VScape: Assessing and Escaping Virtual Call Protections

Kaixiang Chen[1], *Chao Zhang*[1,2,3]✉, *Tingting Yin*[1], *Xingman Chen*[1], *Lei Zhao*[4]

[1]*Institute for Network Science 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*

{ckx18,ytt18,cxm16}@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn, leizhao@whu.edu.cn

Abstract

Many control-flow integrity (CFI) solutions have been proposed to protect indirect control transfers (ICT), including C++ virtual calls. Assessing the security guarantees of these defenses is thus important but hard. In practice, for a (strong) defense, it usually requires great manual efforts to assess whether it could be bypassed, when given a specific (weak) vulnerability. Existing automated exploit generation solutions, which are proposed to assess the exploitability of vulnerabilities, have not addressed this issue yet.

In this paper, we point out that a wide range of virtual call protections, which do not break the C++ ABI (application binary interface), are vulnerable to an advanced attack COOPLUS, even if the given vulnerabilities are weak. Then, we present a solution VScape to assess the effectiveness of virtual call protections against this attack. We developed a prototype of VScape, and utilized it to assess 11 CFI solutions and 14 C++ applications (including Firefox and PyQt) with known vulnerabilities. Results showed that real-world applications have a large set of exploitable virtual calls, and VScape could be utilized to generate working exploits to bypass deployed defenses via weak vulnerabilities.

1 Introduction

To mitigate control flow hijacking attacks, many control-flow integrity (CFI) solutions [1, 2] have been proposed. In principle, CFI solutions validate the transfer targets of each indirect control transfer (ICT) instruction, including indirect call/jump and return instructions, enforcing them fall into a corresponding equivalence class (EC). Virtual functions in C++ programs are lowered to indirect call instructions in binary code, and thus benefit from CFI solutions as well.

Early CFI solutions [3, 4] did not take C++ semantics into consideration, and thus allowed virtual calls (denoted as *vcalls*) to transfer to a large number of targets. Researchers pointed out their weaknesses and proposed the COOP [5] attack to bypass these defenses at virtual call sites. Some other CFI solutions [6, 7] are C++ semantics aware, and provide fine-grained defenses for virtual calls, defeating the COOP attack. Recent CFI solutions [2, 8] take runtime information (e.g., data origin) to further reduce the size of EC for virtual calls and provide a stronger defense.

Despite a considerable amount of efforts to defeat attacks, it is still not clear whether these defenses are strong enough to protect virtual calls from advanced attacks, given the continuously evolving arm-race between offense and defense. For example, according to the C++ language specification, a virtual function call site, which expects a virtual function from a statically declared base class, by design is allowed to jump to all variant virtual functions overridden in derived classes. Thus, the EC set is still large. In practice, it requires great manual efforts to assess the exploitability of (potentially weak) vulnerabilities, especially when some (potentially strong) defenses are deployed. In general, analysts have to comprehend the application and the vulnerability, and search for proper exploit primitives in the target application which may have a large code base, then assemble these primitives to exploit the target vulnerability. This process is time-consuming and needs automated solutions.

To automatically assess the exploitability of vulnerabilities, several automated exploit generation (AEG) solutions [9–11] have been proposed. However, none of them have taken modern defenses into consideration, and thus fail to assess their security guarantees. For instance, AEG solutions targeting heap vulnerabilities, e.g., Revery [11] and Gollum [12], only work well when the defense ASLR [13] is disabled. AEG solutions targeting stack-based buffer overflow, e.g., Q [14], CRAX [15] and PolyAEG [16], cannot bypass stack canary [17]. Thus, developing an AEG solution to assess the security of a defense solution is necessary.

In this paper, we assess the effectiveness of virtual calls defenses and proposed a solution VScape to facilitate the assessment. We point out that, each virtual call protection is vulnerable to an advanced attack COOPLUS, as long as it (1) does not break the application-binary interface (ABI) of virtual calls, (2) cannot guarantee the integrity of C++ objects' VTable pointers, and (3) allows multiple transfer targets at virtual call sites. COOPLUS is essentially a code reuse

attack, which invokes type-conformant (but out-of-context) virtual functions at victim virtual call sites. Such invocations are allowed by C++ semantic aware CFI solutions, but operate on out-of-context objects, and thus could cause further consequences, e.g., control flow hijacking.

VScape could facilitate this exploitation process. Specifically, it analyzes the target application, scans all vcall sites and finds compatible classes, then filters virtual functions that could cause memory safety violations, and finally compiles proper exploit primitives to generate final exploits. To the best of our knowledge, VScape is the first solution to generate exploits to bypass virtual call protections. It shows following intriguing features which previous researches have not exhibited. It is able to assess the security of a large number of defenses for virtual calls and assist in generating exploits to bypass them. It could assess the exploitability of many types of vulnerabilities, even some types of vulnerabilities that are hard to exploit in practice. Further, it could yield a massive number of exploit primitives, which could greatly facilitate manual exploit generation.

We implemented a prototype of VScape based on Clang [18] and Angr [19], and evaluated it on 14 real world C++ applications including Firefox and PyQt, which are hardened with 11 CFI solutions. Results showed there is a large attack surface of exploitable virtual call sites in real world applications. Most virtual call protections can be bypassed by COOPLUS, and VScape could be utilized to generate working COOPLUS exploits when given known vulnerabilities. We pointed out that, to fully mitigate COOPLUS, a solution which protects the integrity of `vptr` with a low performance overhead and good compatibility is demanded.

In summary, we made the following contributions:

- We pointed out an advanced attack COOPLUS, able to bypass a wide range of virtual call protections, even when only weak vulnerabilities are given.
- We presented a solution VScape to assess the effectiveness of virtual call protections against COOPLUS, including the available attack surface and exploit primitives, and to assist in generating working exploits.
- We implemented a prototype of VScape and evaluated it on real world applications Firefox and PyQt hardened with virtual call protections. Results showed that the attack surface is large and bypassing virtual call protections is feasible in practice.

2 Background

2.1 VTables and Virtual Calls

In C++ applications, a virtual function in a base class can be overridden in a derived class. When a virtual function claimed in a base class is invoked at a virtual call site, the actual function invoked at runtime may belong to a derived class, depending on the runtime object's type.

To support this polymorphism feature, compilers employ a dynamic dispatch mechanism, in which polymorphic functions are invoked via indirect call instructions. As presented in the Itanium and MSVC C++ ABI, which are followed by major compilers including GCC, Clang and Microsoft MSVC, pointers to all polymorphic virtual functions (denoted as *vfptr*) of each class are kept in a separate *Virtual Function Table* (VTable) bound to this class, and a pointer *vptr* to the VTable is attached to each object of this class. Since C++ supports multiple types of inheritances, including single, multiple, and virtual inheritance, an object may have multiple *vptr* located at different offsets.

A typical virtual call is shown as below, which comprises of 3 steps: (1) dereference the *this* pointer of the runtime object to get its vptr, i.e., address of the VTable; (2) find the vfptr in target VTable, by adding a fixed offset, and (3) retrieve the vfptr and invoke the virtual function.

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

Note that, `vptr` is retrieved from an object in the heap. Therefore, given a proper vulnerability, an adversary could exploit it to tamper with `vptr`, hijack the followed virtual call. This is the common and well known VTable hijacking [20] attack.

2.2 Virtual Call Protections

To defend against VTable hijacking attacks, researchers have proposed multiple protection techniques.

As tampering with `vptr` is the entry to launch VTable hijacking attacks, a straightforward solution is to guarantee the integrity of `vptr`. Generic data flow integrity (DFI) techniques [21, 22] can serve this purpose. VPS [23] directly provides DFI to `vptr` for binary programs, but suffers from precision issues in binary analysis. This type of defense can protect `vptr` from being overwritten, but in general has high runtime overheads and is rarely deployed in practice.

Another type of defenses breaks the C++ ABI to protect virtual calls. For instance, CFIXX [24] places `vptr` in a separate metadata table, and leverages the Intel Memory Protection Extensions (MPX) hardware feature to protect the metadata table's integrity. VTrust [7] replaces each `vptr` with an index to a protected table, and enforces users to use VTable pointers in the table. However, it does not protect the integrity of the `vptr` , leaving potential attack surfaces.`vptr`, In general, this type of protection breaks the C++ ABI to block attackers, but at the same time, it leads to a severe compatibility issue and hinders the broad deployment.

The third type of protection technique checks the validity of each virtual call's target. Most CFI solutions fall into this category. Some recent CFI solutions, e.g., OS-CFI [2] and μCFI [8], utilize runtime data flow information to reduce the size of EC (even to 1). If a virtual call is only allowed to one target, then it is guaranteed to be safe. However, runtime data collection in general is hard to deploy in practice.

Most CFI solutions aim at both security and practical-

ity. Coarse-grained CFI solutions, e.g., BinCFI [3] and CCFIR [4], do not take type information or C++ semantics into consideration, and thus allow virtual calls to transfer to a large number of targets. Fine-grained CFI solutions, on the other hand, utilize such information to provide stronger defenses. For instance, LLVM-CFI [6] and TypeArmor [25] utilize type information, while VTrust [7] and vfGuard [26] utilize C++ semantics, to provide stronger defenses for virtual calls. As this type of defenses is popular and practical, we focus on assessing their effectiveness in this paper.

2.3 The COOP Attack

Multiple studies [27, 28] have demonstrated that coarse-grained CFI solutions are too permissive and can be bypassed. Specifically, for virtual calls, researchers proposed the counterfeit object-oriented programming (COOP) [5] attack to bypass coarse-grained defenses at virtual call sites.

COOP is, in essence, a code reuse attack, which utilizes the fact that all existing virtual functions (even arbitrary address-taken functions) are allowed at virtual calls if CFI solutions do not precisely consider C++ semantics. COOP exploits two key factors: (F1) a set of virtual call sites (denoted as *vfgadget*) which invoke existing but out-of-context virtual functions, and (F2) a special *vfgadget* which can orchestrate other *vfgadgets*, and accordingly prepares a set of counterfeit C++ objects to chain *vfgadgets* and launch attacks.

However, the factor F2 is rare in applications, while the factor F1 relies on the assumption that deployed defenses have not considered C++ semantics. As the COOP paper [5] claimed, *COOP's control flow can be reliably prevented when precise C++ semantics are considered from source code*. Thus, COOP cannot bypass many CFI solutions, e.g., LLVM-CFI [6] and VTrust [7].

3 COOPLUS Attack

Different from the claim made in [5], we pointed out COOP is more powerful than that realized by its authors. In this section, we present a variant of COOP, named COOPLUS, which is able to bypass C++ semantics aware CFI defenses.

3.1 Assumptions

We assume that widely deployed mitigations like including DEP (Data Execution Prevention [29]), ASLR (Address Space Layout Randomization [13]) and stack canary [17], are enabled on the target. We also assume that the target virtual call protection to assess is C++ semantics aware but does not break the C++ ABI nor protect the integrity of `vptr`.

On the other hand, we assume a weak vulnerability (e.g., one-byte heap overflow) is given[1]. Existing literature on attacks usually assumes the target application has a strong vulnerability, e.g., which allows writing arbitrary values to arbitrary addresses. In this paper, we only assume the target application has one memory corruption vulnerability that can

[1] But weaker vulnerabilities have lower probabilities to be exploited.

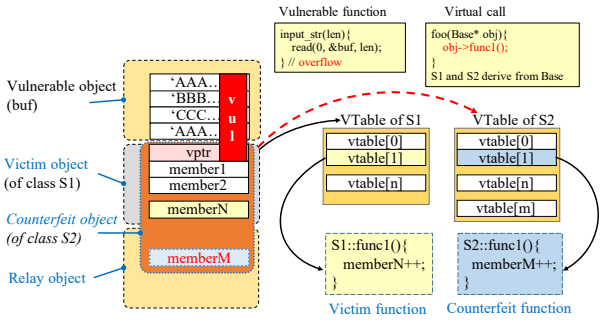


Figure 1: An example COOPLUS attack.

be exploited to tamper with one C++ object's *vptr*. This assumption makes our attack more realistic and reasonable.

We only focus on escaping virtual call protections, but other defenses in use may also hinder end-to-end exploits. Thus, we assume the adversary has necessary capabilities, e.g., information leaks and heap spraying, to bypass other defenses (e.g., ASLR). Automated escaping those defenses is out of the scope of this paper.

3.2 Principle of COOPLUS

COOPLUS is, in essence, a code reuse attack. More specifically, it is a variant of the proposed COOP attack. As COOP bypasses coarse-grained CFI defenses by invoking existing virtual functions at virtual call sites, COOPLUS invokes only type-compatible virtual functions to bypass stronger defenses, e.g., CFI solutions that are C++ semantics aware.

As shown in Figure 1, a virtual call site in the function `foo` expects a virtual function declared in the `Base` class. By design, this vcall site could invoke any overridden virtual function in derived classes (e.g., Sub1 and Sub2 in the figure), according to the C++ specification. In other words, virtual call protection has to allow virtual calls to invoke a large set[2] of compatible virtual functions.

COOPLUS works as follows. The adversary first picks a vcall (e.g., a invocation of Base::func1) to hijack, then utilizes the given (weak) vulnerability to corrupt a victim object (e.g., of class S1, denoted as *victim class*) used at the vcall. Specifically, she/he could replace the victim object's `vptr` with a VTable pointer of another class (e.g., class S2, denoted as *counterfeit class*) derived from the base. Further vcalls of this victim object (e.g., S1::func1) will invoke a different virtual function (e.g., S2::func1, denoted as *counterfeit function*). But ABI-conformant vcall protections will not block this out-of-context invocation. Since objects of different classes have different layouts, the counterfeit function may access fields (e.g., *memberM*) outside the victim object, which may corrupt the *relay object* following this victim object. Eventually, the counterfeit function or future functions operating on the relay object will be hijacked.

Two conditions are required to make the attack work. (1) The counterfeit class is derived from the base class expected

[2] Some defenses, e.g., OS-CFI and μCFI, could reduce the size of this set by tracking runtime information, e.g., the origin of pointers.

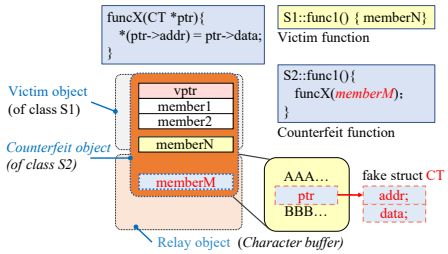


Figure 2: Consequences of out-of-bound data read.

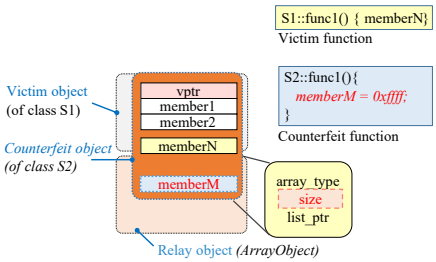


Figure 3: Consequences of out-of-bound data write.

at the virtual call site, to pass the security checks of a C++ semantics aware defense. (2) The counterfeit virtual function performs out-of-bound access on the victim object, to yield exploitable memory safety violations.

Note that, even if the counterfeit function does not cause out-of-bound access, it may corrupt fields of the victim object or cause other unexpected behaviors, and eventually enables exploitation. But it is hard to assess the consequences of all unexpected behaviors in a unified way. Thus, we only consider out-of-bound access in COOPLUS.

3.3 Vulnerability Amplification

With COOPLUS, we could utilize the original vulnerability of limited capability, i.e., which can only tamper with an object's `vptr`, to trigger new out-of-bound memory access on the relay object. Further access to the relay object will cause unexpected behaviors. This new memory violation could amplify the vulnerability's capability, which could even lead to arbitrary address memory writes (AAW), facilitating further exploitation (e.g., control flow hijacking).

Out-of-bound Read. In the first case, the counterfeit function performs an out-of-bound read on the relay object. If the relay object is controllable, then the counterfeit function may misbehave and yield the following four types of gadgets.

If the controllable data loaded (Ld) from the relay object is used by the counterfeit function as a program counter (PC), then it could facilitate control flow hijacking. This type of gadgets is denoted as **Ld-Ex-PC**.

If the controllable data loaded (Ld) from the relay object is used by the counterfeit function as a target memory address to write, then this type of gadgets could cause arbitrary memory write (AW). Depending on the value that can be written to target memory, there are three types.

- **Ld-AW-Const:** the counterfeit function can only write a constant value to target memory. This gadget can be exploited in a limited range of scenarios.
- **Ld-AW-nonCtrl:** the counterfeit function writes a non-constant and non-controllable value to target memory. It could be exploited in a limited range of scenarios.
- **Ld-AW-Ctrl:** the counterfeit function writes a controllable value to target memory. This gadget could facilitate AAW, as shown in the example demonstrated in Figure 2.

If the controllable data loaded from the relay object is used by the counterfeit function as a target memory address to read, then it could cause arbitrary memory read (AAR).

Out-of-bound Write. The other case is that, the counterfeit function performs an out-of-bound write on the relay object. Further operations on the relay object will be misled. Depending on the value written by the counterfeit function, there are two classical types of gadgets.

- **St-Ptr:** the counterfeit function tries to write a pointer value to the relay object. If the relay object could be observed by the adversary, then she/he could leak a pointer to break defenses like ASLR.
- **St-nonPtr:** the counterfeit function tries to write a non-pointer value to the relay object. Depending on how this value is used by the relay object, it may also enable further exploitation. Figure 3 shows an example in which a non-pointer value `0xffff` is stored to the `memberM` field and corrupts the relay object, which interprets the field as a `size` of an array and may lead further AAR or AAW.

Relay Object Manipulation. With proper heap layout manipulation techniques, e.g., heap feng shui [30], we could allocate many types of objects (usually of same sizes) following the victim object, and make them as relay objects to enable potential exploitation paths.

For example, if the counterfeit function causes some out-of-bound read, then a controllable relay object could be allocated to further hijack the counterfeit function, or a non-controllable relay object with sensitive fields could be allocated to leak sensitive information. If the counterfeit function causes an out-of-bound write, then a relay object of the proper layout (e.g., with a size field at proper offsets) could be allocated, to drive the program out-of-control.

3.4 Attack Analysis

3.4.1 Vulnerable Protections

The COOPLUS attack could bypass a wide range of virtual call protections that meet the following two conditions.

- *The target virtual call protection follows the well known C++ ABI.* More specifically, the `vptr` is placed before each object, and the virtual call site allows type-conformant virtual functions.
- *Victim objects' vptr could be corrupted by adversaries.* In other words, the target vcall protection does not guarantee integrity of `vptr`. In practice, following the C++ ABI,

```
1  class Buffer{ int size; char[1024] src_buf;};
2  int input_check(Buffer* obj){
3      uint32_t length = read_uint32();
4      obj->size = length;
5      if (length>1024) return false;
6      read_len(obj->src_buf, length);
7      return true;
8  }
9  int vul_func(Buffer* obj){
10     uint32_t fcs =~0
11     uint8_t *src=obj->src_buf;
12     uint8_t *p=src;
13     while(p!=&src[obj->size]) CRC(fcs,*p++);
14     *(uint32_t *)p=htonl(fcs);//overflow when size>1020
15 }
16 int trigger_func(){
17     Buffer* p = new Buffer();
18     if (input_check(p)){ vul_func(p);}
19 }
```

Listing 1: A four-byte heap overflow (CVE-2015-7504).

```
1  Privilege_Rank *PR;
2  #define administrator 0
3  #define normal_user 1
4  class Privilege_Rank{
5      char* username;
6      uint64_t rank_level;
7      Privilege_Rank(uint64_t rl)::rank_level(rl);
8  };
9  void init_a_thread(){
10     PR = new Privilege_Rank(normal_user);
11 }
12 void sensitive_operation()
13 {
14     if (PR->rank_level==administrator){
15         system("/bin/sh");
16     }else{
17         do_nothing();
18     }
19 }
```

Listing 2: Privilege escalation primitive

`vptr` is associated with objects which reside in writable heap, making it challenging to protect its integrity.

- *Multiple targets are allowed at virtual call sites.* Some defenses are able to limit the number of allowed runtime virtual functions to 1. It leaves no space for exploitation.

Some defenses, e.g., CFIXX [24], breaks the C++ ABI and replaces each `vptr` with a runtime lookup table entry, then protects the integrity of this table with the Intel MPX [31] hardware feature. This type of defense could defeat COOPLUS, but introducing compatibility issues.

Some defenses, e.g., OS-CFI [2] and μCFI [8], could track runtime information to limit the number of allowed virtual functions (even to 1 in some cases). However, they are in general hard to deploy in practice. Moreover, they cannot guarantee a unique runtime target for each virtual call in practice [32]. So, in theory, COOPLUS is still feasible.

3.4.2 Applicable Vulnerabilities

Proposed exploiting techniques in literature, in general, assume applications have vulnerabilities with strong capability, e.g., enough to make many powerful exploitations. But in reality, such qualified vulnerabilities are rare. On the contrary, COOPLUS has a lower expectation on the vulnerability and is applicable to many real world vulnerabilities, as long as the vulnerability can (even partially) corrupt the `vptr`.

For example, a heap overflow vulnerability which could overwrite only one byte is qualified. A use after free vulnerability is also qualified. Listing 1 shows a heap overflow vulnerability which only overwrites the following four bytes with a CRC checksum. Given that CRC could be reversed [33], the adversary could utilize this vulnerability to overwrite 4 bytes (`vptr`) with an arbitrary value.

A weak vulnerability that can only partially overwrite a `vptr` could be exploitable as well. Since the victim class and counterfeit class are often defined in the same program module, thus have VTables close to each other. So, a partial overwrite to one `vptr` could yield another compatible `vptr`, and enable the COOPLUS attack. But, *such weak vulnerabilities will reduce the number of available exploit primitives*, and thus lowers the probability of being exploited.

3.4.3 Attack Effects

The major attack effect of COOPLUS is arbitrary address write or read (AAW or AAR). On one hand, AAW and AAR are the basic assumptions of most exploitation techniques, which could be further utilized to perform kinds of advanced attacks including control flow hijacking. On the other hand, AAW and AAR vulnerabilities are rare in practice. Therefore, COOPLUS provides a robust solution to get AAW and AAR primitives, facilitating many exploits.

Furthermore, in some cases, COOPLUS cannot be utilized to get AAW or AAR primitives. But it could be exploited as well. Take the code in Listing 1 as an example, assuming we could only find one counterfeit function, which only doubles a data field of the relay object, we could still utilize it to launch control flow hijacking attacks. As shown in Listing 2, there is a sensitive operation at line 15, which will only be executed with proper `rank_level`. As a result, we could allocate a `Privilege_Rank` object as the relay object, and utilize COOPLUS to double the `rank_level` field. By launching this attack multiple times, we could overwrite this field to 0, and launch the sensitive operation.

4 Primitive Generator

Given a target application, a vulnerability, and a virtual call protection, we would like to assess whether the vulnerability could be exploited to launch the COOPLUS attack and bypass the deployed protection. However, it is challenging.

To launch a COOPLUS attack, we have to first find a proper tuple of exploit primitives *(virtual call, victim class, counterfeit class)*, where (1) the virtual call invokes a virtual function declared in a base class, (2) the victim class and counterfeit class are derived from the base class but have different virtual function implementations, and (3) the victim object has to be corrupted by the vulnerability. Finding such a tuple in the target application, especially in one that has a large code base, is a heavy task. Furthermore, we have to

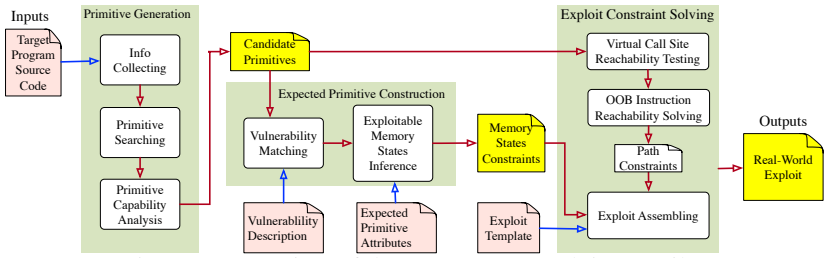


Figure 4: Overview of the COOPLUS exploit compiler: VScape.

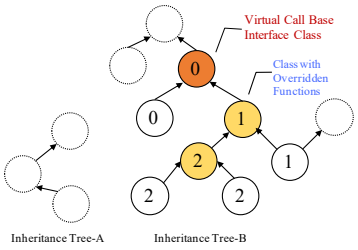


Figure 5: An example CIH and ranks.

generate proper inputs to trigger the virtual call, then trigger the out-of-bound memory access in the counterfeit function, and eventually, trigger sensitive operations on the corrupted data fields, requiring great efforts too.

Therefore, we present a solution VScape to automatically compile candidate primitives and filter practical and reachable primitives, which further facilitates generating the final exploit to bypass the target virtual call protection.

4.1 System Overview

Figure 4 shows the overview of the VScape compiler. It consists of three major components: primitive generation, expected primitive construction, and exploit constraint solving.

The *primitive generation* component takes source code of target applications as inputs, then searches for candidate vcall primitives that can bypass defenses and analyzes their capabilities, and outputs these candidate primitives.

Then, the *expected primitive construction* component takes (1) description of the given vulnerability (e.g., vulnerable object's size and affecting ranges), and (2) expected exploit primitive attributes (e.g., write an arbitrary value to a specific address) as inputs, and outputs qualified candidate primitives, together with memory sates (e.g., certain fields of the relay object take specific values) which could make such primitives work. With such information, security analysts could get desired primitives and compose exploits quickly.

The *exploit constraint solving* component further resolves certain constraints (e.g., path reachability) to make chosen primitives working in the final exploit. VScape takes a user-provided exploit template as an extra input to form a full chain exploit, and outputs the final working exploit.

Currently, there are no solutions able to automatically construct exploit templates, except for simple vulnerabilities like stack-based buffer overflow. Thus, VScape relies on analysts to schedule exploit steps, i.e., preparing an exploit template, and provide knowledge of the vulnerability and what types of primitives are needed by the template. VScape could search for qualified primitives and complete the working exploits. Note that, it is common for modern AEG solutions, e.g., Gollum [12], Revery [11] and BOPC [34], to have user-supplied exploit templates (in different forms) to assist AEG, since a full chain exploit needs to address many challenges that are out of scope (e.g., heap manipulation).

4.2 Primitive Generation

4.2.1 Data Collection

The first step of VScape is collecting virtual call related information during compilation, including compatible classes and different virtual functions implementations. Specifically, VScape collects three types of information as below:

- **Virtual call sites:** COOPLUS bypasses virtual call protections around certain virtual call sites. Therefore, VScape first logs all virtual call sites in the target application, as well as the expected virtual function's statically declared base interface class information.
- **Class layouts:** The victim class and counterfeit class in an exploit primitive all derive from the virtual call's interface class. Therefore, VScape also logs the class layout information during compilation, including its *size, offsets of member fields and base classes*. Note that, the final exploit also relies on relay objects, which may have no virtual functions. So VScape will log all class layouts no matter the class has virtual functions or not.
- **Virtual functions:** VScape logs all type-conformant virtual functions for each virtual call site, i.e., those overridden in classes derived from the interface class. Further, VScape logs the maximum field access offset of each virtual function when generating code for the function, since it has to find potential out-of-bound memory access later.

4.2.2 Primitive Searching

For a given virtual call, we need to find proper victim classes and counterfeit classes, which have different implementations of the target virtual function. Such a pair of functions could yield unexpected behaviors and enable COOPLUS exploitation, thus forming a candidate exploit primitive.

Since the victim class and counterfeit class all derive from the interface class expected in the virtual call, we could first build the class inheritance hierarchy (CIH) tree based on the class layout information we collected, as shown in Figure 5, then search the tree for derived classes that have different implementations of the target virtual function.

More specifically, VScape checks the implementations of the target virtual function in all derived classes. A breadth-first search (BFS) algorithm is applied to iterate all derived classes, starting from the base interface class. A global rank number (starting from 0) is maintained to record versions of the target virtual function. Each time a parent class is iterated, each of its child class will be assigned with a rank

number. If the child class inherits the implementation of this virtual function, then the rank number of the parent class is assigned to this child class. Otherwise, the global rank number increases by 1 and is set to the child class.

Finally, any two classes with different rank numbers, together with the virtual call, could form a candidate exploit primitive *(virtual call, victim class, counterfeit class)*.

4.2.3 Primitive Capability Analysis

As discussed in Section 3.3, different virtual call primitives have different capabilities. VScape further analyzes each primitive to understand its capability. Specifically, it first determines whether the out-of-bound (OOB) memory access in the counterfeit function is a write or a read operation.

For an OOB read, it then analyzes how the loaded value is used in the counterfeit function, i.e., whether it is used as a program counter or a target memory address to write. For the latter case, VScape will further analyze whether the written value is controllable by the adversary, via taint analysis.

For an OOB write, it then analyzes whether the value written to the relay object is a pointer value. If yes, VScape further looks for potential information leakage locations (users of the relay object) to bypass ASLR.

In this way, VScape could determine the capability of each primitive, i.e., Ld-Ex-PC, Ld-AW-Const, Ld-AW-nonCtrl, Ld-AW-Ctrl, St-Ptr, and St-nonPtr. Note that, one primitive could have multiple capabilities, depending on the functionality of the counterfeit function and users of the relay object.

4.3 Expected Primitive Construction

Given all candidate primitives and their capabilities, VScape further selects appropriate ones which can cooperate with the given vulnerability and satisfy expected primitive attributes.

4.3.1 Vulnerability Matching

Given a vulnerability, not all candidate primitives could be invoked. Specifically, the vulnerable object where the vulnerability occurs has to be allocated in the same heap as the victim object of the candidate primitive.

For instance, if there are multiple heap allocators responsible for allocating different objects, or the sole heap allocator puts objects of different types or different sizes into different zones, then the vulnerable object cannot influence the victim object, and the corresponding primitive will not work.

Given the vulnerability description input, VScape learns expert knowledge of the heap allocators, and then matches candidate primitives with the target.

4.3.2 Exploitable Memory States Inference

In an exploit, the vcall primitive has to serve a specific purpose, e.g., write a specific value to a specific address. In order to serve such purposes, which are defined as input expected primitive attributes, the candidate vcall primitive has to run in a specific memory state, e.g., certain fields in the victim object have to take specific values.

VScape could automatically infer such memory state requirements for a candidate vcall primitive via taint analysis and symbolic execution. Given a candidate primitive, i.e., a virtual call site, a victim function and a counterfeit function, VScape will mark the victim object and the adjacent relay object as symbolic values, and symbolically executes the counterfeit function which will access the relay object.

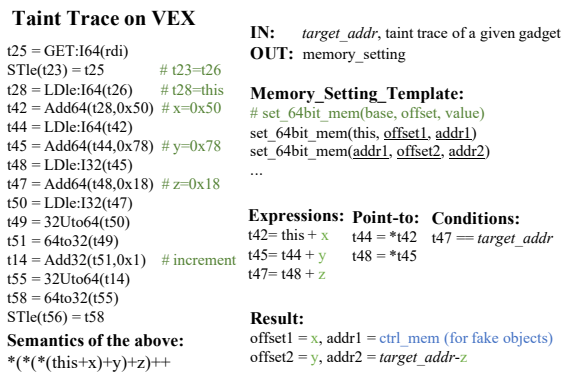


Figure 6: An example memory state inference, for a primitive with the Ld-AW-Ctrl capability.

For instance, Figure 6 shows a primitive with the capability Ld-AW-Ctrl, and the expected primitive attribute is that this primitive should write to a specific address *target_addr*. By performing symbolic execution on the taint-related trace of the counterfeit function, we could infer that, the adversary needs to set the field at offset `x` with a pointer to a *fake object* crafted by the adversary, and set the field at offset `y` of this fake object with *target_addr - z*.

4.4 Exploit Constraint Solving

So far, the candidate primitives are retrieved via static analysis. It is not clear whether such primitives could reveal expected behaviors at runtime.

Given a candidate primitive *(virtual call, victim class, counterfeit class)*, there are three specific questions to answer: (1) Given that not all data flow is feasible at runtime, whether the victim object will be used at the vcall site? In other words, whether the victim function could be invoked at runtime? If not, the counterfeit function will not be invoked either. (2) Given that the counterfeit function has many program paths, whether the OOB memory access instruction could get executed at runtime? If not, the unexpected memory safety violation will not happen. (3) If both answers are yes, what data constraints should be met in order to trigger the victim function and the OOB access instruction?

4.4.1 Reachability of Victim Functions

Directed fuzzing [35] is a straightforward solution to evaluate the runtime reachability of the target function or instruction. However, during the experiment, we figured out the efficiency of existing directed fuzzing solutions is low at exploring reachable targets when there are hundreds of targets in a relatively large application.

As a result, we skip evaluating the reachability of every victim function. Instead, *we only try the best to get an incomplete list of reachable victim functions*, and discard re-

```
1  def main():
2      heap_operation_before_relay_object()
3      gen_relay_object_and_fake_object()
4      heap_operation_before_victim_object()
5      gen_allocate_victim_object()
6      vul_trigger()
7      gen_invoke_counterfeit_function()
8      operations_after_cooplus()
9
10 # Prepare memory for the expected primitive
11 def gen_relay_object_and_fake_object():
12     ''' set_memory(relay_base, offset_1,value_1) '''
13     ''' set_memory(fake_base, offset_2,value_2) '''
14     ...
15 # Ensure victim function's reachability
16 def gen_allocate_victim_object():
17     from PyQt5.QtCore import Qt
18     from PyQt5.QtWidgets import QWidget,QApplication
19     window = QWidget()
20 # Ensure OOB instruction's reachability
21 def gen_invoke_counterfeit_function():
22     window.show()
```

Listing 3: An example exploit template for PyQt.

maining victim functions (although some of them could be reachable). Specifically, we utilize dynamic testing to evaluate target applications with given benchmark test cases, and collect victim functions that are triggered during testing.

Specifically, VScape inserts callback handlers at virtual call sites of each candidate primitive. During testing, the callback handler will log the invoked victim function and the corresponding test case.

4.4.2 Reachability of OOB Instructions

Given a reachable victim function of a candidate primitive, we could launch COOPLUS to execute the counterfeit function. However, the out-of-bound access operation in the counterfeit function may not get executed at runtime, since this function may have multiple paths.

VScape utilizes symbolic execution to infer whether the OOB access instruction is reachable and under what condition it is reachable. More specifically, it takes the logged test case that reaches the victim function as input, and dumps the runtime context when the victim function is hit, and then feeds it to the symbolic execution engine Angr [19].

Starting from the dumped context, Angr begins concolic execution on the counterfeit function (rather than the victim function). The relay object is marked as symbolic values, since it could be controlled by the adversary via heap manipulation. Angr will explore all paths of the counterfeit function and verify whether the OOB instruction is reachable. If yes, it outputs the path constraints (e.g., a specific memory state) that should be satisfied by related objects.

4.4.3 Exploit Assembling

Generating an exploit in practice is extremely challenging, both for humans and machines. There are many open challenges in automated exploit generation [36]. VScape is not able to generate full chain exploits automatically neither.

VScape also relies on a user-provided exploit template to compose a full chain exploit. Specifically, several manual steps are required in the template, including (1) manipulate the heap layouts of the target application, to arrange victim objects and relay objects; (2) reform the vulnerability POC to tamper with `vptr` of the victim object with a proper value, and (3) utilize the capability provided by the COOPLUS primitives to launch final exploits. For the first step, there are several draft solutions to assist heap layout manipulation, e.g., SLAKE [37], SHRIKE [38] and Gollum [12]. However, they are still in an early stage. For the second step, symbolic execution is a potential solution. But it requires great engineering efforts and faces the scalability challenge. For the last step, many well-known exploit patterns are required to assist the exploitation. For instance, the adversary could utilize AAW to overwrite the global offset table or other function pointers to hijack the control flow. We leave the automation of these steps to future work.

Listing 3 shows an example exploit template for PyQt. Operations at line 2, 4, 6, and 8 represent the aforementioned manual steps, where operations at line 3, 5, and 7 could be automatically done by VScape. Specifically, at line 3, VScape infers the memory state in which the vcall primitive has to run. At line 5, VScape builds the victim object from a logged test case and ensures the reachability of the victim function. At line 7, VScape ensures the counterfeit function is invoked and the OOB instruction is executed.

5 Evaluations

To evaluate the effectiveness of COOPLUS attack and VScape, we designed several experiments and tried to answer the following questions:

- **RQ1:** What is the popularity of COOPLUS exploit primitives in real world C++ applications?
- **RQ2:** Is the COOPLUS attack effective at defeating various virtual call protections?
- **RQ3:** Is VScape effective at generating exploit primitives and assisting full chain exploit generation, when given real world vulnerabilities?

5.1 Implementation

We implemented a prototype of VScape. It consists of (1) a compiler plugin based on Clang [53] and LLVM [54] to collect virtual call related information, (2) a primitive searcher which finds candidate primitives and analyzes their capabilities based on the VEX IR [55], (3) an expected primitive constructor which finds matching primitives and required memory states, and (4) an exploit constraint solver which adopts lightweight dynamic tests and symbolic execution based on Angr [19], to filter reachable victim functions and solve memory states that can reach target OOB. The code size of each component is listed in Table 2.

5.2 Attack Surface Analysis

To answer the question RQ1, we evaluated VScape on 14 open source C++ programs, which are widely used and actively maintained. All programs are compiled with default

Table 1: Statistics of virtual functions, virtual call sites, and COOPLUS exploit primitives of 14 C++ applications.

Category	App	Version	LoC	Virtual Functions	Virtual Call Sites	Unique Virtual Call Sites (UVC)			VFunc Variants (Ranks) for #UVC-CVF			#UVC-OVF	VFunc Variants (Ranks) for #UVC-OVF					All Primitives
						All	#UVC-CC	#UVC-CVF	*All*	μ	*Max*		*All*	μ	σ	*Med*	*Max*	
Browser	**firefox** [39]	50.1.0	1,062,487	84,753	101,116	25,224	18,874 (74%)	2,279 (9%)	12,480	5.5	627	969	3,432	3.5	12.8	2	389	83,786
	chromium [40]	77.0.3864.0	3,670,688	171,373	322,583	61,315	34,371 (56%)	7,205 (12%)	30,532	4.2	1,124	3,741	11,80	3.2	16.7	2	978	535,007
Multi-media Tech	oce [41]	0.11	1,979,905	18,097	29,945	3,738	1,877 (50%)	609 (16%)	7,188	11.8	3,323	303	1,123	3.7	4.1	2	62	4,040
	Bento4 [42]	1.5.1.0	77,050	935	1,879	253	141 (55%)	43 (17%)	264	6.1	77	31	152	4.9	7.4	3	44	1,140
	ImageMagick [43]	7.0.8	540,190	294	40	10	7 (70%)	2 (20%)	118	59.0	59	1	18	18.0	0.0	18	18	153
	exiv2 [44]	0.27.1	367,780	908	3,041	300	163 (54%)	36 (12%)	177	4.9	13	19	68	3.6	2.2	2	8	134
	opencv [45]	4.1.2	1,352,028	36,855	28,569	9,183	883 (9%)	182 (2%)	2,907	16.0	160	86	1,216	14.1	33.1	2	157	55,116
	qt [46]	5.12.0	26,292,89	27,590	28,601	6,764	4,730 (69%)	1,662 (25%)	14,027	8.4	2,015	840	4,468	5.3	34.5	2	751	508,141
	aGrum [47]	0.16.3.9	406,787	2,597	33,028	1,006	304 (30%)	36 (4%)	92	2.6	6	8	23	2.9	1.1	2.5	5	26
	SLikeNet [48]	0.2.0	1,062,487	445	1,924	308	135 (43%)	29 (9%)	147	5.1	25	11	79	7.2	7.3	2	24	538
Network & Server	Bitcoin [49]	0.18.1	262,693	2,142	5,875	400	246 (61%)	33 (8%)	100	3.0	7	25	64	2.6	1.0	2	5	62
	znc [50]	1.8.0	26,951	761	1,412	257	225 (87%)	85 (33%)	394	4.6	36	28	73	2.6	1.7	2	11	99
	mongodb [51]	4.3.2	4,755,978	17,025	22,171	4,176	2,738 (65%)	406 (10%)	2,387	5.9	230	206	577	2.8	1.8	2	17	865
Others	openbabel [52]	3.0.0	206,855	2,220	2,569	466	234 (50%)	66 (14%)	674	10.2	121	31	136	4.4	3.8	3	21	455

#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, σ: Standard deviation of VFunc Variants.

Table 2: Implementation of VScape

Component	Language	LoC
Customized Compiler	C++	2097
Primitive Searcher	Python	5209
Expected primitive construction	Python	822
Exploit constraint solving	C++, Python	1118
Total	C++, Python	9246

configurations. In order to replay vulnerabilities found several years ago, we conducted experiments in the outdated Ubuntu 16.04 system.

5.2.1 Popularity of Virtual Calls

Table 1 shows the statistics of virtual functions and virtual call sites of each application. From the fifth and sixth columns, we can see that: All applications have hundreds of virtual functions, while Chromium has over 171 thousands of virtual functions. Moreover, all applications except ImageMagick have thousands of virtual call sites. It shows that *polymorphism is very popular in C++ applications.*

We further analyze those virtual call sites in detail. First, different virtual call sites may invoke the same virtual function, i.e., the same function declared in the same base class. From the perspective of COOPLUS, different virtual call sites expecting the same virtual function could be exploited in the same way. So, we deduplicate the virtual call sites, and count the number of unique virtual call sites (UVC) in column 7.

Then, given a virtual call site, it expects a virtual function declared in a base interface class, and any overridden virtual function implemented in a derivation of the base class is allowed. However, there are two special cases in which only one virtual function exists: (1) the base interface class does not have any derivations, and (2) all derivation classes do not override the implementation in the base class. Therefore, we remove UVCs that satisfy the first condition and list the remained count in column 8, and remove UVCs that satisfy the second condition and list the remained count in column 9. These UVCs form the basis of COOPLUS.

For instance, in the application Chromium, there are over 61 thousands of UVCs, but 44% (=1-56%) of them have only one compatible class (i.e., no derivations), another 44% (=56%-12%) of them have multiple compatible classes but none of them override the target virtual function expected at the UVC, and only 12% of them actually have multiple compatible functions. In other words, about 88% (=1-12%) of these UVCs only have one candidate virtual function to invoke in the whole application, and *therefore could be optimized with the devirtualization technique* [56]. It also implies that, *developers tend to use polymorphism*, even if no derivations are implemented in the current version of code.

5.2.2 COOPLUS Exploit Primitives

For a UVC, if it has multiple compatible functions, then it is a candidate that COOPLUS could utilize to bypass the deployed defense. Column 10-19 in Table 1 shows the detail statistics of candidate exploit primitives in each application.

For UVCs with multiple compatible virtual functions, the average number of compatible functions ranges from 2.6 (aGrum) to 59 (ImageMagick), as shown in the fourth column. Further, the maximum number of compatible virtual functions ranges from 6 (aGrum) to 3323 (oce), as shown in column 12. This number roughly implies the complexity of the class inheritance hierarchy (CIH) tree of the application.

Further, since COOPLUS only works for virtual functions that could cause out-of-bound (OOB) access on objects of compatible classes, we also count the number of UVCs that have at least one pair of compatible functions with OOB access operations, and list the data in column 13. For these filtered UVCs, we also count their numbers of compatible virtual functions in column 14, 15, 16, 17 and 18. From the median and the standard deviation, we can see the number of compatible virtual functions are not spread evenly. Mostly, we can only find a pair of them. But even so, we can find abundant virtual functions for some UVCs. Lastly, the number of candidate COOPLUS exploit primitives is listed in the last column (i.e., column 19).

Further, VScape analyzes each primitive to understand its capability. Details can be found in Appendix A.2. For feature-rich applications, e.g., firefox and opencv, hundreds of primitive gadgets are found. Especially, there are over 5,360 useful COOPLUS gadgets recognized in chromium (shown as Table 4), implying a large attack surface is available for adversaries to bypass potential defenses. Therefore, we can conclude that, COOPLUS *exploit primitives are very popular in C++ applications* (answers to RQ1).

Table 3: Effectiveness of CFI solutions against COOPLUS

Category	CFI Scheme	Granularity	Realization	Theoretical Basis	Effective against COOP*	Effective against COOPLUS
ABI incompatible	CFIXX [24]	-	Source code + MPX [57]	Object integrity	✓	✓
Validity check (with runtime context)	μCFI [8]	Unique	Source code + Intel PT [58]	Path-sensitive	✓	✓
	OS-CFI [2]	Fine	Source code + MPX [57] + TSX [59]	Source-sensitive	✓	✗
Validity check (with C++ semantics)	MCFI [60]	Fine	Source code	Type-based	✓	✗
	πCFI [61]	Fine	Source code	Context-sensitive	✓	✗
	CFI-LB [62]	Fine	Source code + Intel PIN [63]	Call stack based	?	✗
	SafeDispatch* [64]	Fine	Source code	Type-based	✓	✗
	LLVM-CFI [6]	Fine	Source code	Type-based	✓	✗
Generic CFI (Example targets in COOP)	CCFIR [4]	Coarse	Binary	-	✗	✗
	binCFI [3]	Coarse	Binary	-	✗	✗
	LockDown [65]	Coarse	Binary	-	✗	✗

?: CFI-LB has an implementation flaw which makes it fail to defeat COOP. This flaw has also been confirmed by [32].
*: SafeDispatch is not open-source, we evaluate it based on a reproduction work [66].
⋆: Here, we refer COOP to the one claimed in the original paper [5], excluding the variant COOPLUS.

5.3 Test against CFI Solutions

To answer the question RQ2, we further evaluate the effectiveness of 12 virtual call protections against COOPLUS.

5.3.1 Experiment Setup

We crafted a vulnerable benchmark [67], and hardened it with 11 CFI defenses respectively, to evaluate their effectiveness against COOPLUS. Note that, we did not choose large applications like browsers as targets to evaluate, for the following reasons. First, few proposed CFI solutions can be deployed to real world large applications without compatibility issues. For instance, the Clang-CFI [6] fails to compile Firefox due to cross module support. Second, a crafted benchmark is easy to exploit and to validate, since no heap layout manipulation or other advanced exploit skills are required. Third, the evaluation result drew from the crafted benchmark is the same as the result from real world applications, in terms of defenses' effectiveness against COOPLUS.

5.3.2 Result Analysis

Table 3 shows the evaluation results of these defenses. It confirmed that CFI approaches that do not consider C++ semantics ([4], [3], [65]) are all vulnerable to COOP [5] and COOPLUS. The original paper [5] claimed COOP can be reliably prevented when precise C++ semantics are taken into consideration. We believed this is not correct. As the results showed, one variant of COOP, i.e., COOPLUS, successfully bypasses all defenses except CFIXX [24] and μCFI [8] (answers to RQ2).

The defense CFIXX places `vptr` in a separate integrity-protected table, so that the adversary cannot overwrite it to launch COOPLUS. But CFIXX breaks the C++ ABI and may cause compatibility issues in some applications.

The CFI defense μCFI takes runtime data flow information into consideration, and could identify the unique target for each indirect call (including virtual call) in most cases. Essentially, it provides data integrity protection to `vptr` to certain extents. Thus, it is able to defeat COOPLUS in most cases. But it requires Intel PT and a separate process to monitor data, making it hard to deploy in practice. Another CFI solution OS-CFI also takes runtime context into consideration, but could be bypassed by COOPLUS in some cases, due to some trade-offs in its implementation [32].

For all other CFI solutions, including C++ semantic aware ones (e.g., MCFI [60]), they all can be bypassed by COOPLUS, since they (1) keep the C++ ABI, (2) cannot protect the integrity of `vptr`, and (3) allow more than one targets at virtual call sites.

Therefore, to fully mitigate COOPLUS, a solution which protects the integrity of `vptr` with a low performance overhead and good compatibility is demanded.

5.4 Exploit in Practice

To answer the question RQ3, we evaluated VScape on Mozilla Firefox 50.1 (64-bit) and Python-3.6.7 with PyQt-5.12 library in a Linux x64 operating system.

These two applications both have OOB vulnerabilities [68, 69] and large numbers of primitives for the COOPLUS attack. Two key factors affecting exploit success rates are (1) whether these primitives are reachable (i.e., could be invoked by users), and (2) how these primitives can help amplify the vulnerability to acquire more powerful capabilities. VScape will help with these analyses. Given an exploit template taking care of the rest AEG challenges, we finally synthesize expressive exploits for the targets with VScape.

5.4.1 Attack Surface Analysis

We analyzed Firefox and PyQt with VScape, and demonstrated the analysis results in Figure 7 and 8.

After recovering class inheritance hierarchy trees, we get 2,279 unique virtual call sites (UVC) that have multiple candidate virtual functions in Firefox (1,662 in PyQt). Then, after performing *primitive search*, we can filter out UVCs that do not have OOB virtual function pairs, and get 969 and 840 UVCs in Firefox and PyQt respectively. For each UVC, there could be multiple virtual function pairs with OOB behaviors, and thus we could get multiple primitives. As shown in Figure 7, there are 83,786 and 508,141 primitives respectively.

Further, we perform the reachability testing, to get an *incomplete* set of victim functions and their UVCs. Thus, we get 180 and 220 reachable UVCs, together with 1665 and 2299 primitives respectively. Furthermore, we match these primitives with given vulnerabilities (CVE-2018-5146 and CVE-2014-1912), and get 12 and 16 qualified UVCs. Lastly,

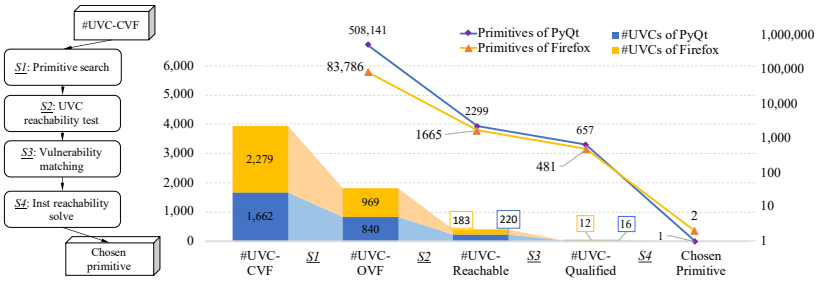


Figure 7: The number of candidates descends along various analyzing stages.

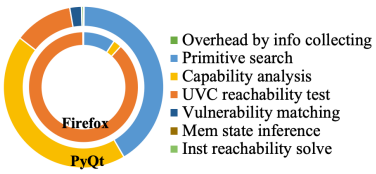


Figure 8: Time cost distribution of each analysis phase. The inner ring is for Firefox, the outer is for PyQt.

we assess the reachability of target OOB instructions, and find one (incomplete set) UVC in these two applications, together with 1 and 2 primitives respectively.

Figure 8 shows the time cost distribution of different analysis steps. For Firefox, UVC reachability testing took the most time, which tested 43,463 test cases from the Firefox project. But for PyQt, we only collected 330 test cases to perform reachability testing. In contrast, VScape spent most of the time in primitive search and capability analysis, which are the main steps to locate the attack surfaces.

5.4.2 Case Study

Due to the page limit, we only present the case study for Mozilla Firefox 50.1 (64 bit) here, and put the case study for PyQt in Appendix A.1.

For Firefox, we used CVE-2018-5146 [69] to demonstrate the attack. This vulnerability is an out-of-bound write with controllable value, which occurs while processing Vorbis audio data with Libvorbis. But the OOB write only affects objects in jemalloc heap [70], separated from easily-controllable JS Objects in Nursery or Tenured memory.

The complicated memory management in Firefox increases the difficulty of exploitation. Controllable JS objects in Firefox are managed by generational garbage collector (GGC) [71], while victim objects (C/C++ objects) qualified for COOPLUS are allocated on the jemalloc heap. Only if the size of a JS object exceeds a certain limit, it will be moved to the jemalloc heap. Moreover, the jemalloc allocates objects in different *runs* with respect to their sizes. So the constraints of object sizes should be considered.

Amplification Strategy. The vulnerability CVE-2018-5146 [69] exists in libvorbis is related to the procedure of decoding ogg data to PCM data. A boundary check is missed in a nested loop, leading to an out-of-bound increment memory in the native heap (jemalloc in this case). And the size of the vulnerable object is adjustable. Since the key instruction in PoC is a floating add, we need to know the original value of `vptr`, then we can replace it with `vptr` of counterfeit class.

Among all types of gadgets, St-nonPtr is the most popular (as shown in Table 4). In most cases, the counterfeit function tries to write a boolean value into OOB area. For COOPLUS, this helps attackers to write exception value, zero or one, into relay objects. If we can manipulate Hi address byte, pointers are very possible to be corrupted and re-pointed to addresses out of the memory segment. When the corrupted pointer value locates in unmapping memory, we can take full control over this range with elaborated heap spray.

In this way, we build a complete controllable faking object in this area. If the faking object contains metadata underlying memory read and write, it can also be used for AAR and AAW in exploitation. Specifically, we counterfeit objects of JSString, modify the data pointers and leak memory in arbitrary addresses. Then with sufficient leaks, we can make it easy to counterfeit complicated objects like ArrayBufferObject and TypedArrayObject, and write arbitrary bytes into target addresses. We find some qualified objects which live across heap managers - the data list for ArrayObject will be moved to jemalloc from Nursery and Tenured when its size grows larger than 128 bytes.

With the heap manager deployed in Firefox, we have to search vulnerable objects, victim objects, and relay objects in the same size range. We choose the relay object whose size must exceed 128 bytes. Thus, to meet the requirements for heap layouts, we can only select victim classes in a primitive database whose size is big enough.

Primitive in Exploit. According to above, we order VScape with several rules (1) the size of victim class exceed 128, (2) the offset of victim member variable *off* mod 8 > 1 (Hi address byte) and (3) the primitive has capabilities of St-nonPtr. The number of matched candidate UVC is 71. And 12 of them are triggered in reachability tests. Then, the primitive tuple that we select is (Animation::UpdateTiming(), Animation, CSSAnimation) in the namespace of mozilla::dom.

Before composing a real exploit, we emulate a PC hijack toward the counterfeit virtual function, and implement symbolic execution by Angr, to assess whether the target instruction is reachable in the assumption that the memory of relay objects is controllable. In this case, the instruction for OOB-Writing only executes when a variable named *mNeedsNewAnimationIndexWhenRun* is not null, which is exactly an overwritten variable in the relay object. And the condition does not conflict with the supposed gadget who is going to zero the boolean type variable.

Specifically, Firefox provides Web Animation APIs for users to describe animations on DOM elements. When we declare animation config with javascript code, corre-

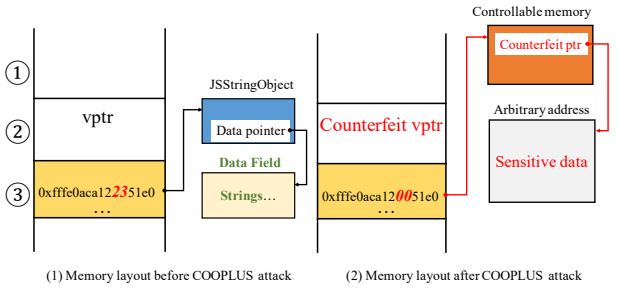


Figure 9: COOPLUS exploit primitives for Firefox. ①vulnerable object - float list ②victim object - mozilla::dom::Animation ③relay object - item list of ArrayObject.

sponding Animation objects will be allocated on jemalloc heap during page rendering. CSSAnimation is a sub class of Animation, the counterfeit function CSSAnimation::UpdateTiming() tries to zero the boolean variable *mNeedsNewAnimationIndexWhenRun*.

Exploit Synthesis. The object size of Animation is 256 bytes. Thus, such objects will only be found in *runs* for 256 bytes. To create the required heap layout, the vulnerable object (float list) and the relay object (data list for ArrayObject) are all modified into the same size.

As shown in Figure 9, after triggering the counterfeit function, we can tamper with the NaN-boxing pointer of ArrayObject's item. When the pointer is redirected toward controllable memory, we use a counterfeit JSStringObject to get AAR. Then with similar technique, replacing counterfeit JSStringObject with a counterfeit TypedArrayObject, AAW is as well achieved.

6 Related Work

6.1 CFI-Oriented Attacks

Researchers have proposed a number of practical yet imprecise CFI solutions. Although these coarse-grained CFI solutions can significantly reduce the attack surface, multiple attacks [72–74] have been proposed to bypass these CFI solutions, by exploiting the fact that the size of equivalence class (EC) for each ICT is still large.

To defeat attacks against coarse-grained CFI solutions, researchers also proposed fine-grained CFI solutions. However, as sound and complete pointer analysis is unfortunately undecidable, fine-grained CFI solutions rely on sound but incomplete pointer analysis in practice, providing conservative over-approximate results and enabling potential attacks. For example, *Control Jujutsu* [75] shows that common software engineering practices force points-to analysis to merge several equivalence classes. Imprecise ECs are large enough for arbitrary computation, to enable an attacker to execute arbitrary malicious code even when fine-grained CFI is enforced. Control-Flow Bending [27] goes one step further and shows that CFI solutions with ideal point-to analysis results are still vulnerable. Some other attacks target implementations of specific CFI solutions. For example, StackDefiler [28] exploits the defect in detail design of IFCC and VTV [6] to realize successful hijack in Chrome.

Recently proposed CFI solutions (e.g., πCFI [61], OSCFI [2] and μCFI [8]) utilize runtime context information to reduce the size of EC, providing better defenses against these attacks. These solutions provide data flow integrity to a certain extent, but in general, are hard to deploy in practice.

COOP [5] first used counterfeit objects to enable Turing-complete malicious computations. But it is wrongly declared that COOP can only circumvent CFI solutions that are not aware of C++ semantic. Instead, one variant of COOP, i.e., COOPLUS, is able to bypass virtual call protections that are C++ semantics aware but neither break the C++ ABI nor protect the integrity of `vptr`, even when only weak vulnerabilities are given.

6.2 Automated Exploit Generation

Automated exploit generation (AEG) can be used to assess the exploitability of vulnerability by generating an exploit. Since David et al. proposed automatic patch-based exploit generation (APEG) [76], AEG [9–11, 16, 77] has become a research focus in recent years.

Representative techniques include AEG [9], Mayhem [10], Q [14] and CRAX [15]. These AEG solutions share a similar workflow. In general, they will first analyze vulnerabilities in detail along with a crashing path, then search for exploitable states, collect vulnerability and exploit constraints respectively, and finally generate exploit inputs.

Repel [78] shows examples to exploit heap-based vulnerabilities with symbolic executions starting from crash points. PRIMGEN [79] automatically counterfeits fake objects to obtain exploit primitives in Web browsers. A key limitation in these AEG solutions is that they only focus on analyzing one single program state in crashing paths. Recently, FUZE [80] and Revery [11] use fuzzing to explore more exploitable states. Note that, all such solutions are not fully automated, and still require expert knowledge or annotations. Gollum [12] first completes an end-to-end AEG system from primitive extraction to heap layout inference in user space.

Another key challenge is heap layout manipulation. ARCHEAP [81] presents an automatic tool to systematically discover the unexplored heap exploitation primitives for specific heap allocators. RELAY [82] simulates human exploitation behavior for metadata corruption and solves layout problems according to the exploit pattern. And HEAPG [83] automates multi-hop exploitation for heap-based vulnerability via known techniques of ptmalloc. These three studies help synthesize exploits in CTF challenges but do not help much when composing a heap-based exploit in real world.

Gollum [12] applies a genetic algorithm to solve this problem and accelerates the performance of the random search algorithm proposed in SHRIKE [38]. Another work SLAKE [37] extracts heap operations and obtains desired slab layouts based on the specific knowledge of kernel heap allocator.

Although some of AEG solutions show their effectiveness in real applications, none of them have taken modern defenses into consideration. For instance, Revery [11] and Gollum [12]) focusing on exploiting heap vulnerabilities only works well when the defense ASLR [13] is disabled. AEG solutions targeting stack-based buffer overflow, e.g., Q [14], CRAX [15] and PolyAEG [16], cannot bypass stack canary [17]. Compared with these AEG solutions, VScape is able to generate exploits to bypass virtual call protections and evaluate COOPLUS with real CVE cases.

DOP attacks. A generalized form of data-only attacks is *Data Oriented Programming* (DOP) [84]. Since DOP does not tamper with control flow, it is outside the scope of most CFI solutions. Data Flow Integrity (i.e., [21, 22]) is a popular defense against DOP. The work [85] has developed a semi-automated framework to search for DOP gadgets. By assuming AAR and AAW capability, BOPC [34] further automatically generates DOP exploit payloads. However, in practice, how to get AAR and AAW capability in practice, especially when the target is fully protected, is not addressed in previous solutions.

In contrast, COOPLUS is a CFI-oriented attack. Given a weak vulnerability, VScape is committed to building AAR and AAW primitives under modern defenses.

7 Discussion

Potential Mitigations. Given the preconditions of the COOPLUS, there are two ways to protect applications from this attack, including: (1) separating `vptr` from writable and vulnerable heap objects, e.g., by putting them in a separate protected memory region; and (2) protecting the integrity of `vptr`, e.g., by applying DFI to block illegal writes to `vptr`.

The first type of defense will break the C++ ABI, as shown in CFIXX [24]. Thus, such defenses are not practical. Proper mitigation would be protecting the integrity of `vptr`. However, traditional data-flow integrity solutions (e.g., [21, 22]) in general have high performance overheads, which also prohibit the adoption.

Instead, we think applicable mitigation is a combination of type-based and context-sensitive CFI solutions, which could provide similar protection as data flow integrity. As shown in our experiment, the context-sensitive CFI solution μCFI [8] successfully protects the benchmark code [67] from COOPLUS with precise runtime information. But μCFI requires Intel PT and works on a customized kernel, making it hard to deploy in practice. By contrast, type-based CFI solutions are popular and take less effort in implementation and deployment. Thus, for a perfect defense, it is necessary to measure the size of overridden virtual functions for each virtual call site. For a virtual call with only one candidate virtual function implementation, a type-based check is enough to ensure the control-flow integrity. But for virtual calls having more than one compatible function, the context information (e.g., the origin of objects) should be considered.

But there are some challenges to address, in order to efficiently track context information without causing compatibility issues. CFI-LB [62] uses call stack to represent the context, is able to reduce the size of EC, but still leaves multiple valid targets. OS-CFI [2] utilizes the origin sensitivity to divide the targets of each ICT into the smallest sets, however, has severe compatibility issues [32]. These challenges are left as future work.

Limitations of VScape. Same to other state-of-the-art AEG researches (i,e, Gollum [12], Revery [11] and BOPC [34],), VScape is also not fully automated. It still greatly depends on exploit templates to prepare for the prospective exploit routine. There are still several open challenges to address, including but not only limited to (1) automated heap layout manipulation, especially in a heap manager with a garbage collector, (2) generating exploits for complicated and large applications (such as browsers), and (3) requiring expert knowledge (e.g., exploiting strategies) to compose multi-step exploits. These challenges greatly limit the availability of these AEG tools, including VScape.

Practicality of COOPLUS. For C++ applications utilizing virtual functions, the COOPLUS attack surface is large, as shown in Table 1. As proved by the examples in Firefox and PyQt, this attack is feasible in real world targets. We believe this type of attack is general and realistic. However, we cannot guarantee this type of attack will always succeed. The key factor affecting the success rate is the number of available exploit primitives existed in target applications.

8 Conclusions

In this paper, we propose an advanced attack COOPLUS, and present a solution VScape to assess the effectiveness of virtual calls defenses against this attack. COOPLUS is a code reuse attack that is able to bypass every virtual call protection as long as it (1) does not break the ABI of virtual calls, (2) cannot guarantee the integrity of C++ objects' VTable pointers, and (3) allow multiple runtime targets at virtual call sites. Following the principle of COOPLUS, our solution VScape analyzes target applications and compiles proper exploit primitives for generating final exploits, to assess the effectiveness of target defenses. We evaluated VScape on C++ applications with known vulnerabilities. Results showed that real-world applications have a large set of exploitable virtual calls, and VScape could be utilized to generate working exploits to bypass virtual call protections with weak vulnerabilities. We concluded that, to fully mitigate COOPLUS in practice, we have to protect the integrity of `vptr` with a low performance overhead and good compatibility.

Acknowledgement

This work was supported in part by National Natural Science Foundation of China under Grant 61772308,

61972224, U1736209 and U1836112, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] M. MartnAbadi and J. L. ÚlfarErlingsson, "Control flow integrity: Principles, implementations, and applications." in *Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, Virginia*, 2005, pp. 340–353.

[2] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity." in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 195–211.

[3] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[4] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables." in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 559–573.

[5] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications." in *2015 IEEE Symposium on Security and Privacy*. IEEE, pp. 745–762. [Online]. Available: https://ieeexplore.ieee.org/document/7163058/

[6] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm." in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.

[7] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "Vtrust: Regaining trust on virtual calls." in *NDSS*, 2016.

[8] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity." in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1470–1486.

[9] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley., "Aeg: Automatic exploit generation," 2011.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code." in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 380–394.

[11] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable." in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1914–1927.

[12] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters." in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1689–1706.

[13] PaX-Team, "PaX ASLR (Address Space Layout Randomization)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in *USENIX Security Symposium*, 2011, pp. 25–41.

[15] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations." in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 78–87.

[16] M. Wang, P. Su, Q. Li, L. Ying, Y. Yang, and D. Feng, "Automatic polymorphic exploit generation for software vulnerabilities." in *International Conference on Security and Privacy in Communication Systems*. Springer, 2013, pp. 216–233.

[17] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *SECURITY*, 1998.

[18] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation." in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[19] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis." in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[20] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS*, 2015.

[21] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 147–160.

[22] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[23] A. Pawlowski, V. van der Veen, D. Andriesse, E. van der Kouwe, T. Holz, C. Giuffrida, and H. Bos, "Vps: excavating high-level c++ constructs from low-level binaries to protect dynamic dispatching," in *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, Dec 2019, p. 97–112. [Online]. Available: https://dl.acm.org/doi/10.1145/3359789.3359797

[24] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for c++ virtual dispatch." in *Prof. of ISOC Network & Distributed System Security Symposium (NDSS). https://hexhive. epfl. ch/publications/files/18NDSS. pdf*, 2018.

[25] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level." in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[26] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries." in *NDSS*, 2015.

[27] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[28] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks." in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 952–963.

[29] S. Andersen and V. Abella, "Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies." http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[30] A. Sotirov, "Heap feng shui in javascript." *Black Hat Europe*, 2007.

[31] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack." *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, 2018.

[32] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms." in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020.

[33] B. Maxwell, D. Thompson, G. Amerson, and L. Johnson, "Analysis of crc methods and potential data integrity exploits." in *International Conference on Emerging Technologies*, 2003, pp. 25–26.

[34] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks." in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1868–1882.

[35] K. Serebryany and M. Böhme, "Aflgo: Directing afl to reach specific target locations." 2017.

[36] J. Vanegue, "The automated exploitation grand challenge." in *presented at H2HC Conference*, 2013.

[37] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel." in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1707–1722.

[38] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation." in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 763–779.

[39] "Firefox Browser." https://www.mozilla.org/, 2020.

[40] "The Chromium Projects." https://www.chromium.org/, 2020.

[41] "Opencascade ." https://github.com/tpaviot/oce, 2019.

[42] "Bento4 | Fast, Modern Tools and C++ Class Library." https://github.com/axiomatic-systems/Bento4, 2020.

[43] "ImageMagick." https://www.imagemagick.org/, 2020.

[44] "Exif, IPTC & XMP metadata and ICC Profile." https://www.exiv2.org/, 2020.

[45] "Open Source Computer Vision Library." https://opencv.org/, 2020.

[46] "Official mirror of the qt-project.org qt." https://github.com/qt, 2020.

[47] "A GRaphical Universal Modeler." https://agrum.gitlab.io/, 2020.

[48] "Open Source/Free Software cross-platform network engine." https://github.com/SLikeSoft/SLikeNet, 2020.

[49] "Bitcoin-Open source P2P money." https://bitcoin.org/en/, 2020.

[50] "ZNC-An advanced IRC bouncer." https://github.com/znc/znc, 2020.

[51] "MongoDB," https://github.com/mongodb/mongo, 2020.

[52] "Open Babel: The Open Source Chemistry Toolbox." http://openbabel.org/wiki/Main_Page, 2020.

[53] "Clang: a C language family frontend for LLVM." https://clang.llvm.org/, 2005.

[54] "The LLVM Compiler Infrastructure." https://llvm.org/, 2000.

[55] "Valgrind Home." https://valgrind.org/, 2020.

[56] M. Namolaru, "Devirtualization in gcc." in *Proceedings of the GCC Developers' Summit*. Citeseer, 2006, pp. 125–133.

[57] "Intel memory protection extensions." 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html

[58] "Processor Tracing." https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[59] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[60] B. Niu and G. Tan, "Modular control-flow integrity." in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 577–587.

[61] ——, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[62] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 95–110.

[63] "Pin - a dynamic binary instrumentation tool." 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[64] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: Securing c++ virtual calls from memory corruption attacks." in *NDSS*, 2014.

[65] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[66] "Reproduction of SafeDispatch." https://github.com/kongxiao0532/safedispatch-reproduce, 2020.

[67] "Benchmark used for testing CFI solutions' effectiveness against the COOP LUS attack." https://github.com/https://github.com/cooplus-vscape/CFIbenchmark, 2021.

[68] "Buffer Overflow in python socket packet." https://bugs.python.org/issue20246, 2014.

[69] "Mozilla Firefox Audio Driver Out of Bounds." https://bugzilla.mozilla.org/show_bug.cgi?id=1446062, 2018.

[70] "jemalloc: A general purpose malloc(3) implementation." 2017. [Online]. Available: https://github.com/jemalloc/jemalloc

[71] "Mozilla Garbage collection." 2005. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Garbage_collection

[72] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity." in *2014 IEEE Symposium on Security and Privacy*. IEEE, pp. 575–589. [Online]. Available: http://ieeexplore.ieee.org/document/6956588/

[73] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard." in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 417–432.

[74] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.

[75] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity." in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, pp. 901–913. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2810103.2813646

[76] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications." in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 143–157.

[77] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *IEEE Symposium on Security and Privacy (Oakland)*. IEEE, May 2017, p. 824–839. [Online]. Available: http://ieeexplore.ieee.org/document/7958612/

[78] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits." ACM Press, 2017, p. 25–35. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3139337.3139346

[79] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, "Towards automated generation of exploitation primitives for web browsers," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, Dec 2018, p. 300–312. [Online]. Available: https://dl.acm.org/doi/10.1145/3274694.3274723

[80] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities." in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 781–797.

[81] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1111–1128. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yun

[82] F. Deng, J. Wang, B. Zhang, C. Feng, Z. Jiang, and Y. Su, "A pattern-based software testing framework for exploitability evaluation of metadata corruption vulnerabilities," *Scientific Programming*, vol. 2020, 2020.

[83] Z. Zhao, Y. Wang, and X. Gong, "Haepg: An automatic multi-hop exploitation generation framework," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020, pp. 89–109.

[84] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits." in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 177–192.

[85] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks." in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[86] "Pymalloc: A Specialized Object Allocator." https://docs.python.org/2.3/whatsnew/section-pymalloc.html, 2002.

[87] "The gnu c library (glibc)." 2019, online: accessed 26-Feb-2019. [Online]. Available: https://www.gnu.org/software/libc/

[88] Y. Chen and X. Xing, "A systematic study of elastic objects in kernel exploitation." in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020.

A Appendix

Due to the page limit, we present the case study of PyQt, the sketch of the exploit for Firefox, the benchmark code protected by CFI but bypassed by COOPLUS, and the detail of how VScape performs capability analysis for primitives.

A.1 Case Study of PyQt-5.12

CPython itself has no virtual calls for COOPLUS since it is a program fully developed with C language. But with binding libraries, Python can easily use APIs compiled into shared libraries. PyQt is a widely used library in Python GUI programming, which is developed with C++. Cooperating with a publicly documented heap overflow vulnerability CVE-2014-1912 [68], we evaluate COOPLUS attack for AAR and AAW as we do for Firefox.

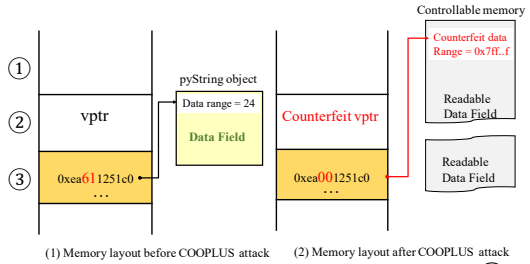


Figure 10: COOPLUS exploit primitives in PyQt. ① vulnerable object: socket character buffer ② victim object: QWidgetPrivate ③ relay object: item list of pyListObject.

Like Firefox, Python implements an independent allocator named Pymalloc [86] to manage user-controlled Pyobjects. But when the size of the Pyobject exceeds a threshold value (512 bytes on 64 bit systems), the native allocator will take it over. In Linux, CPython utilizes ptmalloc [87] to manage the native heap. Comparing with jemalloc, ptmalloc makes the heap layout extremely casual without strict isolation.

Amplification Strategy. CVE-2014-1912 [68] is a typical heap overflow vulnerability. Attackers can write arbitrary value directly into out-of-bound areas. And the number of bytes corrupted is enough for our experiment. When the size of the bytearray is 512 bytes, the three key objects could be placed closely. In this case, VTables of the victim and the counterfeit object are placed near to each other, we don't have to guess the base address of the Qt library. A partial overwrite is enough to make the hijacking.

The vulnerable object is a character buffer in the native heap. Thus, the victim object in primitive and the relay object should be maintained in the native heap too. And there are no other size requirements for objects in this case.

The strategy we used on Firefox still works well in this case. We find the data field of pyListObject in CPython is an ideal relay object. According to our observation, when the items of a pyListObject exceed 64, the item table will be allocated into the native heap. We select primitives that can tamper with Hi Address of pointers in the relay object with boolean values, redirect the pointer to a non-mapping address, conterfiet a pyString object, and set the size of string *0x7ff..f* (set a large value for the size of string). As a result, we can leak memory of a wide range. Then we attain the base address of library *array.cpython*, and fake a bytearray object for AAR and AAW.

Primitive in Exploit. The same as what we do in the case of firefox, we list two rules (1) the offset of victim member variable *offset mod 8 > 1* and (2) the primitive should have gadgets of ST-OW-nonPtr. Finally, the primitive tuple (QWidgetPrivate::endBackingStorePainting(), QWidgetPrivate, QOpenGLWidgetPrivate) is selected for a byte flipping in this case. But there is no constraints for control flow this time. The only thing that the counterfeit function QOpenGLWidgetPrivate::endBackingStorePainting() does is to set the byte at the offset +490 in the counterfeit object to zero. Since the chunk size of the victim object QWidgetPrivate is 464 in ptmalloc, it can actually affect the 27th byte in the relay object in the next chunk.

Exploit Synthesis. As shown in Figure 10, we place a list of string (data items of pyListObject) as the relay object, then the pointer to the string object is corrupted and redirected to attacker-controllable areas. The forged data length in the fake string object helps to leak data of a large range. As long as the library base of *array.cpython* is obtained, a fake object of bytearray is built, with controllable data pointer, we successfully get primitives for AAR and AAW.

Table 4: Statistics of Primitive Capability Analyzing.

App	#UVC -OVF	INVES Units	Failure Rate	St-*		Ld-AW[/Ex]-*				Sum
				nonPtr	Ptr	Ex-PC	nonCtrl	Const	Ctrl	
Bento4	31	361	4.4%	19	8	55	108	87	9	286
Bitcoin	25	60	1.7%	4	0	13	4	4	4	29
qt	840	4,206	4.0%	471	453	106	126	46	73	1,275
firefox	969	4,303	13.3%	633	66	597	361	230	133	2,020
chromium	3,741	14,822	13.8%	1,326	280	1,634	896	838	386	**5,360**
ImageMagick	1	51	0.0%	0	0	0	0	0	0	0
exiv2	19	63	1.6%	10	10	8	16	0	3	47
opencv	86	3,185	6.2%	758	31	506	18	1,641	76	3,030
aGrum	8	21	14.3%	6	2	1	6	16	0	31
SLikeNet	11	222	11.3%	34	1	27	8	3	5	78
mongodb	206	662	11.2%	52	13	174	75	115	141	570
oce	303	1,594	8.5%	156	9	273	78	24	30	570
znc	28	65	13.8%	17	4	10	6	5	4	46
openbable	31	374	24.1%	89	19	29	28	89	80	334

#UVC-**OVF**: UVC with OOB VFunc.
INVS Unit: an investigation unit is that like [counterfeit virtual function, a member variable in counterfeit object].
Failure Rate: the proportion of Failed-to-Analyzing INVS units.

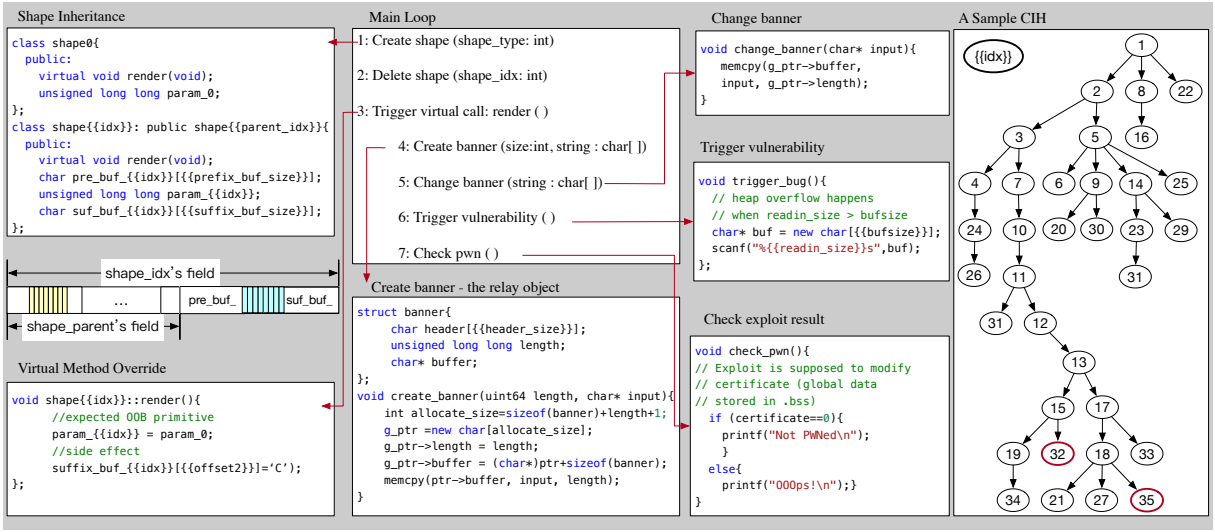


Figure 11: Summary for the motivation example. The template use *{{idx}}* to distinguish different derived *shape* classes. When the expected number for *{{idx}}* ranges from 1 to 35, a sample class inheritance hierarchy (CIH) can be seen from the figure.

A.2 Primitive Capability Analysis

We denote a primitive with a capability (defined in Section 3.3) as a primitive gadget. Table 4 shows the capabilities VScape found from candidate primitives. An investigation unit is a tuple contains a counterfeit function and a member variable of the counterfeit object whose address is in the relay object in COOPLUS attack. The third column shows numbers of investigation units VScape found. Notice that a primitive indicates a combination of a victim function and any counterfeit function which belongs to the subclass of the victim class, while an investigation unit is selected only from counterfeit functions who override the direct parents' method. So that the number of investigation units is much less than the number of total primitives. VScape successfully finds the majority of them as shown in the fourth column. Some of the units are failed to analyze because we restricted the maximum number of the taint paths and the trace depth, to ensure we can get a result in considerable time. Column 4-9 shows the exploitable instructions from the analyzed units, we can see there are sufficient gadgets found in applications except for ImageMagick [43]. And with this analysis, VScape filters out a great number of primitive candidates which are useless for exploitation. For example, 51 primitive candidates in ImageMagick have no capabilities for our requirements, which is hard to make help for further exploitation.

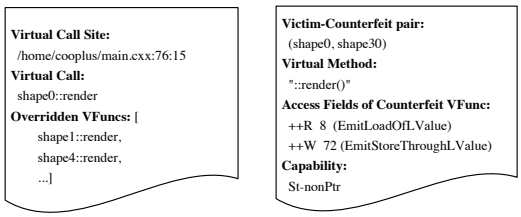
A.3 Motivation Example

In this section, we present a motivation example help readers better understand the steps of VScape as discussed in Section 4. Due to the space limitation, more details can be found online at https://github.com/cooplus-vscape .

A.3.1 Victim Program

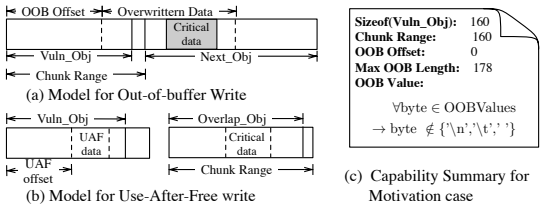
As shown in Figure 11, the target application dispatches tasks with a switch table in the main loop. Analysts can trigger different program behaviors with elaborate inputs.

This program implements polymorphism with a series of *shape* classes. The step 3 in the main loop triggers virtual call *::render()* for each created shape. For simplicity, we do not show the global inheritance in this figure,



(a) Sample Record for Virtual Calls (b) Sample Record for Primitive Pair

Figure 12: A candidate primitive in motivation case.



(a) Model for Out-of-buffer Write

(b) Model for Use-After-Free write

(c) Capability Summary for Motivation case

Figure 13: Vulnerability Description.

but present a template for shape declaration. The derived *shape* implements three exclusive fields - *pref_buf_{{idx}}, param_{{idx}} and sub_buf_{{idx}}*; The overridden *::render()* wirtes *param_0* into *para_{{idx}}*, making it an ideal candidate primitive for COOPLUS.

Furthermore, the *banner* is a flexible structure, which is similar to objects used in kernel exploitation [88]. It has a length field that controls the size for a content buffer, and maintains a pointer to it. For simplicity, *create_banner()* places the buffer close to the *banner* object. Then at step 5 of the main loop, analysts are able to modify data in this buffer. The overflow vulnerability locates at *trigger_bug()*. The *{{bufsize}}* determines the chunk size in the cache, whereas the *{{readin_size}}* defines the maximum length for read-in bytes. Assuming the goal of exploit is to corrupt the *certificate* in the global segment at runtime, we can verify the consequence for our attack with the use of *check_pwn()* at step 7. Lastly, we build the motivation example with the jemalloc heap allocator and the LLVM-CFI defense.

Moreover, to reflect the complexity of class hierarchy in real world application, this sample program implements more than thirty *shape_{{idx}}* classes with randomly generated *pre-* and *suf-* fields. It is hard for analysts to find a correct solution without systematic approaches, to corrupt the *certificate* field when a semantics-aware CFI (i.e., LLVM-CFI) is deployed.

A.3.2 Workflow of the VScape Compiler

The sample CIH is too complex to be analyzed manually, thus, VScape is developed as a systematic approach to compile elements for launching the COOPLUS attack. As shown in Figure 4, VScape has three major components.

The first task *primitive generation* is to search candidate primitives. VScape takes source code of target application as inputs and generates records of candidate primitives, as shown in Figure 12.

The second task, *expected primitive construction* component requires analysts to prepare (1) description of the given vulnerability and (2) expected exploit primitive attributes. Figure 13(a&b) models two types of vulnerabilities, and Figure 13(c) depicts the vulnerability in *trigger_bug()* with formalized language. And if either the pointer or length is corrupted, we can launch COOPLUS from there. For simplicity, we only focus on one exploit strategy, i.e., buffer pointer corruption, in this example. Figure 14 shows the expected primitive attributes which can enable the aforementioned strategy, which is provided by analysts too. Then VScape searches primitives fit for the vulnerability and expected primitive attributes. Figure 15 shows one qualified primitive and its memory state constraint in which the primitive could work.

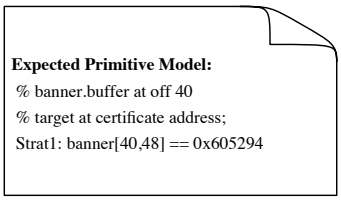


Figure 14: Expected Primitive Attributes.

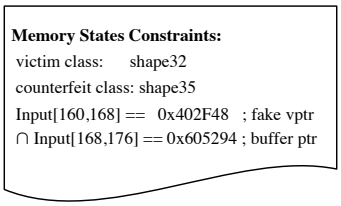


Figure 15: Memory State Constraints.

The user-provided exploit template takes cares of other critical steps of the exploitation, including (1) creating an expected heap layout for the character buffer, the victim object and the *banner*, (2) utilizing the given vulnerability to tamper with `vptr` , and (3) utilizing the primitives provided by VScape to finalize exploitation. VScape will provide qualified primitives for the exploit template to compose the final exploit. Figure 16 shows an example exploit, where texts in yellow background are generated by VScape.

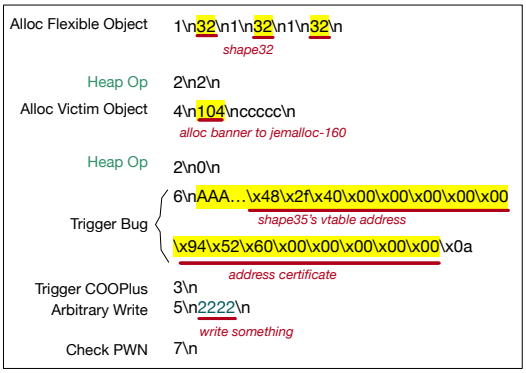


Figure 16: The Final Payload. Bytes in yellow background are automated generated by VScape while manual efforts are responsible for others.