Counterfeit Object-oriented Programming

On the Difficulty of Preventing Code Reuse Attacks in C++ Applications

Felix Schuster*, Thomas Tendyck*, Christopher Liebchen†, Lucas Davi†, Ahmad-Reza Sadeghi†, Thorsten Holz*
*Horst Görtz Institut (HGI) Ruhr-Universität Bochum, Germany
†CASED Technische Universität Darmstadt, Germany

***Abstract*—Code reuse attacks such as *return-oriented programming* (ROP) have become prevalent techniques to exploit memory corruption vulnerabilities in software programs. A variety of corresponding defenses has been proposed, of which some have already been successfully bypassed—and the arms race continues.**

In this paper, we perform a systematic assessment of recently proposed CFI solutions and other defenses against code reuse attacks in the context of C++. We demonstrate that many of these defenses that do not consider object-oriented C++ semantics precisely can be generically bypassed in practice. Our novel attack technique, denoted as *counterfeit object-oriented programming* (COOP), induces malicious program behavior by only invoking chains of existing C++ virtual functions in a program through corresponding existing call sites. COOP is Turing complete in realistic attack scenarios and we show its viability by developing sophisticated, real-world exploits for Internet Explorer 10 on Windows and Firefox 36 on Linux. Moreover, we show that even recently proposed defenses (*CPS*, *T-VIP*, *vfGuard*, and *VTint*) that specifically target C++ are vulnerable to COOP. We observe that constructing defenses resilient to COOP that do not require access to source code seems to be challenging. We believe that our investigation and results are helpful contributions to the design and implementation of future defenses against control-flow hijacking attacks.

I. Introduction

For more than two decades, attackers have been exploiting memory-related vulnerabilities such as buffer overflow errors to hijack the control flow of software applications developed in unsafe programming languages like C or C++. In the past, attackers typically immediately redirected the hijacked control flow to their own injected malicious code. This changed through the broad deployment of the well-known *data execution prevention* (DEP) countermeasure [33] that renders immediate *code injection attacks* infeasible. However, attackers adapted quickly and are typically resorting to *code reuse attacks* today.

Code reuse attack techniques, such as *return-oriented programming* (ROP) [46] or *return-to-libc* [37], avoid injecting code. Instead, they induce malicious program behavior by misusing existing code chunks (called *gadgets*) residing in the attacked application's address space. In general, one can distinguish between two phases of a runtime exploit: (1) the exploitation of a memory corruption vulnerability initially allowing the adversary to hijack the control flow of an application, and (2) the actual adversary-chosen malicious computations and program actions that follow. A generic mitigation of code reuse attacks is to prevent the initial exploitation step. In other words, code reuse attacks cannot be instantiated, if *spatial memory corruptions* like buffer overflows and *temporal memory corruptions* like use-after-free conditions are prevented in the first place [51]. Indeed, a large number of techniques have been proposed that provide means of spatial memory safety [5], [6], temporal memory safety [4], or both [13], [31], [36], [45]. On the downside, for precise guarantees, these techniques typically require access or even changes to an application's source code and may incur considerable overhead. This hampers their broader deployment [51].

Orthogonally, several defenses have been proposed that do not tackle the initial control-flow hijacking, but rather aim at containing or detecting the subsequent malicious control-flow transitions of code reuse attacks. A popular line of work impedes code reuse attacks by hiding [7], shuffling [55], or rewriting [39] an application's code or data in memory; often in a pseudo-random manner. For example, the widely deployed *address space layout randomization* (ASLR) technique ensures that the stack, the heap, and executable modules of a program are mapped at secret, pseudo-randomly chosen memory locations. This way, among others, the whereabouts of useful code chunks are concealed from an attacker. Bypassing these defenses often requires the exploitation of an additional *memory disclosure*—or *information leak*—vulnerability [51].

A complementary line of work concerns a generic security principle called *control-flow integrity* (CFI). It enforces the control flow of the program to adhere to a pre-determined or at runtime generated control-flow graph (CFG) [3]. Precise CFI—also known as *fine-grained* CFI—is conceptually sound [1]. However, similar to memory safety techniques, there are practical obstacles like overhead or required access to source code that hinder its broad deployment. Consequently, different instantiations of imprecise CFI—or *coarse-grained* CFI—and related runtime detection heuristics have been proposed, oftentimes working on binary code only. However, several researchers have recently shown that many of these solutions [3], [14], [23], [40], [56], [58], [59] can be bypassed in realistic adversary settings [11], [16], [25], [26], [43].

Contributions: In this paper, we present *counterfeit object-oriented programming* (COOP), a novel code reuse attack technique against applications developed in C++. With COOP we demonstrate the limitations of a range of proposed defenses against code reuse attacks in the context of C++. We show that it is essential for code reuse defenses to consider C++ semantics like the class hierarchy carefully and precisely. As recovering these semantics without access to source code

can be challenging or sometimes even impossible, our results demand for a rethinking in the assessment of binary-only defenses and make a point for the deployment of precise source code-based defenses where possible.

Our observation is that COOP circumvents virtually all CFI solutions that are not aware of C++ semantics. Further, we also find a range of other types of defenses that do not consider these semantics precisely to be prone to attacks. In fact, we show that even several recently and concurrently proposed defenses against control-flow hijacking/code reuse attacks that specifically target C++ applications (CPS [31], T-VIP [24], vfGuard [41], and VTint [57]) offer at most partial protection against COOP, and we can successfully bypass all of them in realistic attack scenarios. We also discuss how COOP can reliably be prevented by precise C++-aware CFI, defenses that provide (spatial and temporal) integrity for C++ objects, or defenses that prevent certain common memory disclosures.

We demonstrate the viability of our attack approach by implementing working low-overhead exploits for real-world vulnerabilities in Microsoft Internet Explorer 10 (32-bit and 64-bit) on Windows and a proof-of-concept vulnerability in Firefox 36 on Linux x64. To launch our attacks against modern applications, we inspected and identified easy-to-use gadgets in a set of well-known Windows system libraries—among them the standard Microsoft Visual C/C++ runtime that is dynamically linked to many applications—using basic symbolic execution techniques. We also show that COOP is *Turing complete* under realistic conditions.

Attack Technique Overview: Existing code reuse attacks typically exhibit unique characteristics in the control flow (and the data flow) that allow for generic protections regardless of the language an application was programmed in. For example, if one can afford to monitor all return instructions in an application while maintaining a full shadow call stack, even advanced ROP-based attacks [11], [16], [25], [26], [43] cannot be mounted [2], [17], [22]. This is different for COOP: it exploits the fact that each C++ virtual function is *address-taken*, which means that a constant pointer exists to it. Accordingly, C++ applications usually contain a high ratio of address-taken functions; typically a significantly higher one compared to C applications. If, for example, an imprecise CFI solution does not consider C++ semantics, these functions are all likely valid indirect call targets [3] and can thus be abused. COOP exclusively relies on C++ virtual functions that are invoked through corresponding calling sites as *gadgets*. Hence, without deeper knowledge of the semantics of an application developed in C++, COOP's control flow cannot reasonably be distinguished from a benign one. Another important difference to existing code reuse attacks is that in COOP conceptually no code pointers (e. g., return addresses or function pointers) are injected or manipulated. As such, COOP is immune against defenses that protect the integrity and authenticity of code pointers. Moreover, in COOP, gadgets do not work relative to the stack pointer. Instead, gadgets are invoked relative to the `this` pointer on a set of adversary-defined *counterfeit objects*. Note that in C++, the `this` pointer allows an object to access its own address. Addressing relative to the `this` pointer implies that COOP cannot be mitigated by defenses that prevent the stack pointer to point to the program's heap [23], which is typically the case for ROP-based attacks launched through a heap-based memory corruption vulnerability.

The counterfeit objects used in a COOP attack typically overlap such that data can be passed from one gadget to another. Even in a simple COOP program, positioning counterfeit objects manually can become complicated. Hence, we implemented a programming framework that leverages the Z3 SMT solver [18] to derive the object layout of a COOP program automatically.

II. Technical Background

Before presenting the ideas and concepts behind COOP in detail, we review the background necessary for understanding our approach and its relation to existing work.

A. Code Reuse Attack Techniques

Return-oriented programming (ROP) [46] is a widely used code reuse attack technique. The basic idea is to hijack the control flow of an application and redirect it to existing short instruction sequences ending in a return instruction (called *gadgets*) residing in the executable modules of a target application. Gadgets are oftentimes not aligned with the original instruction stream of an executable module. Each gadget fulfills a specific task such as performing an addition, or storing a value to memory. In order to execute a malicious ROP program, the adversary injects a chunk of code pointers into the address space of an application, where each pointer references one gadget. Finally, the attacker, abusing a memory corruption vulnerability, pivots a thread's stack pointer to that area. In the following, the injected code pointers on the (fake) stack are interpreted as return addresses making the control flow "return" from one attacker-chosen gadget to another. ROP can be considered a generalization of the older *return-to-libc* [37] code reuse attack technique where the attacker makes the hijacked control flow immediately "return" to the entry of a sensitive library functions residing for example in libc.

Jump-oriented programming (JOP) is a variant of ROP that uses indirect jumps and calls rather than return instructions [9], [12]. In basic JOP, return instructions are emulated by using a combination of a *pop-jmp* pair. In addition, JOP attacks do not necessarily require the stack pointer as base register to reference code pointers. In particular, an "update-load-branch" sequence with general purpose registers can be used instead [12]. The term *call-oriented programming* (COP) is also sometimes used to refer to ROP-derived techniques that employ indirect calls [11], [25].

Although these code reuse attack techniques are very powerful and return-to-libc, ROP, and JOP have even been shown to enable Turing complete (i. e., arbitrary) malicious computations [12], [46], [53] in realistic scenarios, they differ in several subtle aspects from ordinary program execution, which can be exploited to detect their execution. This is discussed in more detail in §III-A.

B. Control-Flow Integrity

Abadi et al. introduced the principle of *Control-Flow Integrity* (CFI) [3] as a generic defense technique against code reuse attacks. Since then, it has been generally used to refer to the concept of instrumenting indirect branches in native programs to thwart code reuse attacks. Usually, the enforcement of CFI is a two-step process:

1) determination of a program's approximate control-flow graph (CFG) X'.
2) instrumentation of (a subset of) the program's indirect branches with runtime checks that enforce the control flow to be compliant with X'.

The approximate CFG X' can be determined statically or dynamically; on source code or on binary code. X' should be a supergraph of the intrinsic CFG X encoded in the original source code of a program. If X' is equal to X, it is in general difficult for an attacker to divert control flow in a way that is not conform to the semantics of a program's source code. CFI checks are often implemented by assigning IDs to all possible indirect branch locations in a program. At runtime, a check before each indirect branch validates if the target ID is in compliance with X'. When the same ID is assigned to most of a program's address-taken functions and returns are not restricted to comply with the call stack at runtime, one often speaks of *coarse-grained* CFI. It has recently been shown that certain coarse-grained CFI solutions for binary code [3], [58], [59] cannot prevent advanced ROP-based attacks [16], [25].

C. C++ Code on Binary Level

As our attack approach targets C++ applications, we provide a brief introduction to the basic concepts of C++ and describe how they are implemented by compilers in the following.

In C++ and other object-oriented programming languages, programmers define custom types called *classes*. Abstractly, a class is composed of a set of member data fields and member functions [50]. A concrete instantiation of a class at runtime is called *object*. *Inheritance* and *polymorphism* are integral concepts of the object-oriented programming paradigm: new classes can be derived from one or multiple existing ones, *inheriting* at least all visible data fields and functions from their *base classes*. Hence, in the general case, an object can be accessed as instance of its actual class or as instance of any of its (immediate and not immediate) base classes. In C++, it is possible to define a member function as *virtual*. The implementation of an inherited virtual function may be overridden in a derived class. Invoking a virtual function on an object always invokes the specific implementation of the object's class even if the object was accessed as instance of one of its base classes. This is referred to as *polymorphism*.

C++ compilers implement calls to virtual functions (*vcalls*) with the help of *vtables*. A vtable is an array of pointers to all, possibly inherited, virtual functions of a class; hence, each virtual function is *address-taken* in an application. (For brevity, we do not consider the case of *multiple inheritance* here.)

Every object of a class with at least one virtual function contains a pointer to the corresponding vtable at its very beginning (offset $+0$). This pointer is called *vptr*. Typically, a vcall on Windows x64 is translated by a compiler to an instruction sequence similar to the following:

```
mov     rdx, qword ptr [rcx]
call    qword ptr [rdx+8]
```

Here, `rcx` is the object's `this` pointer—also referred to as *this-ptr* in the following. First, the object's vptr is temporarily loaded from offset $+0$ from the this-ptr to `rdx`. Next, in the given example, the second entry in the object's vtable is called by dereferencing `rdx+8`. Compilers generally hardcode the index into a vtable at a vcall site. Accordingly, this particular vcall site always invokes the second entry of a given vtable.

III. Counterfeit Object-oriented Programming

`COOP` is a code reuse attack approach targeting applications developed in C++ or possibly other object-oriented languages. We note that many of today's notoriously attacked applications are written in C++ (or contain major parts written in C++); examples include, among others, Microsoft Internet Explorer, Google Chrome, Mozilla Firefox, Adobe Reader, Microsoft Office, LibreOffice, and OpenJDK.

In the following, we first state our design goals and our attacker model for `COOP` before we describe the actual building blocks of a `COOP` attack. For brevity reasons, the rest of this section focuses on Microsoft Windows and the x86-64 architecture as runtime environment. The `COOP` concept is generally applicable to C++ applications running on any operating system; it also extends to other architectures.

A. Goals

With `COOP` we aim to demonstrate the feasibility of creating powerful code reuse attacks that do not exhibit the revealing characteristics of existing attack approaches. Even advanced existing variants of return-to-libc, ROP, JOP, or COP [8], [10], [11], [16], [25], [26], [43], [53] rely on control flow and data-flow patterns that are rarely or never encountered for regular code; among these are typically one or more of the following:

C-1 indirect *calls/jumps* to non address-taken locations
C-2 *returns* not in compliance with the call stack
C-3 excessive use of indirect branches
C-4 pivoting of the stack pointer (possibly temporarily)
C-5 injection of new code pointers or manipulation of existing ones

These characteristics still allow for the implementation of effective, low-level, and programming language-agnostic protections. For instance, maintaining a full shadow call stack [2], [17], [22] suffices to fend off virtually all ROP-based attacks.

With `COOP` we demonstrate that it is not sufficient to generally rely on the characteristics **C-1**–**C-5** for the design of code reuse defenses; we define the following goals for `COOP` accordingly:

G-1 do not expose the characteristics **C-1**–**C-5**.
G-2 exhibit control flow and data flow similar to those of benign C++ code execution.
G-3 be widely applicable to C++ applications.
G-4 achieve Turing completeness under realistic conditions.

B. Adversary Model

In general, code reuse attacks against C++ applications oftentimes start by hijacking a C++ object and its vptr. Attackers achieve this by exploiting a spatial or temporal memory corruption vulnerability such as an overflow in a buffer adjacent to a C++ object or a use-after-free condition. When the application subsequently invokes a virtual function on the hijacked object, the attacker-controlled vptr is dereferenced and a vfptr is loaded from a memory location of the attacker's choice. At this point, the attacker effectively controls the *program counter* (`rip` in x64) of the corresponding thread in the target application. Generally for code reuse attacks, controlling the program counter is one of the two basic requirements. The other one is gaining (partial) knowledge on the layout of the target application's address space. Depending on the context, there may exist different techniques to achieve this [8], [28], [44], [48].

For `COOP`, we assume that the attacker controls a C++ object with a vptr and that she can infer the base address of this object or another auxiliary buffer of sufficient size under her control. Further, she needs to be able to infer the base addresses of a set of C++ modules whose binary layouts are (partly) known to her. For instance, in practice, knowledge on the base address of a single publicly available C++ library in the target address space can be sufficient.

These assumptions conform to the attacker settings of most defenses against code reuse attacks. In fact, many of these defenses assume far more powerful adversaries that are, e. g., able to read and write large (or all) parts of an application's address space with respect to page permissions.

C. Basic Approach

Every `COOP` attack starts by hijacking one of the target application's C++ objects. We call this the *initial object*. Up to the point where the attacker controls the program counter, a `COOP` attack does not deviate much from other code reuse attacks: in a conventional ROP attack, the attacker typically exploits her control over the program counter to first manipulate the stack pointer and to subsequently execute a chain of short, return-terminated gadgets. In contrast, in `COOP`, virtual functions existing in an application are repeatedly invoked on counterfeit C++ objects carefully arranged by the attacker.

1) Counterfeit Objects: Typically, a counterfeit object carries an attacker-chosen vptr and a few attacker-chosen data fields. Counterfeit objects are *not* created by the target application, but are injected in bulk by the attacker. Whereas the *payload* in a ROP-based attack is typically composed of fake return addresses interleaved with additional data, in a `COOP` attack, the payload consists of counterfeit objects and possibly additional data. Similar to a conventional ROP payload, the `COOP` payload containing all counterfeit objects is typically written as one coherent chunk to a single attacker-controlled memory location.

2) Vfgadgets: We call the virtual functions used in a `COOP` attack *vfgadgets*. As for other code reuse attacks, the attacker identifies useful vfgadgets in an application prior to the actual attack through source code analysis or reverse engineering of binary code. Even when source code is available, it is necessary to determine the actual object layout of a vfgadget's class on binary level as the compiler may remove or pad certain fields. Only then the attacker is able to inject compatible counterfeit objects.

We identified a set of vfgadget types that allows to implement expressive (and Turing complete) `COOP` attacks in x86 and x64 environments. These types are listed in Table I. In the following, we gradually motivate our choice of vfgadget types based on typical code examples. These examples revolve around the simple C++ classes `Student`, `Course`, and `Exam`, which reflect *some* common code patterns that we found to induce useful vfgadgets. From §III-C3 to §III-C5, we first walk through the creation of a `COOP` attack code that writes to a dynamically calculated address; along the way, we introduce `COOP`'s integral concepts of *The Main Loop*, *Counterfeit Vptrs*, and *Overlapping Counterfeit Objects*. After that, from §III-D to §III-F, extended concepts for *Passing Arguments to Vfgadgets*, *Calling API Functions*, and *Implementing Conditional Branches and Loops* in `COOP` are explained.

The reader might be surprised to find more C++ code listings than actual assembly code in the following. This is owed to the fact that most of our vfgadgets types are solely defined by their high-level C++ semantics rather than by the side effects of their low level assembly code. These types of vfgadgets are thus likely to survive compiler changes or even the transition to a different operating system or architecture. In the cases where assembly code is given, it is the output of the Microsoft Visual C++ compiler (MSVC) version *18.00.30501* that is shipped with Microsoft Visual Studio 2013.

3) The Main Loop: To repeatedly invoke virtual functions without violating goals **G-1** and **G-2**, every `COOP` program essentially relies on a special *main loop vfgadget* (ML-G). The definition of an ML-G is as follows:

A virtual function that iterates over a container (e. g., a C-style array or a vector) of pointers to C++ objects and invokes a virtual function on each of these objects.

Virtual functions that qualify as ML-G are common in C++ applications. Consider for example the code in Figure 1: the class `Course` has a field `students` that points to a C-style array of pointers to objects of the abstract base class `Student`. When a `Course` object is destroyed (e. g., via `delete`), the virtual destructor[1] `Course::~Course` is executed and each `Student` object is informed via its virtual function `decCourseCount()` that one of the courses it was subscribed to does not exist anymore.

a) Layout of the Initial Object: The attacker shapes the initial object to resemble an object of the class of the ML-G. For our example ML-G `Course::~Course`, the initial object should look as depicted in Figure 2: its vptr is set to point into an existing vtable that contains a reference to the ML-G such that the first vcall under attacker control

[1]It is common practice to declare a virtual destructor when a C++ class has virtual functions.

Vfgadget type	*Purpose*	*Code example*
ML-G	The main loop; iterate over container of pointers to counterfeit object and invoke a virtual function on each such object.	see Figure 1
ARITH-G	Perform arithmetic or logical operation.	see Figure 4
W-G	Write to chosen address.	see Figure 4
R-G	Read from chosen address.	no example given, similar to W-G
INV-G	Invoke C-style function pointer.	see Figure 8
W-COND-G	Conditionally write to chosen address. Used to implement conditional branching.	see Figure 6
ML-ARG-G	Execute vfgadgets in a loop and pass a field of the *initial object* to each as argument.	see Figure 6
W-SA-G	Write to address pointed to by first argument. Used to write to *scratch area*.	see Figure 6
MOVE-SP-G	Decrease/increase stack pointer.	no example given
LOAD-R64-G	Load argument register `rdx`, `r8`, or `r9` with value (x64 only).	see Figure 4

TABLE I: Overview of `COOP` vfgadget types that operate on object fields or arguments; general purpose types are atop; auxiliary types are below the double line.

```
class Student {
public:
    virtual void incCourseCount() = 0;
    virtual void decCourseCount() = 0;
};

class Course {
private:
    Student **students;
    size_t nStudents;
public:
    /* ... */
    virtual ~Course() {
        for (size_t i = 0; i < nStudents; i++)
            students[i]->decCourseCount();          ML-G
        delete students;
    }
};
```

Fig. 1: Example for ML-G: the virtual destructor of the class `Course` invokes a virtual function on each object pointer in the array `students`.

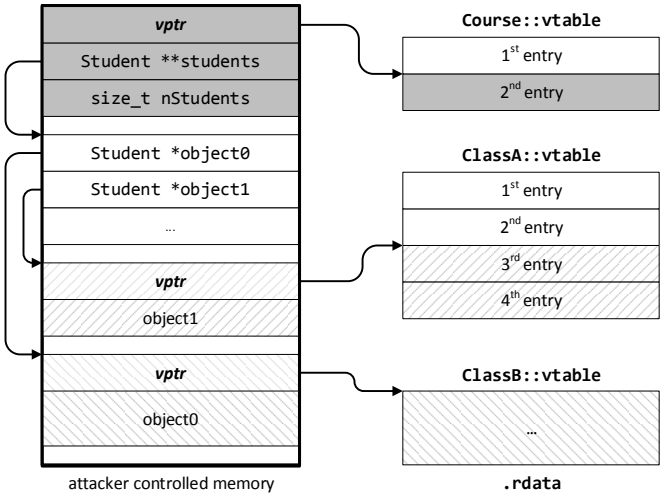

Fig. 2: Basic layout of attacker controlled memory (left) in a `COOP` attack using the example ML-G `Course::~Course`. The initial object (dark gray, top left) contains two fields from the class `Course`. Arrows indicate a *points-to* relation.

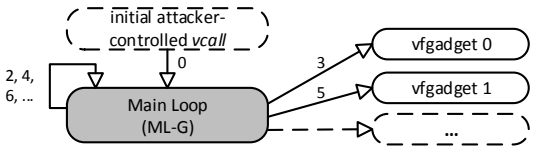

Fig. 3: Schematic control flow in a `COOP` attack; transitions are labeled according to the order they are executed.

leads to the ML-G. In contrast, in a ROP-based attack, this first vcall under attacker control typically leads to a gadget moving the stack pointer to attacker controlled memory. The initial object contains a subset of the fields of the class of the ML-G; i. e., all data fields required to make the ML-G work as intended. For our example ML-G, the initial object contains the fields `students` and `nStudents` of the class `Course`; the field `students` is set to point to a C-style array of pointers to counterfeit objects (*object0* and *object1* in Figure 2) and `nStudents` is set to the total number of counterfeit objects. This makes the `Course::~Course` ML-G invoke a vfgadget of the attacker's choice for each counterfeit object. Note how the attacker controls the vptr of each counterfeit object. Figure 3 schematically depicts the control-flow transitions in a `COOP` attack.

4) Counterfeit Vptrs: The control flow and data flow in a `COOP` attack should resemble those of a regular C++ program (**G-2**). Hence, we avoid introducing fake vtables and reuse existing ones instead. Ideally, the vptrs of all counterfeit objects should point to the *beginning* of existing vtables. Depending on the target application, it can though be difficult to find vtables with a useful entry at the offset that is fixed for a given vcall site. Consider for example our ML-G from Figure 1: counterfeit objects are treated as instances of the abstract class `Student`. For each counterfeit object, the 2^{nd} entry—corresponding to `decCourseCount()`—in the supplied vtable is invoked. (The 1^{st} entry corresponds to `incCourseCount()`.) Here, a `COOP` attack would ideally only use vfgadgets that are the 2^{nd} entry in an existing vtable. Naturally, this largely shrinks the set of available vfgadgets.

This constraint can be sidestepped by relaxing goal **G-2** and letting vptrs of counterfeit objects not necessarily point to the exact beginning of existing vtables but to certain positive or negative offsets as is shown for *object1* in Figure 2. When such *counterfeit vptrs* are used, any available virtual function can be invoked from a given ML-G.

5) Overlapping Counterfeit Objects: So far we have shown how, given an ML-G, an arbitrary number of virtual functions

```
class Exam {
private:
     size_t scoreA, scoreB, scoreC;
public:
     /* ... */
     char *topic;
     size_t score;
     virtual void updateAbsoluteScore() {
         score = scoreA + scoreB + scoreC;
     }                                                    ARITH-G

     virtual float getWeightedScore() {
         return (float)(scoreA*5+scoreB*3+scoreC*2) / 10;
     }                                                    LOAD-R64-G
};

struct SimpleString {
     char* buffer;
     size_t len;
     /* ... */
     virtual void set(char* s) {
         strncpy(buffer, s, len);                         W-G
     }
};
```

Fig. 4: Examples for ARITH-G, LOAD-R64-G, and W-G; for simplification, the native integer type `size_t` is used.

(vfgadgets) can be invoked while control flow and data flow resemble those of the execution of benign C++ code.

Two exemplary vfgadgets of types ARITH-G (arithmetic) and W-G (writing to memory) are given in Figure 4: in `Exam::updateAbsoluteScore()` the field `score` is set to the sum of three other fields; in `SimpleString::set()` the field `buffer` is used as destination pointer in a write operation. In conjunction, these two vfgadgets can be used to write attacker-chosen data to a dynamically calculated memory address. For this, two *overlapping* counterfeit objects are needed and their alignment is shown in Figure 5.

The key idea here is that the fields `score` in *object0* and `buffer` in *object1* share the same memory. This way, the result of the summation of the fields of *object0* in `Exam::updateAbsoluteScore()` is written to the field `buffer` of *object1*. Note how here, technically, also *object0.topic* and *object1.vptr* overlap. As the attacker does not use *object0.topic* this not a problem and she can simply make the shared field carry *object1.vptr*. Of course, in our example, the attacker would likely not only wish to control the *destination address* of the write operation through *object1.buffer* but also the *source address*. For this, she needs to be able to set the argument for the vfgadget `SimpleString::set()`. How this can be achieved in `COOP` is described next.

D. Passing Arguments to Vfgadgets

The overlapping of counterfeit objects is an important concept in `COOP`. It allows for data to flow between vfgadgets through object fields regardless of compiler settings or calling conventions. Unfortunately, we found that useful vfgadgets that operate exclusively on object fields are rare in practice. In fact, most vfgadgets we use in our real world exploits (see

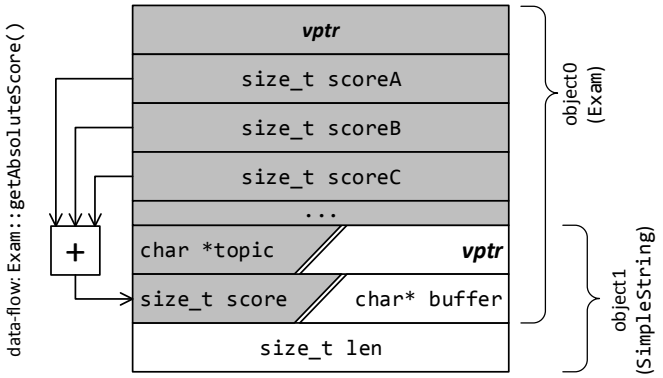

Fig. 5: Overlapping counterfeit objects of types `Exam` and `SimpleString`

§V) operate on both fields and arguments as is the case for `SimpleString::set()`.

Due to divergent default calling conventions, we describe different techniques for passing arguments to vfgadgets for x64 and x86 in the following.

1) Approach Windows x64: In the default x64 calling convention on Windows, the first four (non-floating point) arguments to a function are passed through the registers `rcx`, `rdx`, `r8`, and `r9` [35]. In case there are more than four arguments, the additional arguments are passed over the stack. For C++ code, the this-ptr is passed through `rcx` as the first argument. All four argument registers are defined to be caller-saved; regardless of the actual number of arguments a callee takes. Accordingly, virtual functions often use `rdx`, `r8`, and `r9` as scratch registers and do not restore or clear them on returning. This circumstance makes passing arguments to vfgadgets simple on x64: first, a vfgadget is executed that loads one of the corresponding counterfeit object's fields into `rdx`, `r8`, or `r9`. Next, a vfgadget is executed that interprets the contents of these registers as arguments.

We refer to vfgadgets that can be used to load argument registers as LOAD-R64-G. For the x64 arguments passing concept to work, a ML-G is required that itself does not pass arguments to the invoked virtual functions/vfgadgets. Of course, the ML-G must also not modify the registers `rdx`, `r8`, and `r9` between such invocations. In our example, the attacker can control the source pointer `s` of the write operation (namely `strncpy()`) by invoking a LOAD-R64-G that loads `rdx` *before* `SimpleString::set()`.

As an example for a LOAD-R64-G, consider `Exam::getWeightedScore()` from Figure 4; MSVC compiles this function to the following assembly code:

```
mov rax, qword ptr [rcx+10h]
mov r8, qword ptr [rcx+18h]
xorps   xmm0, xmm0
lea rdx, [rax+rax*2]
mov rax, qword ptr [rcx+8]
lea rcx, [rax+rax*4]
lea r9, [rdx+r8*2]
add r9, rcx
cvtsi2ss    xmm0, r9
addss   xmm0, dword ptr [__real0]
divss   xmm0, dword ptr [__real1]
ret
```

In condensed from, this LOAD-R64-G provides the following useful semantics to the attacker:

$$\begin{aligned} \texttt{rdx} &\leftarrow 3 \cdot [\mathit{this} + 10h] \\ \texttt{r8} &\leftarrow [\mathit{this} + 18h] \\ \texttt{r9} &\leftarrow 3 \cdot [\mathit{this} + 18h] + 2 \cdot [\mathit{this} + 10h] \end{aligned}$$

Thus, by carefully choosing the fields at offsets $10h$ and $18h$ from the this-ptr of the corresponding counterfeit object, the attacker can write arbitrary values to the registers `rdx`, `r8`, and `r9`.

In summary, to control the source pointer in the writing operation in `SimpleString::set()`, the attacker would first invoke `Exam::getWeightedScore()` for a counterfeit object carrying the desired source address divided by 3 at offset $10h$. This would load the desired source address to `rdx`, which would next be interpreted as the argument `s` in the vfgadget `SimpleString::set()`.

a) Other Platforms: In the default x64 C++ calling convention used by GCC [32], e. g., on Linux, the first six arguments to a function are passed through registers instead of only the first four registers. In theory, this should make `COOP` attacks simpler to create on Linux x64 than on Windows x64, as two additional registers can be used to pass data between vfgadgets. In practice, during the creation of our example exploits (see §V), we did not experience big differences between the two platforms.

Although we did not conduct experiments on RISC platforms such as ARM or MIPS, we expect that our x64 approach directly extends to these because in RISC calling conventions arguments are also primarily passed through registers.

2) Approach Windows x86: The standard x86 C++ calling convention on Windows is *thiscall* [35]: all regular arguments are passed over the stack whereas the this-ptr is passed in the register `ecx`; the callee is responsible for removing arguments from the stack. Thus, the described approach for x64 does not work for x86.

In our approach for Windows x86, contrary to x64, we rely on a *main loop* (ML-G) that *passes* arguments to vfgadgets. More precisely, a 32-bit ML-G should pass one field of the *initial object* as argument to each vfgadget. In practice, any number of arguments may work; for brevity we only discuss the simplest case of *one* argument here. We call this field the *argument field* and refer to this variant of ML-G as ML-ARG-G. For an example of an ML-ARG-G, consider the virtual destructor of the class `Course2` in Figure 6: the field `id` is passed as argument to each invoked virtual function. Given such an ML-ARG-G, the attacker can employ one of the two following approaches to pass chosen arguments to vfgadgets:

A-1 fix the *argument field* to point to a writable *scratch area*.
A-2 dynamically rewrite the *argument field*.

In approach **A-1**, the attacker relies on vfgadgets that interpret their first argument not as an immediate value but as a *pointer* to data. Consider for example the virtual function `Student2::getLatestExam()` from Figure 6 that copies an `Exam` object; MSVC produces the optimized

```
class Student2 {
private:
   std::list<Exam> exams;
public:
   /* ... */
   virtual void subscribeCourse(int id) { /* ... */ }
   virtual void unsubscribeCourse(int id) { /* ... */ }

   virtual bool getLatestExam(Exam &e) {
      if (exams.empty()) return false;          W-SA-G
      e = exams.back();
      return true;                              W-COND-G
   }
};

class Course2 {
private:
   Student2 **students;
   size_t nStudents;
   int id;
public:
   /* ... */
   virtual ~Course2() {
      for (size_t i = 0; i < nStudents; i++)
         students[i]->unsubscribeCourse(id);
      delete students;                          ML-ARG-G
   }
};
```

Fig. 6: Examples for W-SA-G, W-COND-G, ML-ARG-G

```
push    ebp
mov ebp, esp
cmp dword ptr [ecx+8], 0
jne copyExam
xor al, al
pop ebp
ret 4
copyExam:
mov eax, dword ptr [ecx+4]
mov ecx, dword ptr [ebp+8]
mov edx, dword ptr [eax+4]
mov eax, dword ptr [edx+0Ch]
mov dword ptr [ecx+4], eax
mov eax, dword ptr [edx+10h]
mov dword ptr [ecx+8], eax
mov eax, dword ptr [edx+14h]
mov dword ptr [ecx+0Ch], eax
mov eax, dword ptr [edx+18h]
mov dword ptr [ecx+10h], eax
mov al, 1
pop ebp
ret 4
```

Listing 1: Optimized x86 assembly code produced by MSVC for `Student2::getLatestExam()`.

x86 assembly code shown in Listing 1 for the function. In condensed form, lines 9–22 of the assembly code provide the following semantics:

$$\begin{aligned} [\mathit{arg0} + 4] &\leftarrow [[[\mathit{this} + 4] + 4] + Ch] \\ [\mathit{arg0} + 8] &\leftarrow [[[\mathit{this} + 4] + 4] + 10h] \\ [\mathit{arg0} + Ch] &\leftarrow [[[\mathit{this} + 4] + 4] + 14h] \\ [\mathit{arg0} + 10h] &\leftarrow [[[\mathit{this} + 4] + 4] + 18h] \end{aligned}$$

Note that for approach **A-1**, *arg0* always points to the *scratch area*. Accordingly, this vfgadget allows the attacker to copy 16 bytes (corresponding to the four 32-bit fields of `Exam`) from the attacker-chosen address $[[this + 4] + 4+] + Ch$ to the scratch area. We refer to this type of vfgadget that writes attacker-controlled fields to the scratch area as W-SA-G.

Using `Student2::getLatestExam()` as W-SA-G in conjunction with a ML-ARG-G allows the attacker, for example, to pass a string of up to 16 characters as first argument to the vfgadget `SimpleString::set()`.

In approach **A-2**, the argument field of the initial object is not fixed as in approach **A-1**. Instead, it is dynamically rewritten during the execution of a `COOP` attack. This allows the attacker to pass *arbitrary* arguments to vfgadgets; as opposed to *a pointer to arbitrary data* for approach **A-1**. For this approach, naturally, a usable W-G is required. As stated above, we found vfgadgets working solely with fields to be rare. Hence, the attacker would typically initially follow approach **A-1** and implement **A-2**-style argument writing on top of that when required.

a) Passing Multiple Arguments and Balancing the Stack: So far, we have described how a single argument can be passed to each vfgadget using a ML-ARG-G main loop gadget on Windows x86. Naturally, it can be desirable or necessary to pass more than one argument to a vfgadget. Doing so is simple: the ML-ARG-G pushes one argument to each vfgadget. In case a vfgadget does not expect any arguments, the pushed argument remains on the top of the stack even after the vfgadget returned. This effectively moves the stack pointer permanently one slot up as depicted in Figure 7 ③. This technique allows the attacker to gradually "pile up" arguments on the stack as shown in Figure 7 ④ before invoking a vfgadget that expects multiple arguments. This technique only works for ML-ARG-Gs that use `ebp` and not `esp` to access local variables on the stack (i.e., no *frame-pointer omission*) as otherwise the stack frame of the ML-ARG-G is destroyed.

Analogously to how vfgadgets without arguments can be used to move the stack pointer *up* under an ML-ARG-G, vfgadgets with more than one argument can be used to move the stack pointer *down* as shown in Figure 7 ②. This may be used to compensate for vfgadgets without arguments or to manipulate the stack. We refer to vfgadgets with little or no functionality that expect less or more than one argument as MOVE-SP-Gs. Ideally, a MOVE-SP-G is an empty virtual function that just adjusts the stack pointer.

The described technique for passing multiple arguments to vfgadgets in 32-bit environments can also be used to pass more than three arguments to vfgadgets in 64-bit environments.

b) Other Platforms: The default x86 C++ calling convention used by GCC, e. g., on Linux, is not *thiscall* but *cdecl* [35]: all arguments including the this-ptr are passed over the stack; instead of the callee, the caller is responsible for cleaning the stack. The technique of "piling up" arguments described in §III-D2a does thus not apply to GCC-compiled (and compatible) C++ applications on Linux x86 and other POSIX x86 platforms. Instead, for these platforms, we propose using ML-ARG-Gs that do not pass one but many controllable arguments to vfgadgets. Conceptually, passing too many arguments to a function does not corrupt the stack in the *cdecl* calling convention. Alternatively, ML-ARG-Gs could be switched during an attack depending on which arguments to a vfgadget need to be controlled.

E. Calling API Functions

The ultimate goal of code reuse attacks is typically to pass attacker-chosen arguments to critical API functions or system calls, e. g., Windows API (WinAPI) functions such as `WinExec()` or `VirtualProtect()`. We identified the following ways to call a WinAPI function in a `COOP` attack:

W-1 use a vfgadget that legitimately calls the WinAPI function of interest.

W-2 invoke the WinAPI function like a virtual function from the `COOP` main loop.

W-3 use a vfgadget that calls a C-style function pointer.

While approach **W-1** may be practical in certain scenarios and for certain WinAPI functions, it is unlikely to be feasible in the majority of cases. For example, virtual functions that call `WinExec()` should be close to non-existent.

Approach **W-2** is simple to implement: a counterfeit object can be crafted whose vptr does not point to an actual vtable but to the *import table* (IAT) or the *export table* (EAT) [42] of a loaded module such that the ML-G invokes the WinAPI function as a virtual function. Note that IATs, EATs, and vtables are all arrays of function pointers typically lying in read-only memory; they are thus in principle compatible data structures. As simple as it is, the approach has two important drawbacks: *(i)* it goes counter to our goal **G-2** as a C function is called at a vcall site without a legitimate vtable being referenced; and *(ii)* for x64, the this-ptr of the corresponding counterfeit object is always passed as the first argument to the WinAPI function due to the given C++ calling convention. This circumstance for example effectively prevents the passing of a useful command line to `WinExec()`. This can be different for other WinAPI functions, though. For example, calling `VirtualProtect()` with a this-ptr as first argument still allows the attacker to mark the memory of the corresponding counterfeit object as *executable*. Note that `VirtualProtect()` changes the memory access rights for a memory region pointed to by the first argument. Other arguments than the first one can be passed as described in §III-D1 for x64. For x86, *all* arguments can be passed using the technique from §III-D2.

For approach **W-3** a special type of vfgadget is required: a virtual function that calls a C-style function pointer with non-constant arguments. We refer to this type of vfgagdet as INV-G, an example is given in Figure 8: the virtual function `GuiButton::clicked()` invokes the field `GuiButton::callbackClick` as C-style function pointer. This particular vfgadget allows for the invocation of arbitrary WinAPI functions with at least three attacker-chosen arguments. Note that, depending on the actual assembly code of the INV-G, a fourth argument could possibly be passed

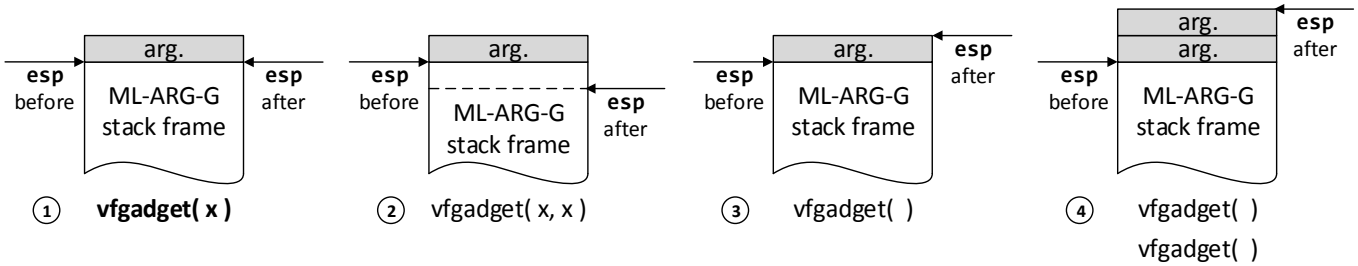

Fig. 7: Examples for stack layouts *before* and *after* invoking vfgadgets under an ML-ARG-G (*thiscall* calling convention). The stack grows upwards. ① vfgadget with one argument: the stack is balanced. ② vfgadget with two arguments: `esp` is moved down. ③ vfgadget without arguments: `esp` is moved up. ④ two vfgadgets without arguments: two arguments are piled up.

```
class GuiButton {
private:
      int id;
      void(*callbackClick)(int, int, int);
public:
      void registerCbClick(void(*cb)(int, int, int)) {
        callbackClick = cb;
      }

      virtual void clicked(int posX, int posY) {
            callbackClick(id, posX, posY);
      }                                                INV-G
};
```

Fig. 8: Example for INV-G: `clicked` invokes a field of `GuiButton` as C-style function pointer.

through `r9` for x64. Additional stack-bound arguments for x86 and x64 may also be controllable depending on the actual layout of the stack. Calling WinAPI functions through INV-Gs should generally be the technique of choice as this is more flexible than approach **W-1** and stealthier than **W-2**. An INV-G also enables seemingly legit transfers from C++ to C code (e. g., to libc) in general. On the downside, we found INV-Gs to be relatively rare overall. For our real-world example exploits discussed in §V, we could though always select from multiple suitable ones.

F. Implementing Conditional Branches and Loops

Up to this point, we have described all building blocks required to practically mount `COOP` code reuse attacks. As we do not only aim for `COOP` to be stealthy, but also to be *Turing complete* under realistic conditions (goal **G-4**), we now describe the implementation of *conditional branches* and *loops* in `COOP`.

In `COOP`, the *program counter* is the index into the container of counterfeit object pointers. The program counter is incremented for each iteration in the ML-G's main loop. The program counter may be a plain integer index as in our exemplary ML-G `Course::~Course` or may be a more complex data structure such as an iterator object for a C++ linked list. Implementing a conditional branch in `COOP` is generally possible in two ways: through *(i)* a conditional increment/decrement of the program counter or *(ii)* a conditional manipulation of the next-in-line counterfeit object pointers in the container. Both can be implemented given a conditional write vfgadget, which we refer to as W-COND-G. An example for this vfgadget type is again `Student2::getLatestExam()` from Figure 6. As can be seen in lines 3–7 of the function's assembly code in Listing 1, the controllable write operation is only executed in case $[\textit{this-ptr} + 8] \neq 0$. With this semantics, the attacker can rewrite the `COOP` program counter or upcoming pointers to counterfeit objects under the condition that a certain value is not null. In case the program counter is stored on the stack (e. g., in the stack frame of the ML-G) and the address of the stack is unknown, the technique for moving the stack pointer described in §III-D2a can be used to rewrite it.

Given the ability to conditionally rewrite the program counter, implementing loops with an exit condition also becomes possible.

IV. A Framework for Counterfeit Object-Oriented Programming

Implementing a `COOP` attack against a given application is a three step process: *(i)* identification of vfgadgets, *(ii)* implementation of attack semantics using the identified vfgadgets, and *(iii)* arrangement of possibly overlapping counterfeit objects in a buffer. Since the individual steps are cumbersome and hard to perform by hand, we created a framework in the Python scripting language that automates steps *(i)* and *(iii)*. This framework greatly facilitated the development of our example exploits for Internet Explorer and Firefox (see §V). In the following, we provide an overview of our implementation.

A. Finding Vfgadgets Using Basic Symbolic Execution

For the identification of useful vfgadgets in an application, our *vfgadget searcher* relies on binary code only and optionally debug symbols. Binary x86-64 C++ modules are disassembled using the popular *Interactive Disassembler* (IDA) version 6.5. Each virtual function in a C++ module is considered a potential vfgadget. The searcher statically identifies all vtables in a C++ module using debug symbols or, if these are not available, a set of simple but effective heuristics. Akin to other work [41], [57], our heuristics consider each address-taken array of function pointers a potential vtable. The searcher examines all identified virtual functions whose number of basic blocks does not exceed a certain limit. In practice, we found it sufficient and convenient to generally only consider

virtual functions with one or three basic blocks as potential vfgadgets; the only exception being ML-Gs and ML-ARG-Gs that due to the required loop often consist of more basic blocks. Using short vfgadgets is favorable as their semantics are easier to evaluate automatically and they typically exhibit fewer unwanted side effects. Including long vfgadgtes can, however, be necessary to fool heuristics-based code reuse attack detection approaches (see §VI).

The searcher summarizes the semantics of each basic block in a vfgadget in *single static assignment* (SSA) form. These summaries reflect the I/O behavior of a basic block in a compact and easy to analyze form. The searcher relies for this on the *backtracking* feature of the METASM binary code analysis toolkit [27], which performs symbolic execution on the basic block level. An example of a basic block summary as used by our searcher was already provided in the listed semantics for the second basic block of `Exam::getWeightedScore()` in §III-D1. To identify useful vfgadgets, the searcher applies filters on the SSA representation of the potential vfgadgets' basic blocks. For example, the filter: "*left side of assignment must dereference any argument register; right side must dereference the this-ptr*" is useful for identifying 64-bit W-Gs; the filter: "*indirect call independent of [this]*" is useful for finding INV-Gs; and the filter: "*looped basic block with an indirect call dependent on [this] and a non-constant write to [esp-4]*" can in turn be used to find 32-bit ML-ARG-Gs.

B. Aligning Overlapping Objects Using an SMT Solver

Each `COOP` "program" is defined by the order and positioning of its counterfeit objects of which each corresponds to a certain vfgadget. As described in §III-C5, the overlapping of counterfeit objects is an integral concept of `COOP`; it enables immediate data flows between vfgadgets through fields of counterfeit objects. Manually obtaining the alignment of overlapping counterfeit objects right on the binary level is a time-consuming and error-prone task. Hence, we created a `COOP` *programming environment* that automatically, if possible, correctly aligns all given counterfeit objects in a fixed-size buffer. In our programming environment, the "programmer" defines counterfeit objects and labels. A label may be assigned to any byte within a counterfeit object. When bytes within different objects are assigned the same label, the programming environment takes care that these bytes are mapped to the same location in the final buffer, while assuring that bytes with different labels are mapped to distinct locations. Fields without labels are in turn guaranteed to never overlap. These constraints are often satisfiable, as actual data within counterfeit objects is typically sparse.

For example, the counterfeit object *A* may only contain its vptr (at relative offset $+0$), an integer at the relative offset $+16$ and have the label *X* for its relative offset $+136$; the counterfeit object *B* may only contain its vptr and have the same label *X* for its relative offset $+8$. Here, the object *B* fits comfortably and without conflicts inside *A* such that *B* $+8$ maps to the same byte as *A* $+136$.

Our programming environment relies on the Z3 SMT solver [18] to determine the alignment of all counterfeit objects within the fixed-size buffer such that, if possible, all label-related constraints are satisfied. At the baseline, we model the fixed-size buffer as an *array* mapping integers indexes to integers in Z3. To prevent unwanted overlaps, for each byte in each field, we add a *select* constraint [19] in Z3 of the form

$$select(\textit{offset-obj} + \textit{reloffset-byte}) = \textit{id-field}$$

where *offset-obj* is an integer variable to be determined by Z3 and *reloffset-byte* and *id-field* are constant integers that together uniquely identify each byte. For each desired overlap (e. g., between objects *A* and *B* using label *X*), we add a constraint of the form

$$\textit{offset-objA} + \textit{reloffset(A,X)} = \textit{offset-objB} + \textit{reloffset(B,X)}$$

where *offset-objA* and *offset-objB* are integer variables to be determined by Z3 and $\textit{reloffset(A,X)} = 136$ and $\textit{reloffset(B,X)} = 8$ are constants.

In the programming environment, for convenience, symbolic pointers to labels can be added to counterfeit objects. Symbolic pointers are automatically replaced with concrete values once the offsets of all labels are determined by Z3. This way, multiple levels of indirection can be implemented conveniently.

V. Proof of Concept Exploits

To demonstrate the practical viability of our approach, we implemented exemplary `COOP` attacks for Microsoft Internet Explorer 10 (32-bit and 64-bit) and Mozilla Firefox 36 for Linux x64. In the following, we discuss different aspects of our attack codes that we find interesting. We used our framework described in §IV for the development of all three attack codes. Each of them fits into 1024 bytes or less. All employed vfgadgets and their semantics are listed in Tables A.I–A.IV in the Appendix.

For our Internet Explorer 10 examples, we used a publicly documented vulnerability related to an integer signedness error in Internet Explorer 10 [30] as foundation. The vulnerability allows a malicious website to perform arbitrary reads at any address and arbitrary writes within a range of approximately 64 pages on the respective heap using JavaScript code. This gives the attacker many options for hijacking C++ objects residing on the heap and injecting her own buffer of counterfeit objects; it also enables the attacker to gain extensive knowledge on the respective address space layout. We successfully tested our `COOP`-based exploits for Internet Explorer 10 32-bit and 64-bit on Windows 7. Note that our choice of Windows 7 as target platform is only for practical reasons; the described techniques also apply to Windows 8. To demonstrate the flexibility of `COOP`, we implemented different attack codes for 32-bit and 64-bit. Both attack codes could be ported to the respective other environment without restrictions.

A. Internet Explorer 10 64-bit

Our `COOP` attack code for 64-bit only relies on vfgadgets contained in mshtml.dll that can be found in every Internet Explorer process; it implements the following functionality: *(1)* read pointer to kernel32.dll from IAT; *(2)* calculate pointer

to WinExec() in kernel32.dll; *(3)* read the current tick count from the KUSER_SHARED_DATA data structure; *(4)* if tick count is odd, launch calc.exe using WinExec() else, execute alternate execution path and launch mspaint.exe.

The attack code consists of 17 counterfeit objects with counterfeit vptrs and four counterfeit objects that are pure data containers. Overall eight different vfgadgets are used; including one LOAD-R64-G for loading rdx through the dereferencing of a field that is used five times. The attack code is based on a ML-G similar to our exemplary one given in Figure 1 that iterates over a plain array of object pointers. With four basic blocks, the ML-G is the largest of the eight vfgadgets. The conditional branch depending on the current tick count is implemented by overwriting the next-in-line object pointer such that the ML-G is recursively invoked for an alternate array of counterfeit object pointers. In summary, the attack code contains eight overlapping counterfeit objects and we used 15 different labels to create it in our programming environment.

1) Attack Variant Using only Vptrs Pointing to the Beginning of Vtables: The described 64-bit attack code relies on counterfeit vptrs (see §III-C4) that do not necessarily point to the beginning of existing vtables but to positive or negative offset from them. As a proof of concept, we developed a stealthier variant of the attack code above that *only* uses vptrs that point to the beginning of existing vtables. Accordingly, at each vcall site, we were restricted to the set of virtual functions compatible with the respective fixed vtable index. Under this constraint, our exploit for the given vulnerability is still able to launch calc.exe through an invocation of WinExec(). The attack code consists of only five counterfeit objects, corresponding to four different vfgadgets (including the main ML-G) from mshtml.dll. Corresponding to the given vulnerability, the used main ML-G can be found as fourth entry in an existing vtable whereas, corresponding to the vcall site of the ML-G, the other three vfgadgets can be found as third entries in existing vtables. The task of calculating the address of WinExec is done in JavaScript code beforehand.

B. Internet Explorer 10 32-bit

Our 32-bit attack code implements the following functionality: *(1)* read pointer to kernel32.dll from IAT; *(2)* calculate pointer to WinExec() in kernel32.dll; *(3)* enter loop that launches calc.exe using WinExec() n times; *(4)* finally, enter an infinite waiting loop such that the browser does not crash.

The attack code does not rely on an array-based ML-ARG-G (recall that in 32-bit ML-ARG-Gs are used instead of ML-Gs); instead, it uses a more complex ML-ARG-G that traverses a linked list of object pointers using a C++ iterator. We discovered this ML-ARG-G in jscript9.dll that is available in every Internet Explorer process. The ML-ARG-G consists of four basic blocks and invokes the function SListBase::Iterator::Next() to get the next object pointer from a linked list in a loop. The assembly code of the ML-ARG-G is given in Listing A.1 in the Appendix.

Figure 9 depicts the layout of the linked list: each item in the linked list consists of one pointer to the next item and another pointer to the actual object. This layout allows for the low-overhead implementation of conditional branches and loops. For example, to implement the loop in our attack code, we simply made parts of the linked list circular as shown in Figure 9. Inside the loop in our attack code, a counter within a counterfeit object is incremented for each iteration. Once the counter overflows, a W-COND-G rewrites the *backward* pointer such that the loop is left and execution proceeds along another linked list. Our attack code consists of 11 counterfeit objects, and 11 linked list items of which two point to the same counterfeit object. Four counterfeit objects overlap and one counterfeit object overlaps with a linked list item to implement the conditional rewriting of a *next* pointer.

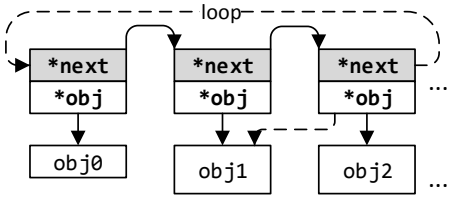

Fig. 9: Schematic layout of the linked list of object pointers the ML-ARG-G traverses in the Internet Explorer 10 32-bit exploit; dashed arrows are examples for dynamic pointer rewrites for the implementation of conditional branches.

This example highlights how powerful linked list-based ML-Gs/ML-ARG-Gs are in general.

C. Firefox 36.0a1 for Linux x64

To demonstrate the wide applicability of COOP, we also created an attack code for the GCC-compiled Firefox 36.0a1 for Linux x64. For this proof of concept, we created an artificial vulnerable application and loaded Firefox's main library *libxul.so* into the address space. Our COOP attack code here invokes system("/bin/sh"). It is comprised of nine counterfeit objects (of which two overlap) corresponding to five different vfgadgets. The attack code reads a pointer to libc.so from the *global offset table* (GOT) and calculates the address of system() from that.

VI. Discussion

We now analyze the properties of COOP, discuss different defense concepts against it, and review our design goals **G-1**–**G-4** from §III-A. The effectiveness against COOP of several existing defenses is discussed afterwards in §VII.

A. Preventing COOP

We observe that the characteristics **C-1**–**C-5** of existing code reuse attack approaches cannot be relied on to defend against COOP (goal **G-1**): in COOP, control flow is only dispatched to existing and address-taken functions within an application through existing indirect calls. In addition, COOP does neither inject new nor alter existing return addresses as well as other code pointers directly. Instead, only existing vptrs (i. e., pointers to code pointers) are manipulated or injected. Technically, depending on the choice of vfgadgets, a COOP

attack may however execute a high ratio of indirect branches and thus exhibit characteristic **C-3**. But we note that ML-Gs (which are used in each COOP attack as central dispatchers) are legitimate C++ virtual functions whose original purpose is to invoke many (different) virtual functions in a loop. Any heuristics attempting to detect COOP based on the frequency of indirect calls will thus inevitably face the problem of high numbers of false positive detections. Furthermore, similar to existing attacks against behavioral-based heuristics [16], [26], it is straightforward to mix-in long "dummy" vfgadget to decrease the ratio of indirect branches.

As a result, COOP cannot be effectively prevented by *(i)* CFI that does not consider C++ semantics or *(ii)* detection heuristics relying on the frequency of executed indirect branches and is unaffected by *(iii)* shadow call stacks that prevent rogue returns and *(iv)* the plain protection of code pointers.

On the other hand, a COOP attack can only be mounted under the preconditions given in §III-B. Accordingly, COOP is conceptually thwarted by defense techniques that prevent the hijacking or injection of C++ objects or conceal necessary information from the attacker, e. g., by applying ASLR and preventing information leaks.

1) Generic Defense Techniques: We now discuss the effectiveness of several other possible defensive approaches against COOP that do not require knowledge of precise C++ semantics and can thus likely be deployed without analyzing an application's source code or recompiling it.

a) Restricting the Set of Legitimate API Invocation Sites: A straightforward approach to tame COOP attacks is to restrict the set of code locations that may invoke certain sensitive library functions. For example, by means of binary rewriting it is possible to ensure that certain WinAPI functions may only be invoked through constant indirect branches that read from a module's IAT (see *CCFIR* [58]). In the best case, this approach could effectively prevent the API calling techniques **W-2** and **W-3** described in §III-E. However, it is also common for benign code to invoke repeatedly used or dynamically resolved WinAPI functions through non-constant indirect branches like `call rsi`. Accordingly, in practice, it can be difficult to precisely identify the set of a module's legitimate invocation sites for a given WinAPI function. We also remark that even without immediate access to WinAPI functions or systems calls COOP is still potentially dangerous, because, for example, it could be used to manipulate or leak critical data.

b) Monitoring of the Stack Pointer: In 64-bit COOP, the stack pointer is virtually never moved in an irregular or unusual manner. For the 32-bit *thiscall* calling convention though, this can be hard to avoid as long as not only vfgadgets with the same fixed number of arguments are invoked. This is a potential weakness that can reveal a COOP attack on Windows x86 to a C++-unaware defender that closely observes the stack pointer. However, we note that it may be difficult to always distinguish this behavior from the benign invocation of functions in the *cdecl* calling convention.

c) Fine-grained Code Randomization: COOP is conceptually resilient against the fine-grained randomization of locations of binary code, e. g., on function, basic block, or instruction level. This is because in a COOP attack, other than for example in a ROP attack, knowing the exact locations of certain instruction sequences is not necessary but rather only the locations of certain vtables. Moreover, in COOP, the attacker mostly misuses the *actual* high-level semantics of existing code. Most vfgadget types, other than ROP gadgets, are thus likely to be unaffected by semantics-preserving rewriting of binary code. Only LOAD-R64-Gs that are used to load x64 argument registers could be broken by such means. However, the attacker could probably oftentimes fall back to x86-style ML-ARG-G-based COOP in such a case.

2) C++ Semantics-aware Defense Techniques: We observe that the control flow and data flow in a COOP attack are similar to those of benign C++ code (goal **G-2**). However, there are certain deviations that can be observed by C++-aware defenders. We now discuss several corresponding defenses.

a) Verification of Vptrs: In basic COOP, vptrs of counterfeit objects point to existing vtables but not necessarily to their beginning. This allows for the implementation of viable defenses against COOP when all legitimate vcall sites and vtables in an application are known and accordingly each vptr access can be augmented with sanity checks. Such a defense can be implemented without access to source code by means of static binary code rewriting as concurrently shown by Prakash et al. [41]. While such a defense significantly shrinks the available vfgadget space, our exploit code from §V-A1 demonstrates that COOP-based attacks are still possible, at least for large C++ target applications.

Ultimately, a defender needs to know the set of allowed vtables for each vcall site in an application to reliably prevent malicious COOP control flow (or at least needs to arrive at an approximation that sufficiently shrinks the vfgadget space). For this, the defender needs *(i)* to infer the global hierarchy of C++ classes with virtual functions and (*ii*) to determine the C++ class (within that hierarchy) that corresponds to each vcall site. Both can easily be achieved when source code is available. Without source code, given only binary code and possibly debug symbols or RTTI metadata[2], the former can be achieved with reasonable precision while, to the best of our knowledge, the latter is generally considered to be hard for larger applications by means of static analysis [20], [21], [24], [41].

b) Monitoring of Data Flow: COOP also exhibits a range of data-flow patterns that can be revealing when C++ semantics are considered. Probably foremost, in basic COOP, vfgadgtes with varying number of arguments are invoked from the same vcall site. This can be detected when the number of arguments expected by each virtual function in an application is known. While trivial with source code, deriving this information from binary code can be challenging [41]. An even stronger (but also likely costlier) protection could be created by considering the actual types of arguments.

[2]*Runtime Type Information* (RTTI) metadata is often linked into C++ applications for various purposes. RTTI includes the literal names of classes and the precise class hierarchy.

In a COOP attack, counterfeit objects are not created and initialized by legitimate C++ constructors, but are injected by the attacker. Further, the concept of overlapping objects creates unusual data flows. To detect this, the defender needs to be aware of the life-cycle of C++ objects in an application. This requires knowledge of the whereabouts of (possibly inlined) constructors and destructors of classes with virtual functions.

c) Fine-grained Randomization of C++ Data Structures: In COOP, the layout of each counterfeit object needs to be byte-compatible with the semantics of its vfgadget. Accordingly, randomizing C++ object layouts on application start-up, e. g., by inserting randomly sized paddings between the fields of C++ objects, can hamper COOP. Also, the fine-grained randomization of the positions or structures of vtables could be a viable defense against COOP.

We conclude that COOP can be mitigated by a range of means that do not require knowledge of C++ semantics. But we regard it as vital to consider and to enforce C++ semantics to reliably prevent COOP. Doing so by means of static binary analysis and rewriting only is challenging as the compilation of C++ code is in most cases a *lossy* process. For example, in binary code, distinguishing the invocation of a virtual function from the invocation of a C-style function pointer that happens to be stored in a read-only table can be difficult. Hence, unambiguously recovering essential high-level C++ semantics afterwards can be hard or even impossible. In fact, as we discuss in more detail in §VII, we know of no binary-only CFI solution that considers C++ semantics precisely enough to fully protect against COOP.

B. Applicability and Turing Completeness

We have shown that COOP is applicable to popular C++ applications on different operating systems and hardware architectures (goal **G-3**). Naturally, a COOP attack can only be mounted in case at least a minimum set of vfgadgets is available. We did not conduct a quantitative analysis on the general frequency of usable vfgadgets in C++ applications: determining the actual usefulness of potential vfgadgets in an automated way is challenging and we leave this for future work. In general, we could choose from many useful vfgadgets in the libraries mshtml.dll (around 20 MB) and libxul.so (around 60 MB) and found the basic vfgadget types ARITH-G, W-G, R-G, LOAD-R64-G, and W-SA-G to be common even in smaller binaries. The availability of ML-Gs/ML-ARG-Gs is vital to every COOP attack. While sparser than the more basic types, we found well-usable representatives, e. g., in Microsoft's standard C/C++ runtime libraries msvcr120.dll and msvcp120.dll (both smaller than 1 MB; dynamically linked to many C and C++ applications on Windows): the virtual function `Scheduler-Base::CancelAllContexts()` with five basic blocks in msvcr120.dll is a linked list-based ML-G and the virtual function `propagator_block::unlink_sources()` with eight basic blocks in msvcp120.dll is an array-based ML-ARG-G. Interestingly, this particular ML-ARG-G is also defined in Visual Studio's standard header file agents.h. In msvcr120.dll, we also found the INV-G `Cancellation-TokenRegistration_TaskProc::_Exec()` that consists of one basic block and is suitable for x86 and x64 COOP.

Given the vfgadget types defined in Table I, COOP has the same expressiveness as unrestricted ROP [46]. Hence, it allows for the implementation of a Turing machine (goal **G-4**) based on memory load/store, arithmetic, and branches. In particular, the COOP examples in §V show that complex semantics like loops can be implemented under realistic conditions.

VII. COOP and Existing Defenses

Based on the discussions in §VI, we now assess a selection of contemporary defenses against code reuse attacks and discuss whether they are vulnerable to COOP in our adversary model. A summary of our assessment is given in Table II.

A. Generic CFI

We first discuss CFI approaches that do not consider C++ semantics for the derivation of the CFG that should be enforced. We observe that all of them are vulnerable to COOP.

The basic implementation of the original CFI work by Abadi et al. [3] instruments binary code such that indirect calls may only go to address-taken functions (coarse-grained CFI). This scheme and a closely related one [59] have recently been shown to be vulnerable to advanced ROP-based attacks [16], [25]. Abadi et al. also proposed to combine their basic implementation with a shadow call stack that prevents call/return mismatches. This extension effectively mitigates these advanced ROP-based attacks while, as discussed in §VI, it does not prohibit COOP.

Davi et al. described a hardware-assisted CFI solution for embedded systems that incorporates a shadow call stack and a certain set of runtime heuristics [15]. However, the indirect call policy only validates whether an indirect call targets a valid function start. As COOP only invokes entire functions, it can bypass this hardware-based CFI mechanism.

CCFIR [58], a CFI approach for Windows x86 binaries, uses a randomly arranged "springboard" to dispatch all indirect branches within a code module. On the baseline, CCFIR allows indirect calls and jumps to target all address-taken locations in a binary and restricts returns to certain call-preceded locations. One of CCFIR's core assumptions is that the attacker is unable to "[...] selectively reveal [s]pringboard stub addresses of their choice" [58]. Göktaş et al. recently showed that ROP-based bypasses for CCFIR are possible given an up-front information leak from the springboard [25]. In contrast, COOP breaks CCFIR without violating its assumptions: the springboard technique is ineffective against COOP as we do not inject code pointers but only vptrs (pointers to code pointers). CCFIR though also ensures that sensitive WinAPI functions (e. g., `CreateFile()` or `WinExec()`) can only be invoked through constant indirect branches. However, as examined in §VI-A1a, this measure does not prevent dangerous attacks and can probably also be sidestepped in practice. In any case, COOP can be used in the first stage of an attack to selectively readout the springboard.

Many system modules in the Microsoft Windows 10 Technical Preview are compiled with *Control Flow Guard* (CFG),

Category	Scheme	Realization	Effective against COOP ?
Generic CFI	Original CFI + shadow call stack [3]	Binary + debug symbols	✗
	CCFIR [58]	Binary	✗
	O-CFI [54]	Binary	✗
	SW-HW Co-Design [15]	Source code + specialized hardware	✗
	Windows 10 Tech. Preview CFG	Source code	✗
	LLVM IFCC [52]	Source code	?
C++-aware CFI	**—various—** [5], [29], [52]	Source code	✓✓✓
	T-VIP [24]	Binary	✗
	VTint [57]	Binary	✗
	vfGuard [41]	Binary	?
Heuristics-based detection	**—various—** [14], [40], [56]	CPU debugging/performance monitoring features	✗✗✗
	HDROP [60]	CPU performance monitoring counters	✗
	Microsoft EMET 5 [34]	WinAPI function hooking	✗
Code hiding, shuffling, or rewriting	STIR [55]	Binary	✗
	G-Free [38]	Source code	✗
	XnR [7]	Binary / source code	?
Memory safety	**—various—** [4]–[6], [13], [36], [45]	Mostly source code	(✓✓✓) - see §VII-E
	CPI/CPS [31]	Source code	✓/✗

TABLE II: Overview of the effectiveness of a selection of code reuse defenses and memory safety techniques (below double line) against COOP; ✓ indicates effective protection and ✗ indicates vulnerability; **?** indicates at least partial protection.

a simple form of CFI. We analyzed the proprietary implementation of Microsoft CFG. In summary, Microsoft CFG ensures that protected indirect calls may only go to a certain set of targets. This set is specified in a module's PE header [42]. If multiple CFG-enabled modules reside in a process, their sets are merged. For system libraries (written in C), this set is mostly comprised of exported functions. For the C++ `mshtml.dll` we discovered that all virtual functions are contained in the set and can thus be invoked from any indirect call site. Accordingly, Microsoft CFG in its current form does not prevent COOP, but also likely not advanced ROP-based attacks like the one by Göktaş et al.

Tice et al. recently described two variants of *Forward-Edge CFI* for the GCC and LLVM compiler suites [52] that solely aim at constraining indirect calls and jumps but not returns. As such, taken for itself, forward-edge CFI does not prevent ROP in any way. One of the proposed variants is the C++-aware *virtual table verification* (VTV) technique for GCC. It tightly restricts the targets of each vcall site according to the C++ class hierarchy and thus prevents COOP. VTV is available in mainline GCC since version *4.9.0*. However, the variant for LLVM called *indirect function-call checks* (IFCC) "[...] does not depend on the details of C++ or other high-level languages" [52]. Instead, each indirect call site is associated with a set of valid target functions. A target is valid if *(i)* it is address-taken and *(ii)* its signature is *compatible* with the call site. Tice et al. discuss two definitions for the *compatibility* of function signatures for IFCC: *(i)* all signatures are compatible or *(ii)* signatures with the same number of arguments are compatible. We observe that the former configuration does not prevent COOP, whereas the latter can still allow for powerful COOP-based attacks in practice as discussed in §VI-A2b.

B. C++-aware CFI

As discussed in §VI, COOP's control flow can be reliably prevented when precise C++ semantics are considered from source code. Accordingly, various source code-based CFI solutions exist that prevent COOP, e. g., GCC VTV as described above, Safedispatch [29], or WIT [5].

Recently and concurrently, three C++-aware CFI approaches for legacy binary code have been proposed: T-VIP [24], vfGuard [41], and VTint [57]. They follow a similar basic approach:

1) identification of vcall sites and vtables (only vfGuard and VTint) using heuristics and static data-flow analysis
2) instrumentation of vcall sites to restrict the set of allowed vtables.

T-VIP ensures at each instrumented vcall site that the vptr points to read-only memory. Optionally, it also checks if a random entry in the respective vtable points to read-only memory. Similarly, VTint copies all identified vtables into a new read-only section and instruments each vcall site to check if the vptr points into that section. Both effectively prevent attacks based on the injection of fake vtables, but as in a COOP attack only actual vtables are referenced, they do not prevent COOP. VfGuard instruments vcall sites to check if the vptr points to the *beginning* of any known vtable. As discussed §VI-A2a, such a policy restricts the set of available vfgadgets significantly, but still cannot reliably prevent COOP. VfGuard also checks the compatibility of calling conventions and consistency of the this-ptr at vcall sites, but this does not affect COOP. Nonetheless, we consider vfGuard to be one of the strongest available binary-only defenses against COOP. VfGuard significantly constraints attackers and we expect it to be a reliable defense in at least some attack scenarios, e. g., for small to medium-sized x86 applications.

C. Heuristics-based Detection

Microsoft EMET [34] is probably the most widely deployed exploit mitigation tool. Among others, it implements different heuristics-based strategies for the detection of ROP [23]. Additionally, several related heuristics-based defenses have been proposed that utilize certain debugging features available in modern x86-64 CPUs [14], [40], [56]. All of these defenses have recently been shown to be unable to detect more

advanced ROP-based attacks [11], [16], [26], [43]. Similarly, the HDROP [60] defense utilizes the *performance monitoring counters* of modern x86-64 CPUs to detect ROP-based attacks. The approach relies on the observation that a CPU's internal branch prediction typically fails in abnormal ways during the execution of common code reuse attacks.

As discussed in §VI-A, such heuristics are unlikely to be practically applicable to COOP and we can in fact confirm that our Internet Explorer exploits (§V-A and §V-B) are not detected by EMET version 5.

D. Code Hiding, Shuffling, or Rewriting

STIR [55] is a binary-only defense approach that randomly reorders basic blocks in an application on each start-up to make the whereabouts of gadgets unknown to an attacker—even if she has access to the exact same binary. As discussed in §VI-A1c, approaches like this do conceptually not affect our attack, as COOP only uses entire functions as vfgadgets and only knowledge on the whereabouts of vtables is required. This applies also to the recently proposed O-CFI approach [54] that combines the STIR concept with coarse-grained CFI.

Execute-no-Read (XnR) [7] is a proposed defense against so-called *JIT-ROP* [49] attacks that prevents code pages from being read. We note that, depending on the concrete scenario, a corresponding JIT-COOP attack could not always be thwarted by such measures as it can suffice to readout vtables and possibly RTTI metadata (which contains the literal names of classes) from data sections and apply pattern matching to identify the addresses of the vtables of interest.

G-Free [38] is an extension to the GCC compiler. G-Free produces x86 native code that (largely) does not contain *unaligned* indirect branches. Additionally, it aims to prevent attackers from misusing *aligned* indirect branches: return addresses on the stack are encrypted/decrypted on a function's entry/exit and a "cookie" mechanism is used to ensure that indirect jump/call instructions may only be reached through their respective function's entry. While effective even against many advanced ROP-based attacks [11], [16], [25], [26], [43], G-Free does not affect COOP.

E. Memory Safety

Systems that provide forms of memory safety for C/C++ applications [4]–[6], [13], [31], [36], [45] can constitute strong defenses against control-flow hijacking attacks in general. As our adversary model explicitly foresees an initial memory corruption and information leak (see §III-B), we do not explore the defensive strengths of these systems in detail. Instead, we exemplarily discuss two recent approaches in the following.

Kuznetsov et al. proposed *Code-Pointer Integrity* (CPI) [31] as a low-overhead control-flow hijacking protection for C/C++. On the baseline, CPI guarantees the spatial and temporal integrity of code pointers and, recursively, that of pointers to code pointers. As in C++ applications typically many pointers to code pointers exist (i. e., each object's vptr), CPI can still impose a significant overhead there. As a consequence, Kuznetsov et al. also proposed *Code-Pointer Separation* (CPS) as a less expensive variant of CPI that specifically targets C++. In CPS, sensitive pointers are not protected recursively, but it is still enforced that "[...] (i) code pointers can only be stored to or modified in memory by code pointer store instructions, and (ii) code pointers can only be loaded by code pointer load instructions from memory locations to which previously a code pointer store instruction stored a value" [31] where *code pointer load/store instructions* are fixed at compile time. Kuznetsov et al. argue that the protection offered by CPS could be sufficient in practice as it conceptually prevents recent advanced ROP-based attacks [11], [16], [26]. We observe that CPS does not prevent our attack, because COOP does not require the injection or manipulation of code pointers. In the presence of CPS, it is though likely hard to invoke library functions not imported by an application. But we note that almost all applications import critical functions. The invocation of library functions through an INV-G could also be complicated or impossible in the presence of CPS. This is however not a hurdle, because, as CPS does not consider C++ semantics, imported library functions can always easily be called without taking the detour through an INV-G as described in §III-E in approach **W-2**.

VIII. Related Work

Since we covered related work throughout the paper, we only briefly review contributions similar to ours in this section. Closely related to our work, several advanced ROP-based attacks were recently demonstrated [11], [16], [25], [26], [43] that bypassed certain coarse-grained CFI systems [3], [58], [59] or heuristics-based systems [14], [23], [40], [56]. However, to the best of our knowledge, we are the first to demonstrate bypasses of the latest defenses CPS [31], T-VIP [24], vfGuard [41], and VTint [57] and the *coarse-grained CFI + shadow call stack* [3] concept. We also regard COOP's tolerance against the fine-grained rewriting, shuffling, and hiding of executable code as unique.

Bosman and Bos presented *Sigreturn Oriented Programming* (SROP) [10], a distinct code reuse attack approach that misuses UNIX signals. SROP is Turing complete and in contrast to ROP does not chain short chunks of instructions sequences. In SROP, the UNIX system call *sigreturn* is repeatedly invoked on an attacker supplied *signal frames* lying on the stack. Accordingly, as prerequisites, the attacker needs to control the stack and needs to be able to divert the control flow such that *sigreturn* is invoked. SROP was not specifically designed to circumvent modern protection techniques, but rather as an easy-to-use and portable alternative to ROP and for implementing stealthy backdoors.

Tran et al. demonstrated that Turing complete *return-to-libc* attacks are possible [53]. In their described attack, a thread's stack is prepared in such a way that certain functions from libc such as `longjmp()` or `wordexp()` are subsequently executed for varying arguments, where each function *returns* to the entry point of its successor. At its core, their approach shares similarities with ours. However, it can conceptually not be used to bypass modern CFI systems. Skowyra et al. demonstrated how the attack by Tran et al. can also be implemented using other libraries than libc [47].

IX. Conclusion

In this paper, we introduced *counterfeit object-oriented programming* (COOP), a novel code reuse attack technique to bypass almost all CFI solutions and many other defenses that do not consider object-oriented C++ semantics. We discussed the specifics of object-oriented programming and explained the technical details behind COOP. We believe that our results contribute to the ongoing research on designing practical and secure defenses against control-flow hijacking attacks, a severe threat that has been around for more than two decades. Our basic insight that higher-level programming language-specific semantics need to be taken into account is a valuable guide for the design and implementation of future defenses. In particular, our results demand for a rethinking in the assessment of defenses that rely solely on binary code.

Acknowledgment

We thank the anonymous reviewers and Herbert Bos for their constructive comments that guided the final version of this paper. This work has been supported by several organization: the German Federal Ministry of Education and Research (BMBF) under support code 16BP12302 (EUREKA project SASER), the German Science Foundation as part of project S2 within the CRC 1119 CROSSING, and the European Unions Seventh Framework Programme under grant agreement No. 609611, PRACTICE project.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control-flow. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 111–124, 2005.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1), 2009.

[3] M. Abadi, M. Budiu, lfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2005.

[4] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, 2010.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, 2008.

[6] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.

[7] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you cant read: Preventing disclosure exploits in executable code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2014.

[8] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, 2014.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[10] E. Bosman and H. Bos. Framing signals—a return to portable shellcode. In *IEEE Symposium on Security and Privacy*, 2014.

[11] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2010.

[13] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[14] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[15] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *DAC*, 2014.

[16] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[18] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[19] L. De Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer Aided Design (FMCAD)*, 2009.

[20] D. Dewey and J. T. Giffin. Static detection of C++ vtable escape vulnerabilities in binary code. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[21] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. SmartDec: Approaching C++ decompilation. In *Working Conference on Reverse Engineering (WCRE)*, 2011.

[22] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium*, 2001.

[23] I. Fratric. Runtime Prevention of Return-Oriented Programming Attacks. http://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf.

[24] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Anual Computer Security Applications Conference (ACSAC)*, 2014.

[25] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[26] 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 *USENIX Security Symposium*, 2014.

[27] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in Comp. Virology*, 2010.

[28] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, 2013.

[29] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[30] N. Joly. Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php, 2013.

[31] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[32] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System V Application Binary Interface: AMD64 architecture processor supplement. http://x86-64.org/documentation/abi.pdf, 2013.

[33] Microsoft. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/, 2006.

[34] Microsoft Corp. Enhanced mitigation experience toolkit (EMET) 5.1. http://technet.microsoft.com/en-us/security/jj653751, November 2014.

[35] Microsoft Developer Network. Argument passing and naming conventions. http://msdn.microsoft.com/en-us/library/984x0h58.aspx.

[36] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management*, 2010.

[37] Nergal. The advanced return-into-lib(c) exploits: PaX case study. http://phrack.org/issues/58/4.html, 2001.

[38] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Anual Computer Security Applications Conference (ACSAC)*, 2010.

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[40] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.
[41] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
[42] M. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1*. Microsoft Press, 6th edition, 2012.
[43] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.
[44] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2014.
[45] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-Sanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.
[46] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2007.
[47] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein. Systematic analysis of defenses against return-oriented programming. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013.
[48] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.
[49] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.
[50] B. Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley, 4th edition, 2013.
[51] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.
[52] 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 *USENIX Security Symposium*, 2014.
[53] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.
[54] M. Vishwath, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
[55] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, 2012.
[56] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2012.
[57] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
[58] 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 *IEEE Symposium on Security and Privacy*, 2013.
[59] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.
[60] H. Zhou, X. Wu, W. Shi, J. Yuan, and B. Liang. HDROP: Detecting ROP attacks using performance monitoring counters. In *Information Security Practice and Experience*. Springer International Publishing, 2014.

APPENDIX

```
mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    ecx
push    esi
mov     esi, ecx
lea     eax, [esi+3ACh]
; -- inlined constructor of iterator --
mov     [ebp+iterator.end], eax
mov     [ebp+iterator.current], eax
; --

loop:
lea     ecx, [ebp+iterator]
call    SListBase::Iterator::Next()
test    al, al
jnz     end

mov     eax, [ebp+iterator.current]
push    [esi+140h] ; push argument field
mov     ecx, [eax+4] ; read object pointer from iterator
mov     eax, [ecx]
call    [eax+4] ; call 2nd virtual function
jmp     loop

end:
pop     esi
mov     esp, ebp
pop     ebp
ret
```

Listing A.1: Assembly code of ML-ARG-G in jscrip9.dll version *10.0.9200.16521* used in exemplary Internet Explorer 10 32-bit exploit: a linked list of object pointers is traversed; a virtual function with one argument is invoked on each object.

Symbol name of vfgadget (mshtml.dll Win. 7 64-bit)	*# in attack code*	*Vfgadget type*	*Function*
`CExtendedTagNamespace::Passivate`	1, 9b	ML-G	array-based main loop
`CCircularPositionFormatFieldIterator::Next`	2, 5, 7, 9a, 10b	LOAD-R64-G	load `rdx` from dereferenced field
`XHDC::SetHighQualityScalingAllowed`	3	ARITH-G	store `rdx&1`
`CWigglyShape::OffsetShape`	4	LOAD-R64-G	load `r9` from field
`CStyleSheetArrayVarEnumerator::MoveNextInternal`	6	LOAD-R64-G	load `r8` from field
`CDataCache<class CBoxShadow>::InitData`	8	W-COND-G	write `r8` to `[rdx]` if `r9` is not zero
`CRectShape::OffsetShape`	10a, 11b	ARITH-G	add `[rdx]` to field
`Ptls6::CLsBlockObject::Display`	11a, 12b	INV-G	invoke field as function pointer

TABLE A.I: Vfgadgets in mshtml.dll *10.0.9200.16521* used in exemplary Internet Explorer 10 64-bit exploit (§V-A); execution splits into paths *a* and *b* after index 8.

Symbol name of vfgadget (mshtml.dll Win. 7 64-bit)	*# in attack code*	*Vfgadget type*	*Function*
`CExtendedTagNamespace::Passivate`	1	ML-G	array-based main loop
`CMarkupPageLayout::IsTopLayoutDirty`	2, 4	LOAD-R64-G	load `edx` from field
`HtmlLayout::GridBoxTrackCollection::GetRangeTrackNumber`	3	ARITH-G	`r8` $= 2 \cdot$ `rdx`
`CAnimatedCacheEntryTyped<float>::UpdateValue`	4	INV-G	invoke field from argument as function pointer

TABLE A.II: Vfgadgets in mshtml.dll *10.0.9200.16521* used in exemplary Internet Explorer 10 64-bit exploit that only uses vptrs pointing to the beginning of existing vtables (§V-A1)

Symbol name of vfgadget	*# in attack code*	*Vfgadget type*	*Function*
`jscript9!ThreadContext::ResolveExternalWeakReferencedObjects`	1	ML-ARG-G	linked list-based main loop
`CDataTransfer::Proxy`	2	W-SA-G	write deref. field to scratch area
`CDCompSwapChainLayer::SetDesiredSize`	3	R-G	load field from scratch area
`CDCompSurfaceTargetSurface::GetOrigin`	4	ARITH-G and W-SA-G	write summation of two fields to scratch area
`CDCompLayerManager::SetAnimationCurveToken`	5	R-G	load field from scratch area
`HtmlLayout::SvgBoxBuilder::PrepareBoxForDisplay`	*loop_entry:* 6, 11	W-G	rewrite *argument field*
`CDXTargetSurface::OnEndDraw`	7, 8	MOVE-SP-G	move stack pointer up
`ieframe!Microsoft::WRL::Callback::ComObject::Invoke`	9	INV-G	invoke function pointer with 2 arguments
`CMarkupPageLayout::AddLayoutTaskOwnerRef`	10	ARITH-G	increment field
`Ptls6::CLsDnodeNonTextObject::SetDurFmtCore`	12	W-COND-G	conditionally write argument to field; rewrites linked list; resumes at *loop_entry* or *loop_exit*
`CDispRecalcContext::OnBeforeDestroyInitialIntersectionEntry`	*loop_exit*	NOP	nop; loops to self

TABLE A.III: Vfgadgets used in exemplary Internet Explorer 10 32-bit exploit (§V-B); vfgadgets taken from mshtml.dll (if not marked differently), jscript9.dll, or ieframe.dll version *10.0.9200.16521*.

Symbol name of vfgadget (libxul.so Linux 64-bit)	*# in attack code*	*Vfgadget type*	*Function*
`nsMultiplexInputStream::Close`	1	ML-G	array-based main loop
`mozilla::a11y::xpcAccessibleGeneric::~xpcAccessibleGeneric` and `js::jit::MVariadicInstruction::getOperand`	2, 4	LOAD-R64-G	load `rsi` from memory
`nsDisplayItemGenericGeometry::MoveBy`	3	ARITH-G	add `[rsi]` to field
`ProfileSaveEvent::AddSubProfile`	5	INV-G	invoke field as function pointer

TABLE A.IV: Vfgadgets used in exemplary Firefox 36.0a1 64-bit exploit (§V-C)