

- several equivalent classes are merged due to imprecise points-to analysis

Our Work: COOPLUS

- Remaining attack surface:
  - call derived classes' virtual functions
  - at a virtual call site, ***all derived classes' vf*** *are callable*
    - *the object is not bound to its vftable.*
    - invoke any child class' vfunc on the existing compatible object.



What if we hijack PC toward counterfeit functions in such sets ?
No one dug into such a disordering execution.

Our Work: COOPLUS

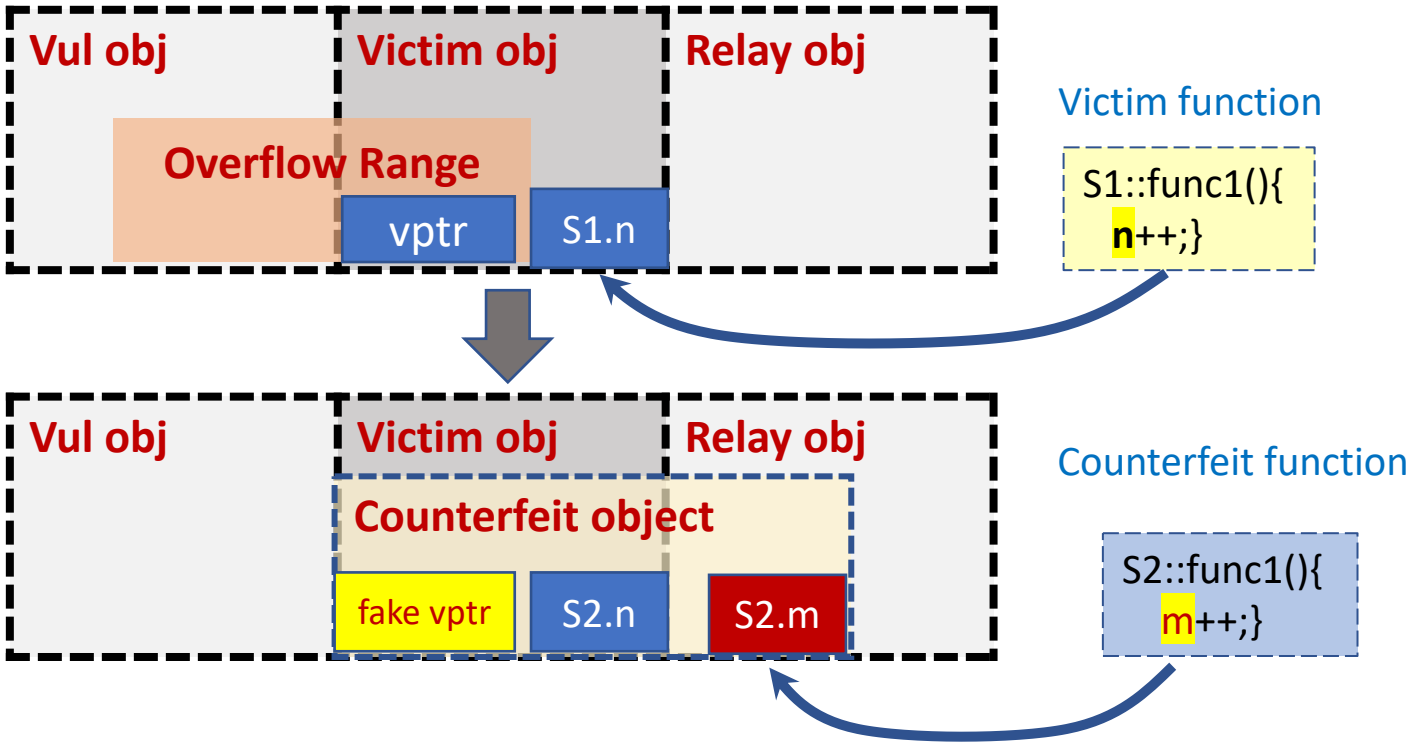
1. The adversary picks a vcall to hijack
2. Utilize the given vulnerability to corrupt a victim object (class S1) used at the vcall.
3. Virtual calls of this victim object (S1::func1) invokes a different virtual function

Vulnerable function

```
input_str(len){
   read(0, &buf, len);
} // overflow
```

Virtual call

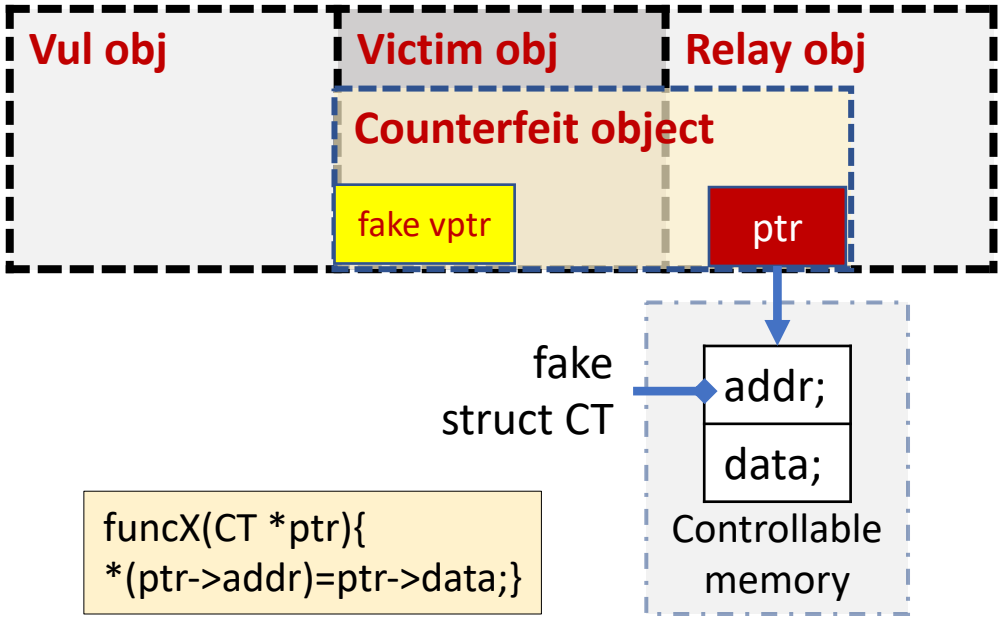
```
foo(Base* obj){
   obj->func1();
}//S1 and S2 derive from Base
```



The counterfeit function operates on the relay object

Our Work: COOPLUS

- Utilizing child classes' virtual functions to perform:
  - out-of-bound read： ***Ld-AW-Const/-nonCtrl/-Ctrl***、***Ld-EX-PC***
    - e.g. read and dereference a data pointer from relay object, denoted as Ld-AW-#XXX



Consequences of out-of-bound data read

- Ld-AW-Ctrl:

```
funcX(CT *ptr){
*(ptr->addr)=ptr->data;}
```

- Ld-AW-nonCtrl:

```
funcX(CT *ptr, int param){
*(ptr->addr)=param;}
```

- Ld-AW-Const:

```
funcX(CT *ptr){
*(ptr->addr)=FIXED_VALUE;}
```

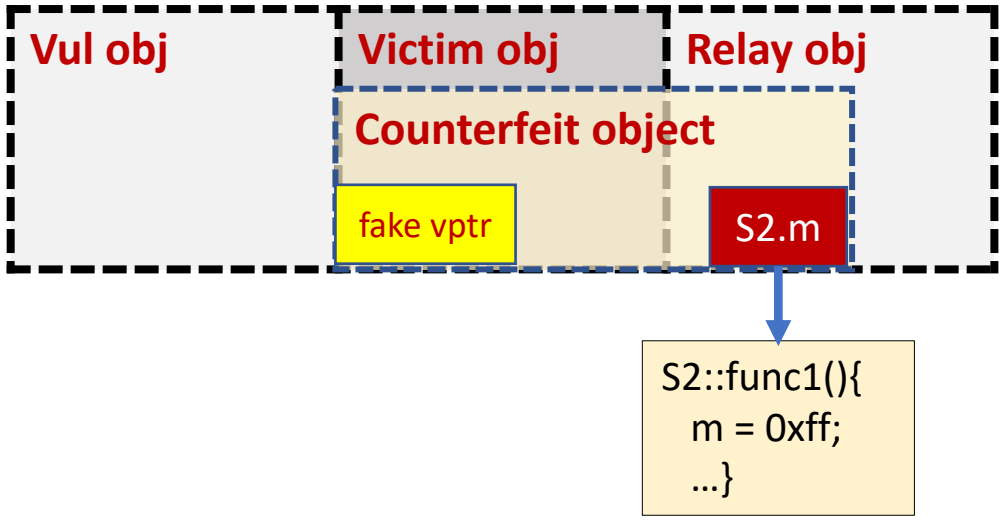
- Ld-EX-PC:

```
funcX(CT *ptr){
*(ptr->addr)();}
```

Exploit illegal memory read from relay object

Our Work: COOPLUS

- Utilizing child classes' virtual functions to perform:
  - out-of-bound write: ***St-nonPtr/-Ptr***



Consequences of out-of-bound data write.

- St-nonPtr:

```
S2::func1(){
  m = 0xff;
  …}
```

- St-Ptr:

```
S2::func1(){
  ptr = &global_ptr;
  …}
```

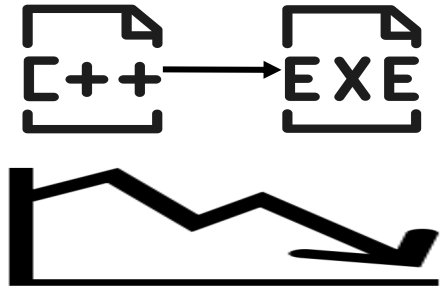
Exploit illegal memory write from relay object

How to launch COOPLUS ?

≥101,000 virtual calls
too many for manual analysis

```
#define TRACE_EVENT0(category_group, name)    \
  INTERNAL_TRACE_EVENT_ADD_SCOPED(category_group, name)
#define TRACE_EVENT_WITH_FLOW0(category_group, name, bind_id, flow_flags)  \
  INTERNAL_TRACE_EVENT_ADD_SCOPED_WITH_FLOW(category_group, name, bind_id, \
                                            flow_flags)
#define TRACE_EVENT1(category_group, name, arg1_name, arg1_val) \
  INTERNAL_TRACE_EVENT_ADD_SCOPED(category_group, name, arg1_name, arg1_val)
#define TRACE_EVENT_WITH_FLOW1(category_group, name, bind_id, flow_flags,  \
                               arg1_name, arg1_val)                        \
  INTERNAL_TRACE_EVENT_ADD_SCOPED_WITH_FLOW(category_group, name, bind_id, \
                                            flow_flags, arg1_name, arg1_val)
```

macro and template codes
make analysis mazy



info like class inheritance hierarchy
is lost in executable binary

Need a solution to help launch COOPLUS

Challenge 1:

find exploit primitives ***(virtual call, victim class, counterfeit class)***

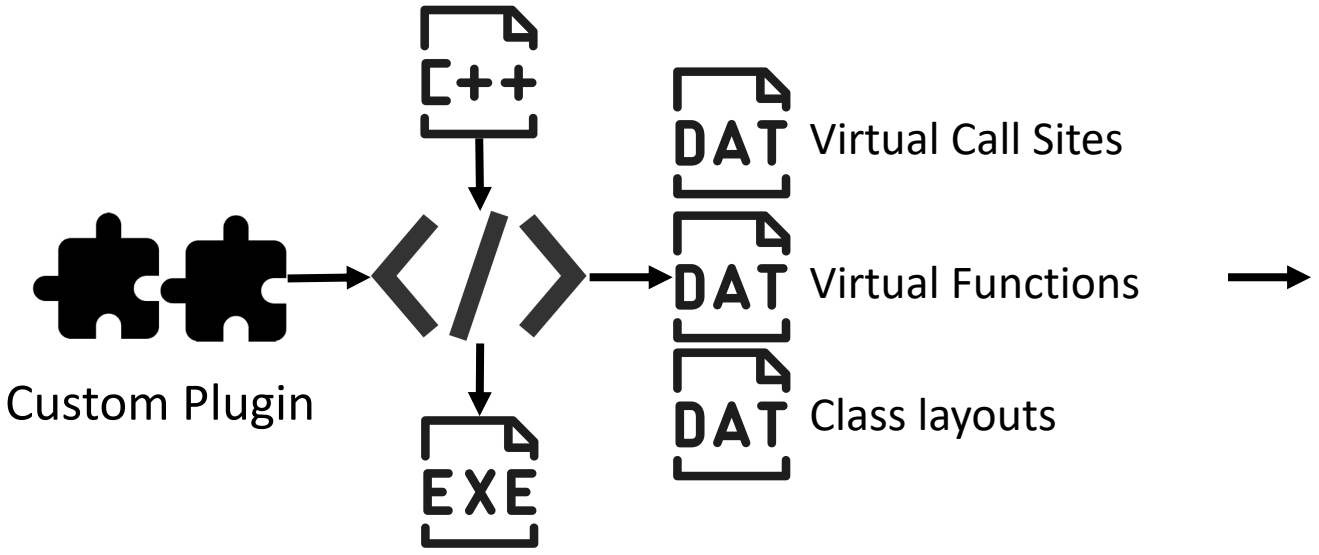
Challenge 2:

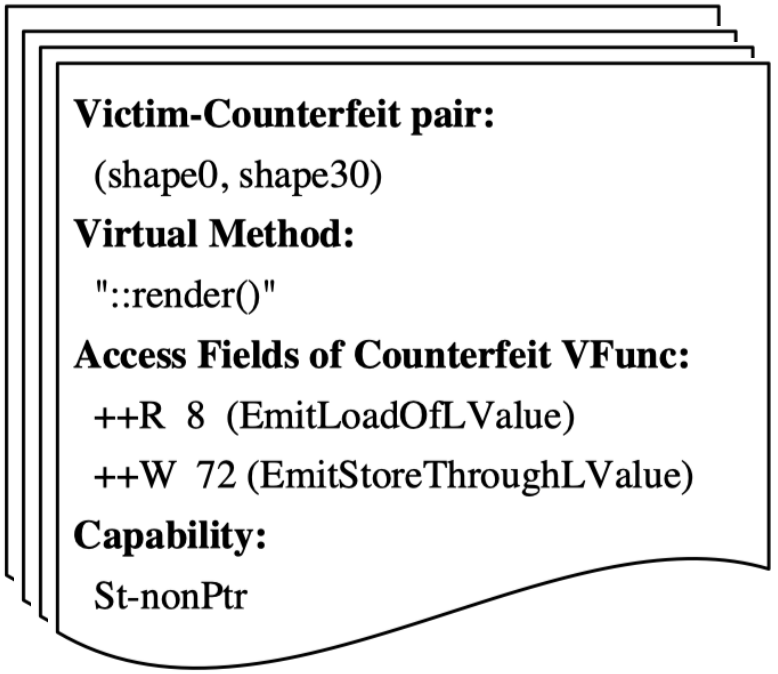
proper **inputs** to trigger virtual call, out-of-bound memory access, sensitive operations

Primitive Generator: VScape

We name our primitive generator as **VScape**:

Step 1: Primitive Generation
- Search primitive candidates from C++ source code





Sample Records for Primitive Pair

Primitive Generator

We name our primitive generator as **VScape**:

Step 1: Primitive Generation

Step 2: Expected Primitive Construction
- Match vulnerability with applicable primitives

IN:
Candidate Primitives,
Expected Primitive Attributes,
Vulnerability Description

OUT:
Memory States Constraints(MSC)

- We design a **description model** for vulnerability.
- **Expected Primitive Attribute** is used to determine which type of COOPLUS primitive is expected in current case. (i.e., Ld-AW-Ctrl)
- Victim object and the adjacent relay object are marked as symbols
- **Symbolically executes** the counterfeit function to get MSC

Primitive Generator

We name our primitive generator as **VScape**:

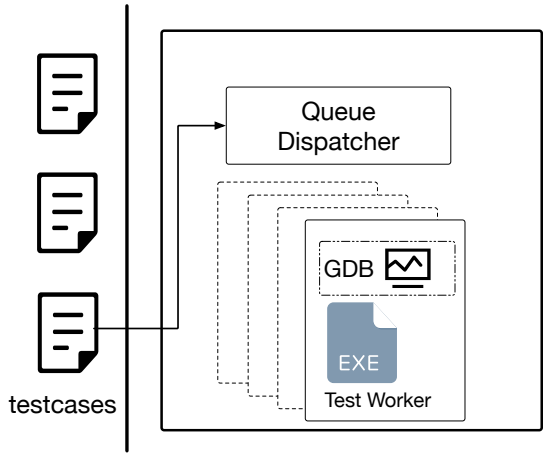
Step 1: Primitive Generation

Step 2: Expected Primitive Construction

Step 3: Exploit Constraint Solving
- Generate proper inputs in dynamic tests
  - Reachability of Victim Functions
  - Reachability of OOB Instructions
  - Exploit Assembling

```
def main():
    heap_operation_before_relay_object()
    gen_relay_object_and_fake_object()
    heap_operation_before_victim_object()
    gen_allocate_victim_object()
    vul_trigger()
    gen_invoke_counterfeit_function()
    operations_after_cooplus()
```

- The user-provided exploit template determines the exploit strategy



- Dynamic tests are implied to deal with
- reachability issues

Evaluations

- RQ1: Is COOPLUS exploit primitives **prevalent** in real world ?
- RQ2: Can COOPLUS bypasses various **virtual call protections** ?
- RQ3: Is VScape **practical** when given real world vulnerabilities?

Evaluations (1/3)

- **RQ1:** What is the popularity of COOPLUS exploit primitives in real world C++ applications?
  - Evaluated on 14 open source C++ programs
  - All applications have hundreds of virtual functions
  - 88% of virtual calls only have **one candidate**
  - COOPLUS exploit primitives are **very popular in big projects**

App	Unique Virtual Call (UVC)	#UVC-CC	#UVC-CVF	#UVC-OVF	#UVC-OVF (μ/σ/Med:)	All Primitives
chromium	61,315	18,874 (74%)	2,279 (9%)	11,808	3.5/12.8/2	535,007
firefox	25,224	34,371 (56%)	7,205 (12%)	3,432	3.2/16.7/2	83,786
qt	6,764	4,730 (69%)	1,662 (25%)	4,468	5.3/34.5/2	508,141
opencv	9,183	883 (9%)	182 (2%)	1,216	14.1/33.1/2	55,116
oce	3,738	1,877 (50%)	609 (16%)	1,123	3.7/4.1/2	4,040

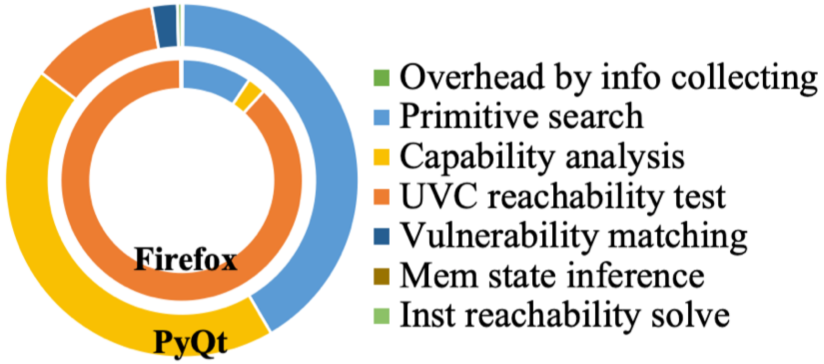
#UVC-CC: UVCs with ***multiple Compatible Classes***, **#UVC-CVF**: UVCs with ***multiple Compatible VFuncs***.
#UVC-OVF: UVC with ***OOB VFunc pairs***. μ: Average number of VFunc Variants for each UVC

Evaluations (2/3)

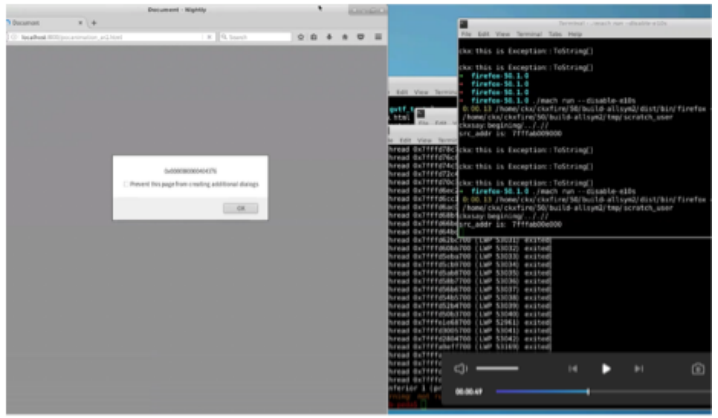
- **RQ2**: Is COOPLUS effective at defeating various virtual call protections?
  - We test 11 virtual call protections against COOPLUS
  - CFI without considering C++ semantics (CCFIR, binCFI, LockDown) are all vulnerable to COOP and COOPLUS
  - **CFIXX** and **µCFI** are effective against COOPLUS
  - C++ semantic aware approaches (OS-CFI, MCFI, πCFI, CFI-LB, SafeDispatch) can be bypassed by COOPLUS

Evaluations (3/3)

- **RQ3**: Is VScape effective at generating exploit primitives when given real world vulnerabilities?
  - 2 real world cases
    - Mozilla Firefox 50.1 + *CVE-2018-5146*
    - Python-3.6.7 with PyQt-5.12 library + *CVE-2014-1912*



Time cost distribution of each analysis phase



Exploits with capability of arbitrary read and write

Conclusion

- An advanced attack **COOPLUS** to bypass virtual call protection
- A solution **VScape** to assess the effectiveness of virtual calls defenses against this attack
- Experiments show real-world applications have a large set of exploitable virtual calls
- CFI protecting the integrity of *vptr* with a low performance overhead and good compatibility is still badly in need

Thanks for listening!

Q & A

Contact: Kaixiang Chen, ckx18@mails.tsinghua.edu.cn