第一部分：介绍

1.内存漏洞（比如缓冲区溢出）
利用代码注入　对策：DEP
2.代码重用攻击（比如 return-oriented pro-gramming (ROP) or return-to-libc）

代码重用攻击，无需注入代码，它们通过滥用驻留在受攻击应用程序的地址空间中的现有代码块称为 gadgets）来诱导恶意程序。

两步：利用内存漏洞劫持控制流；恶意程序行为。

针对第一步：

许多防御技术已经提出，但这些技术通常需要访问甚至更改应用程序的源代码，并可能产生相当大的开销。这妨碍了他们更广泛的部署。

针对第二步：

一种流行的工作方式是通过隐藏，改组或在应用程序中重写应用程序的代码或数据来阻止代码重用攻击; 通常以伪随机的方式。比如 ASLR。将有用代码块对攻击者隐藏起来，绕过这些防御通常需要利用额外的内存泄露或信息泄漏漏洞。

互补方法：

控制流完整性（CFI）：它强制程序的控制流遵循预先确定的或在运行时生成的控制流图（CFG）。

精确 CFI（细粒度 CFI）在概念上是合理的，但存在诸如开销或需要访问源代码的实际障碍，这阻碍了其广泛部署。

不精确的 CFI（粗粒度 CFI）的不同实例化已经提出了 CFI 和相关的运行时检测启发式算法。但这些解决方案可以在设置中被绕过。

3.COOP
一种针对 C ++开发的应用程序的新型代码重用攻击技术。
通过 COOP，我们展示了在 C ++环境中针对代码重用攻击的一系列防御的局限性。我们表明，代码重用防御必须仔细而精确地考虑类层次结构等 C ++语义。
COOP 几乎绕开了所有不了解 C ++语义的 CFI 解决方案。
4.攻击技术概述
现有的代码重用攻击通常在控制流（和数据流）中具有独特的特性，无论应用程序编程的语言如何，都允许进行通用保护。例如，如果能够监控一个应用程序的所有返回指令，同时保持一个完整的影子调用堆栈，那么即使是基于 RaP 的高级攻击也无法安装。
而 **COOP 不同**：
（1）它利用了每个 C ++虚函数有一个常量指针指向它。
如果不精确的 CFI 解决方案不考虑 C ++语义，那么这些函数都可能是有效的间接调用目标，因此可能被滥用。
COOP 完全依赖于通过相应的调用站点作为 gadgets 调用的 C ++虚函数。因此，如果没有使用 C ++开发的应用程序的语义更深入的知识，COOP 的控制流程就无法与良性控制流程区分开来。
（2）在 COOP 中，没有代码指针（例如，返回地址或函数指针）被注入或操作。因此，COOP 可以抵御保护代码指针的完整性和真实性的防御。
而且，**在 COOP 中，gadgets 不相对于堆栈指针，而相对于一组伪造对象上的 this 指针。**相对于 this 指针的寻址意味着无法通过防止堆栈指针指向程序堆的防御来缓解 COOP。
COOP 攻击中使用的伪造对象通常重叠的，使得数据可以从一个小配件传递到另一个小配件。

即使在简单的 COOP 程序中，手动定位伪造对象也会变得复杂。因此，我们实现了一个编程框架，利用 Z3 SMT 求解器自动导出 COOP 程序的对象布局。

第二部分：技术背景

A. 代码重用攻击技术

Return-oriented programming（**ROP**）是一种广泛使用的代码重用攻击技术。基本思想是劫持应用程序的控制流并将其重定向到现有的短指令序列，该指令序列以驻留在目标应用程序的可执行模块中的返回指令（称为 gadgets）结束。gadgets 通常不与可执行模块的原始指令流对齐。每个 gadgets 都执行特定任务，例如执行添加或将值存储到内存中。为了执行恶意 ROP 程序，攻击者将一大块代码指针注入应用程序的地址空间，其中每个指针引用一个 gadget。最后，攻击者滥用内存损坏漏洞，将线程的堆栈指针转向该区域。在下文中，（假）堆栈上注入的代码指针被解释为返回地址，使控制流从一个攻击者选择的 gadgets“返回”到另一个。ROP 可以被认为是旧的 return-to-libc 代码重用攻击技术的一般化，其中攻击者使被劫持的控制流立即“返回”到驻留在例如 libc 中的敏感库函数的条目。

Jump-oriented programming（**JOP**）是 ROP 的一种变体，它使用间接跳转和调用而不是返回指令。在基本 JOP 中，返回指令通过使用 pop-jmp 对的组合来模拟。此外，JOP 攻击不一定需要堆栈指针作为基址寄存器来引用代码指针。特别是，可以使用具有通用寄存器的“更新 - 加载 - 分支”序列。术语面向呼叫编程（**COP**）有时也用于指代采用间接调用的 ROP 派生技术。

尽管这些代码重用攻击技术非常强大，并且 return-to-libc， ROP 和 JOP 甚至已被证明可以在现实场景中实现图灵完备的恶意操作，但它们不同在普通程序执行的几个微妙方面，可以利用它来检测它们的执行。这在 § III-A 中有更详细的讨论。

B. 控制流完整性

Control-Flow Integrity 即 CFI 可以作为防御代码重用攻击的一般方法。它一般被用来指代在本机程序中检测间接分支以阻止代码重用攻击的概念。通常，**CFI 的实施是两步过程：**

（1）确定程序近似的控制流图 CFG X'

（2）通过运行时检查来检测程序（的子集）的间接分支，来确保控制流符合 X'

控制流图 X'可以被动态或者静态的确定，在源代码或者二进制代码级别。X'应该是固有的 CFG X 的一个子图。如果 X 和 X'相等，攻击者通常难以以不符合程序源代码语义的方式转移控制流。CFI 检查通常通过将 ID 分配给程序中的所有可能的间接分支位置来实现。在运行时，在间接跳转之前，如果检查到目标 ID 与 X'一致，分支有效。如果相同的 ID 分配给了程序中大多数 address-taken 函数，运行时返回时不会严格限制符合调用堆栈，这样的话就叫做粗粒度 CFI。最近研究表**粗粒度 CFI 不能够避免 ROP 攻击**。

C. 二进制级别的 C++代码

由于我们的攻击针对 C++应用，我们简要介绍 C++的基本概念并且描述他们是如何通过

编译器实现的。

在 C++和其他面向对象编程语言中，程序员定义定制的类型叫做类。抽象地说，一个类由成员数据字段和成员函数组成。类的一个具体实例叫做对象。继承和多态是面向对象的概念：新的类可以派生自一个或多个已经存在的类，从基类继承至少所有可以见到的数据域和函数。因此，在一般的例子中，一个对象可以作为其实际类的实例访问，也可以作为其任何基类(直接的和非直接的)的实例访问。在 C ++中，可以将成员函数定义为虚函数。可以在派生类中重写继承的虚函数的实现。在对象上调用虚函数总是调用对象类的特定实现，即使该对象是作为其基类之一的实例进行访问的。这被称为多态性。

C++编译器实现了在虚表 vtables 帮助下对虚函数的调用(vcalls)。**vtable 是指向类的所有可能继承的虚函数的指针数组**。因此，每一个虚函数在一个应用中都是 addresss-taken 的。**具有至少一个虚函数的类的每个对象都包含一个指向相应 vtable 的指针（偏移+0）。该指针称为 vptr。**通常，Windows x64 上的 vcall 由编译器转换为类似于以下内容的指令序列：

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

在这里，rcx 是对象的 this 指针，也就是下边的 this-ptr。首先，对象的 vptr 是从 offset+0 加载，即从 this-ptr 加载到 rdx。接下来，在给定的例子中，对象的 vtable 的第二个条目通过解引用rdx+8 来调用。因此，这个特殊的 vcall site 通常调用给定 vtable 的第二个条目。

第三部分：COOP

COOP 是一个针对 C++或者其他可能的面向对象语言开发应用的代码重用攻击。我们注意到今天许多著名的被攻击程序都是使用C++写的(或者主要部分是 C++写的)。包括 Microsoft Internet Explorer,Google Chrome, Mozilla Firefox,Adobe Reader, MicrosoftOffice, LibreOffice 和 OpenJDK 等。

在下面，我们首先陈述我们的设计目标和 COOP 的攻击者模型，然后我们描述 COOP 攻击的实际构建块。出于简洁的原因，本节的其余部分将重点放在 Microsoft Windows 和 x86-64 体系结构上作为运行时环境。 COOP 概念通常适用于在任何操作系统上运行的 C ++应用程序；它还可以扩展到其他架构。

A. COOP 目标

通过 COOP，我们的目标是展示创建强大的代码重用攻击的可行性，这些攻击没有显示现有攻击方法的特征。即使是高级的现有的 return-to-libc，ROP，JOP 或者 COP 攻击的变体，也要依赖于控制流或数据流模式，在常规代码中这些数据流或者控制流很少或从未遇到过。他们包括：

C-1 间接调用/跳转到非 address-taken 位置

C-2 和调用栈不一致的 return

C-3 过度使用间接跳转

C-4 pivoting 栈指针（可能是暂时的）

C-5 注入新的代码指针或操作现有的代码指针

这些特性仍然允许实现有效的，低级的和编程语言无关的保护。 例如，维护一个完整的影子调用堆栈，足以抵御几乎所有基于 ROP 的攻击。

利用COOP，我们证明了一般依赖 C-1 到 C-5 特性来设计代码重用防御是不够的;我们相应地**为 COOP 定义了以下**目**标：**

G-1 并不暴露出 C-1 到 C-5 的特征

G-2 展示出和良性 C++代码执行相似的控制流和数据流

G-3 广泛应用到 C++应用中

G-4 在实际条件下实现图灵完备

B．攻击模型

一般情况下，针对 C++应用的代码重用攻击通常从劫持一个 C++对象和它的 vptr 开始。攻击者通过利用一个空间或者时间上的内存损坏漏洞来实现这一点，比如溢出和 C++对象毗邻的数组或者一个 use-after-free 条件。接下来程序在被劫持对象上调用虚函数，攻击者控制的 vptr 被解除引用，并且一个 vfptr 从攻击者选择的位置内存中加载。在这个点上，攻击者能够有效控制目标程序中对应线程的程序计数器（x64 中 rip）。对代码重用攻击来说，**控制程序计数器**是两个需求之一。另一个需求是**获得（部分）关于**目**标程序地址空间布局的实现**。依赖于上下文，这个可能存在不同的技术实现。

对于 COOP（防御）来说，我们假设攻击者控制了一个 C++对象和它的 vptr，他就可以推断这个对象的基址或其他足够大小的辅助缓冲区。此外，他需要能够推断出一组 C++模块的基地址，这些模块的二进制布局（部分地）为他所知。

C．COOP 攻击的基本方法

每次 COOP 攻击都是通过劫持目标应用程序的 C ++对象之一开始的。我们称之为初始对象。直到攻击者控制程序计数器，COOP 攻击与其他代码重用攻击没有太大差别：在传统的 ROP 攻击中，攻击者通常利用她对程序计数器的控制来首先操纵堆栈指针并随后执行一系列简短的，以 return 终止的 gadgets。相反，在 COOP 中，应用程序中存在的虚拟函数会在攻击者精心安排的伪造 C ++对象上重复调用。

COOP基本方法

1.劫持一个目标应用程序的C++对象，称之为初始对象(initial object)，至此控制了PC

2.小心组织伪造的C++对象，重复调用应用中存在的虚函数

这和传统的**ROP攻击**很像：

1.劫持PC

2.操控栈帧，执行一系列短的，return结尾的gadgets

（1）伪造对象

伪造对象携带攻击者选择的 vptr 和一些攻击者选择的数据字段。伪造对象不是由目标应用程序创建的，而是由攻击者批量注入的。尽管基于 ROP 的攻击中的有效载荷通常由与其他数据交织的伪返回地址组成，但在 COOP 攻击中，有效载荷由伪造对象和可能的附加数据组成。类似于传统的 ROP 有效载荷，包含所有伪造对象的 COOP 有效载荷通常被写为一个相干块到单个攻击者控制的存储器位置。

（2）Vfgadgets

Vfgadgets：COOP 攻击中使用的虚函数

对于其他代码重用攻击，攻击者通过源代码分析或二进制代码的逆向工程在实际攻击之前识别应用程序中的有用 vfgadgets。即使源代码可用，也需要在二进制级别确定 vfgadget 类的实际对象布局，因为编译器可能会删除或填充某些字段。只有这样攻击者才能注入兼容的伪造对象。

接下来三个小节是一个写入动态计算地址的 COOP 攻击代码。D->F 是对这个简单模型的扩展。

（3）The Main Loop

为了在不违反目标 G-1 和 G-2 的情况下重复调用虚函数，每个 COOP 程序基本上依赖于特殊的主循环 vfgadget（ML-G）。**ML-G 的定义**如下：

一种虚函数，它遍历指向 C ++对象的指针的容器（例如，C 形式的数组或向量），并在每个对象上调用虚函数。

考虑下面的程序：

```
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();
        delete students;
    }
};
```

ML-G

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`.

类 Course 有一个字段 students，指向一个数组，数组里是指向 student 的指针。当一个 Course 对象被销毁时，虚析构函数执行，就会通过调用每个 Student 对象的虚函数 decCourseCount（）来告知其课程不存在。

初始对象的布局

攻击者使初始对象类似 ML-G 类的对象。对于我们的示例 ML-G，初始对象的布局如下：

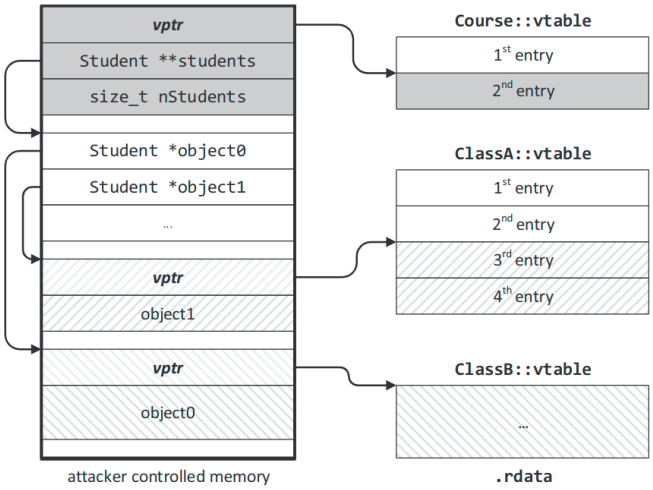

Fig. 2: Basic layout of attacker controlled memory (left) in a COOP attack using the example ML-G `Course::~Course`. The initial object (dark gray, top left) contains two fields from the class `Course`. Arrows indicate a *points-to* relation.

它的 vptr 设置为指向现有的 vtable，其中包含对 ML-G 的引用，使得受攻击者控制的第一个 vcall 指向 ML-G。相反，在基于 ROP 的攻击中，受攻击者控制的第一个 vcall 通常会导致 gadgets 将堆栈指针移动到攻击者控制的内存。初始对象包含 ML-G 类的字段的子集；即，使 ML-G 按预期工作所需的所有数据字段。

对于我们的示例 ML-G，初始对象包含 Course 类的 students 和 nStudents;students 被设置成指向一个数组，数组里的指针指向伪造对象（object0 和 object1），而 nStudents 设置为伪造对象的总数。这使得每个伪造对象调用一个攻击者选择的 vfgadget。请注意攻击者如何控制每个伪造对象的 vptr。图 3 示意性地描绘了 COOP 攻击中的控制流转换：

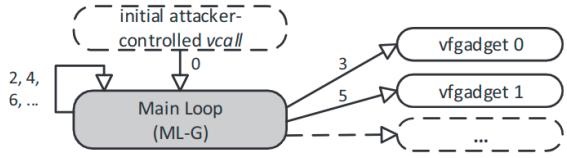

Fig. 3: Schematic control flow in a COOP attack; transitions are labeled according to the order they are executed.

（4）伪造 Vptrs

COOP 攻击中的控制流和数据流应该类似于常规 C++程序（G-2）。因此，我们**避免引入假 vtable 并重用现有的 vtable**。理想情况下，所有伪造对象的 vptrs 应指向现有 vtable 的开头。根据目标应用程序，可能很难找到在给定 vcall 站点固定的偏移量处具有有用条目的 vtable。

对于我们的示例 ML-G，伪造对象被视为抽象类 Student 的实例，对于每一个伪造对象，第二项对应于 decCourseCount（）。（第一个对应于 incCourseCount（）。）这里，COOP 攻击将

理想地只使用第二项。当然，这大大缩小了可用的 vfgadgets 集。

这个约束可以通过放宽目标 G-2 来回避，并且**让假冒对象的 vptrs 不一定指向现有 vtable 的确切开始，而是指向某些正或负偏移**，如图 2 中的对象 1 所示。当使用这种伪造的 vptrs 时，可以从给定的 ML-G 调用任何可用的虚函数。

（5）重叠的伪造对象

我们已经展示了如何在给定 ML-G 的情况下调用任意数量的虚函数（vfgadgets），同时控制流和数据流类似于良性 C ++代码的执行。

ARITH-G（算术）和 WG（写入存储器）类型的两个示例性 vfgadgets 在图 4 中给出：

```
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.

在 Exam :: updateAbsoluteScore（）中，字段分数被设置为三个其他字段的总和；在 SimpleString :: set（）中，字段缓冲区用作写操作中的目标指针。结合使用，这两个 vfgadgets 可用于将攻击者选择的数据写入动态计算的内存地址。为此，需要两个重叠的伪造物体，它们的对齐如图 5 所示：

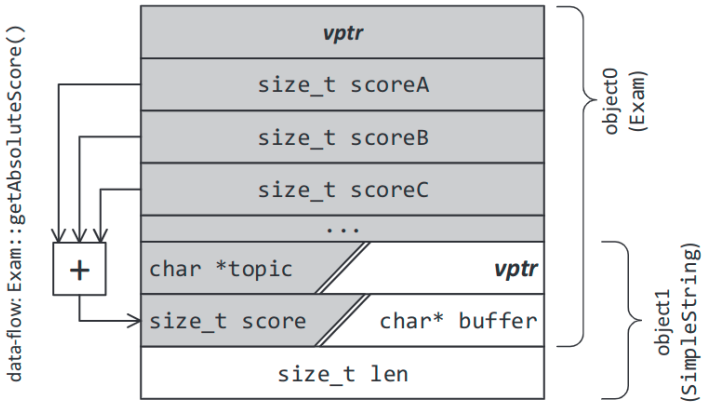

Fig. 5: Overlapping counterfeit objects of types `Exam` and `SimpleString`

这里的关键思想是 object0 中的字段 score 和 object1 中的 buffer 共享相同的内存。这

样，Exam :: updateAbsoluteScore（）中 object0 的字段求和结果将写入 object1 的字段 buffer。请注意，在技术上，这也使对象 0 的 topic 和对象 1 的 vptr 重叠。由于攻击者不使用 object0 的 topic 则无需理会。当然，在我们的示例中，攻击者可能不仅希望通过 object1.buffer 来控制写操作的目标地址。还有源地址。为此，她需要能够为 vfgadget SimpleString :: set（）设置参数。接下来将描述如何在 COOP 中实现这一点。

D．给 Vfgadgets 传递参数

由于不同的默认调用约定，我们在下面描述了将参数传递给 x64 和 x86 的 vfgadgets 的不同技术。

（1）Windows x64 传参方法

在 Windows 上的默认 x64 调用约定中，函数的前四个（非浮点）参数通过寄存器 rcx，rdx，r8 和 r9 传递。。如果有超过四个参数，则附加参数将通过堆栈传递。对于 C ++代码，this-ptr 作为第一个参数传递给 rcx。所有四个参数寄存器都被定义为调用者保存; 无论被调用者的实际参数数量是多少。因此，虚函数通常使用 rdx，r8 和 r9 作为临时寄存器，并且在返回时不会恢复或清除它们。这种情况使得在 x64 上传递 vfgadgets 的参数变得简单：**首先，执行一个 vfgadget，将相应的伪造对象的一个字段加载到 rdx，r8 或 r9 中。接下来，执行 vfgadget，将这些寄存器的内容解释为参数。**

其他平台

在 GCC 使用的默认 x64 C ++调用约定中，例如，在 Linux 上，函数的前六个参数通过寄存器而不是前四个寄存器传递。理论上，这应该**使得在 Linux x64 上创建的 COOP 攻击比在 Windows x64 上更简单**，因为可以使用两个额外的寄存器在 vfgadgets 之间传递数据。实际上，在创建我们的示例漏洞（参见 § V）期间，我们没有遇到两个平台之间的巨大差异。

虽然我们没有在诸如 ARM 或 MIPS 之类的 RISC 平台上进行实验，但我们期望我们的 x64 方法直接扩展到这些，因为在 RISC 调用约定中，参数也主要通过寄存器传递

（2）Windows x86 传参方法

Windows 上的标准 x86 C ++调用约定是 thiscall：**所有常规参数都通过堆栈传递**，而 this-ptr 在寄存器 ecx 中传递; 前; 被调用者负责从堆栈中删除参数。因此，所描述的 x64 方法不适用于 x86。

在我们针对 Windows x86 的方法中，与 x64 相反，我们依赖于将参数传递给 vfgadgets 的主循环（ML-G）。更确切地说，32 位 ML-G 应该将初始对象的一个字段作为参数传递给每个 vfgadget。攻击者可以使用以下两种方法之一将所选参数传递给 vfgadgets：

1）将参数字段**固定**为指向可写的临时区域。

2）**动态**重写参数字段。

在方法 A-2 中，初始对象的参数字段不像方法 A-1 那样固定。相反，它在执行 COOP 攻击期间被动态重写。这允许攻击者将任意参数传递给 vfgadgets; 而不是指向方法 A-1 的任意

数据的指针。对于这种方法，自然地，需要可用的 WG。如上所述，我们发现仅使用字段的 vfgadgets 很少见。因此，攻击者通常最初会遵循方法 A-1 并在需要时在其上实现 A-2 风格的参数编写。

a）传递多个参数并平衡栈

到目前为止，我们已经描述了如何使用 Windows x86 上的 ML-ARG-G 主循环 gadget 将单个参数传递给每个 vfgadget。当然，可能需要或必须将多个参数传递给 vfgadget。这样做很简单：**ML-ARG-G 将一个参数推送到每个 vfgadget。如果 vfgadget 不期望任何参数，则即使在返回 vfgadget 之后，推送的参数仍保留在堆栈的顶部**。这有效地将堆栈指针永久地向上移动一个槽，如图 7 中③所示。这种技术允许攻击者逐渐“堆积”堆栈上的参数，如图 7 ④所示在调用 vfgadget 之前需要多个参数。该技术仅适用于使用 ebp 而非 esp 访问堆栈上的局部变量的 ML-ARG-Gs（即，没有帧指针省略），否则 ML-ARG-G 的堆栈帧被破坏。

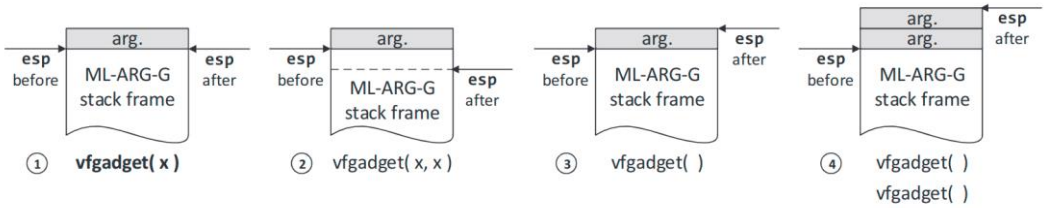

Fig. 7: Examples for stack layouts *before* and *after* invoking vfgadgets under an ML-ARG-G (*thiscall* calling convention). The stack grows upwards. ① vfgadget with one argument: the stack is balanced. ② vfgadget with two arguments: `esp` is moved down. ③ vfgadget without arguments: `esp` is moved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

所描述的在 32 位环境中将多个参数传递给 vfgadgets 的技术**也可用于在 64 位**环境中向 vfgadgets 传递三个以上的参数。

b）其他平台

默认 x86 C++调用约定所使用的 GCC，例如，在 Linux，不是 thiscall 但是 cdcel：所有参数包括 this-ptr 都是通过栈传递的。 不是被调用者，而是调用者负责清理堆栈。因此，§III-D2a 中描述的**“堆积”参数的技术不适用于 Linux x86 和其他 POSIX x86 平台上的 GCC 编译（和兼容）C++应用程序**。相反，对于这些平台，我们建议使用 ML-ARG-Gs，它们不会将一个而是很多个可控参数传递给 vfgadgets。从概念上讲，在 cdecl 调用惯例上，向函数传递过多参数不会破坏栈。或者，可以在攻击期间切换 ML-ARG-Gs，这取决于需要控制 vfgadget 的哪些参数。

E．调用 API 函数

代码重用攻击的最终目标通常是将攻击者选择的参数传递给关键 API 函数或系统调用，例如 Windows API（WinAPI）函数，如 WinExec（）或 VirtualProtect（）。我们确定了以下**在 COOP 攻击中调用 WinAPI 函数的方法：**

w-1：使用一个能合法调用感兴趣的 WinAPI 函数的 vfgadget。

w-2：调用 WinAPI 函数就像从 COOP 主循环中调用一个虚函数。

w-3：使用一个调用 C 风格函数指针的 vfgadget。

虽然方法 W-1 在某些场景下对某些 WinAPI 函数可能是实用的，但在**大多数情况下它是**

不可行的。例如，调用 WinExec（）的虚函数应该接近不存在。

方法 W-2 易于实现：**可以制作伪造对象，其 vptr 不指向实际的 vtable，而是指向已加载模块的导入表（IAT）或导出表（EAT）**，以便 ML-G 将 WinAPI 函数作为虚函数调用。注意，IAT，EAT 和 vtable 都是函数指针的数组，通常位于只读存储器中; 因此，它们原则上是兼容的数据结构。这很简单，该方法有两个重要的**缺点**：（i）它违背了我们的目标 G-2，因为在 vcall 站点调用 C 函数而没有引用合法的 vtable; 和（ii）对于 x64，由于给定的 C++调用约定，相应的伪造对象的 this-ptr 总是作为第一个参数传递给 WinAPI 函数。例如，这种情况有效地阻止了将有用的命令行传递给 WinExec（）。但是，对于其他 WinAPI 函数，这可能会有所不同。例如，使用 this-ptr 作为第一个参数调用 VirtualProtect（）仍允许攻击者将相应伪造对象的内存标记为可执行。请注意，VirtualProtect（）更改第一个参数指向的内存区域的内存访问权限。除第一个之外的其他参数可以按照第 III-D1 节中针对 x64 的描述进行传递。对于 x86，可以使用该技术传递所有参数 § III-D2

对于 W-3 方法，需要一种特殊类型的 vfgadget：**一个用非常量参数调用 C 风格函数指针的虚函数**。我们将这种类型的 vfgagdet 称为 INV-G，图 8 给出了一个例子：虚函数 GuiButton :: clicked（）调用字段 GuiButton :: callbackClick 作为 C 风格的函数指针。这个特定的 vfgadget 允许使用至少三个攻击者选择的参数调用任意 WinAPI 函数。请注意，根据 INV-G 的实际汇编代码，第四个参数可能通过 r9 传递给 x64。x86 和 x64 的附加堆栈绑定参数也可以是可控的，具体取决于堆栈的实际布局。通过 INV-G 调用 WinAPI 函数通常应该是选择的技术，因为这**比 W-1 更灵活并且比 W-2 更隐蔽**。INV-G 通常还能实现从 C++到 C 代码（例如，到 libc）的看似合法的转移。在不利方面，我们发现 **INV-G 是相对的罕见的**整体。对于我们在 § V 中讨论的现实世界的利用示例，我们总是可以选择多个合适的。

```
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.

F. 实现条件分支和循环

到目前为止，我们已经描述了实际安装 COOP 代码重用攻击所需的所有构建块。由于我们不仅要求 COOP 隐蔽，而且要在现实条件下实现图灵完备（目标 G-4），我们现在描述 COOP 中条件分支和循环的实现。

在 COOP 中，程序计数器是伪造对象指针容器的索引。对于 ML-G 主循环中的每次迭代，程序计数器递增。程序计数器可以是普通的整数索引，如我们的示例性 ML-G Course ::Ëœ ~TheCourse 中的，或者可以是更复杂的数据结构，例如 C++链表的迭代器对象。在 COOP 中**实现条件分支通常可以以两种方式实现：通过（i）程序计数器的条件递增/递减或（ii）容器中的下一个伪造对象指针的条件操作。两者都可以在条件写入 vfgadget 的情况下实现，我们将其称为 W-COND-G。**这个 vfgadget 类型的一个例子是图 6 中的 Student2 :: getLatestExam（）。从清单 1 中函数汇编代码的第 3-7 行可以看出，可控写操作仅在[this-ptr + 8]≠0 的情

况下执行。使用此语义，攻击者可以重写 COOP 程序计数器或在某个值不为空的情况下，即将发生的伪造对象指针。如果程序计数器存储在堆栈中（例如，在 ML-G 的堆栈帧中）并且堆栈的地址未知，则移动第 III-D2a 段中描述的堆栈指针的技术可用于重写它。

鉴于能够有条件地重写程序计数器，实现具有退出条件的循环也成为可能。

第四部分：COOP 框架

实施 COOP 攻击的三步：

（i）识别 vfgadgets

（ii）使用识别的 vfgadgets 实现攻击语义

（iii）在缓冲区中安排可能重叠的伪造对象

由于各个步骤繁琐且难以手动执行，因此我们使用 Python 脚本语言创建了一个框架，该框架自动执行步骤（i）和（iii）。该框架极大地促进了我们针对 Internet Explorer 和 Firefox 的示例漏洞的开发。

A．使用基本符号执行查找 vfGadget

为了识别应用程序中有用的 vfgadgets，我们的 **vfgadget 搜索器**仅依赖于二进制代码，并可选择调试符号。C ++模块中的每个虚函数都被视为潜在的 vfgadget。**搜索器使用调试符号静态识别 C ++模块中的所有 vtable，或者如果这些符号不可用，则使用一组简单但有效的启发式算法**。我们的启发式算法将每个函数指针的数组视为潜在的 vtable。搜索者检查所有已识别的虚拟函数，其基本块的数量不超过某个限制。在实践中，我们发现通常只考虑具有**一个或三个基本块的虚函数作为潜在的 vfgadgets** 是足够和方便的; 唯一的例外是 ML-Gs 和 ML-ARG-Gs，由于所需的循环通常由更多的基本块组成。使用短 vfgadgets 是有利的，因为它们的语义更容易自动评估，并且它们通常表现出较少的不需要的副作用。但包括长 vfgadgtes 可以用来欺骗基于启发式代码重用攻击检测方法。

搜索器以单一静态分配（SSA）形式总结了 vfgadget 中每个基本块的语义。这些总结以紧凑且易于分析的形式反映了基本块的 I / O 行为。搜索器依赖于 METASM 二进制代码分析工具包的回溯特性，它在基本块级别上执行符号执行。为了识别有用的 vfgadgets，搜索器在潜在的 vfgadgets 基本块的 SSA 表示上应用过滤器。

B．使用 SMT 求解器对齐重叠对象

每个 COOP“程序”由其伪造对象的顺序和定位来定义，每个伪造对象对应于某个 vfgadget。伪造对象的重叠是 COOP 的一个整体概念; 它通过伪造对象的字段实现 vfgadgets 之间的即时数据流。在二进制级别上手动获取重叠伪造对象的对齐是一项耗时且容易出错的任务。因此，我们创建了一个 COOP 编程环境，如果可能的话，自动将所有给定的伪造对象正确对齐在固定大小的缓冲区中。在我们的编程环境中，“程序员”**定义了伪造对象和标签**。可以将标签分配给伪造对象内的任何字节。当为不同对象中的字节分配相同的标签时，编程环境会注意这些字节映射到最终缓冲区中的相同位置，同时确保具有不同标签的字节映射到

不同的位置。没有标签的字段反过来保证永不重叠。这些约束通常是可以满足的，因为伪造对象内的实际数据通常是稀疏的。

第五部分：概念漏洞证明

A．Internet Explorer 10 64-bit

我们的 64 位 COOP 攻击代码仅依赖于 mshtml.d11 中包含的 vfgadgets，可以在每个 Internet Explorer 进程中找到; 它实现了以下功能：（1）从 IAT 读取指向 kerne132.d11 的指针; （2）在 kerne132.dll 中计算指向 WinExec（）的指针; （3）从 KUSER_SHARED_DATA 数据结构中读取当前的滴答计数;（4）如果 tick count 为 odd，则使用 WinExec（）else 启动 calc.exe，执行备用执行路径并启动 mspaint.exe。

该攻击代码是由 17 个拥有伪造 vptrs 的伪造对象和 4 个纯粹数据容器的伪造对象组成。总共使用了八种不同的 vfgadgets; 包括一个 LOAD-R64-G，用于通过解除一个使用五次的字段的引用来加载 rdx。攻击代码基于类似于图 1 中给出的示例性 ML-G 迭代一个普通的对象指针数组。ML-G 有四个基本块，是八个 vfgadgets 中最大的一个。依赖于当前 tick count 的条件分支是通过重写下一个对象指针来实现，使得 ML-G 被递归地调用用于伪造对象指针的替代阵列。总之，攻击代码包含八个重叠的伪造对象，我们使用 15 种不同的标签在我们的编程环境中创建它。

仅使用指向 vtables 开头的 vptr 的攻击变体：

所描述的 64 位攻击代码依赖于伪造的 vptrs（参见 § III-C4），它们不一定指向现有 vtable 的开头，而是指向它们的正偏移或负偏移。作为概念证明，我们仅在上面开发了一种隐含的攻击代码变体使用指向现有 vtable 开头的 vptrs。因此，在每个 vcall 站点，我们被限制为与相应的固定 vtable 索引兼容的一组虚函数。在此约束下，我们对给定漏洞的利用仍然能够通过调用 WinExec（）来启动 calc.exe。攻击代码只包含五个伪造对象，对应于 mshtml.dll 中的四个不同的 vfgadgets（包括主 ML-G）。对应于给定的漏洞，使用的主 ML-G 是现有 vtable 第四个条目，而对应于 ML-G 的 vcall 站点，其他三个 vfgadgets 可以作为现有 vtable 中的第三个条目。计算 WinExec 地址的任务是事先在 JavaScript 代码中完成的。

B．Internet Explorer 10 32-bit

我们的 32 位攻击代码实现了以下功能：（1）从 IAT 读取指向 kerne132.dll 的指针; （2）在 kerne132.dll 中计算指向 WinExec（）的指针; （3）进入使用 WinExec（）n 次启动 calc.exe 的循环; （4）最后，进入一个无限的等待循环，使浏览器不会崩溃。

攻击代码不依赖于基于阵列的 ML-ARG-G（回想一下，使用 32 位 ML-ARG-G 代替 ML-Gs）;相反，它使用更复杂的 ML-ARG-G，它使用 C++迭代器遍历对象指针的链接列表。我们在 jscript9.dll 中发现了这个 ML-ARG-G，它可以在每个 Internet Explorer 进程中使用。ML-ARG-G 由四个基本块组成，并调用函数 SListBase :: Iterator :: Next（）从循环中的链表中获取下一个对象指针。

图 9 描绘了链表的布局：链表中的每个项都包含一个指向下一个项的指针和另一个指向实际对象的指针。此布局允许条件分支和循环的低开销实现。例如，要在我们的攻击代码中实现循环，我们只需将链表的一部分制作成圆形，如图 9 所示。在我们的攻击代码的循环

内部，伪造对象内的计数器在每次迭代时递增。一旦计数器溢出，W-COND-G 就会向后重写指针使得循环离开并且执行沿着另一个链表继续。我们的攻击代码包含 11 个伪造对象和 11 个链接列表项，其中两个指向同一个伪造对象。四个伪造对象重叠，一个伪造对象与链接列表项重叠，以实现下一个指针的条件重写。

C． Firefox 36.0a1 for Linux x64

为了证明 COOP 的广泛适用性，我们还为 GCC 编译的 Firefox 36.0al for Linux x64 创建了一个攻击代码。为了证明这一点，我们创建了一个易受攻击的虚拟应用程序，并将 Firefox 的主库 libxul.so 加载到地址空间中。我们的 COOP 攻击代码在这里调用 system（“/ bin / sh”）。它由九个伪造对象（其中两个重叠）组成，对应于五个不同的 vfgadgets。攻击代码读取指向 libc 的指针。所以从全局偏移表（GOT）中计算出系统（）的地址。

第六部分：讨论

A．防御 COOP

我们观察到不能依赖于现有代码重用攻击方法的特征 C1-C5 来防御 COOP（G1）：在 COOP 中，控制流仅通过现有的间接调用分派给应用程序内的现有的并且 address-taken 函数。此外，COOP 既不会注入新的也不会直接修改现有的返回地址以及其他代码指针。相反，仅操纵或注入现有的 vptrs（即指向代码指针的指针）。从技术上讲，根据 vfgadgets 的选择，COOP 攻击可能会执行高比例的间接分支，从而表现出特征 C-3。但我们注意到 ML-Gs（在每次 COOP 攻击中用作中央调度程序）是合法的 C ++虚函数，其最初目的是在循环中调用许多（不同的）虚函数。因此，任何基于间接调用的频率检测 COOP 的启发式算法都将不可避免地面临大量误报检测的问题。类似于现有的针对基于行为的启发式攻击的攻击，可以直接混合使用长“dummy”vfgadget 来降低间接分支的比率。

因此，**COOP 不会被这些方法有效防御**：（i）不考虑 C ++语义的 CFI 或（ii）依赖于执行间接分支的频率的检测启发式算法（iii）不受阻止恶意返回的影子调用栈的影响（iv）代码指针的简单保护。

另一方面，COOP 攻击只能在 § III-B 中给出的先决条件下进行。因此，COOP 在概念上受到防御技术的阻碍，防御技术可以防止劫持或注入 C ++对象或隐藏攻击者的必要信息，例如，通过应用 ASLR 和防止信息泄漏。

（1）通用防御技术：

我们现在讨论针对 COOP 的其他几种可能的防御方法的有效性，这些方法不需要精确的 C ++语义知识，因此可以在不分析应用程序的源代码或重新编译它的情况下进行部署。

a）限制合法 API 调用站点集

防御 COOP 攻击的直接方法是限制可能调用某些敏感库函数的代码位置集。例如，通过

二进制重写，可以确保某些 WinAPI 函数只能通过从模块的 IAT 读取的常量间接分支来调用。在最好的情况下，这种方法可以有效地防止§III-E 中描述的 API 调用技术 W-2 和 W-3。但是，良性代码通过非常量间接分支（如调用 rsi）调用重复使用或动态解析的 WinAPI 函数也很常见。因此，实际上，对于给定的 WinAPI 函数，**可能难以精确地识别模块的合法调用站点的集合**。我们还指出，即使没有对 WinAPI 函数的立即访问或系统调用，COOP 仍然存在潜在危险，因为，例如，它可能用于操纵或泄漏关键数据。

b）监视栈指针

在 64 位 COOP 中，堆栈指针实际上不会以不规则或不寻常的方式移动。但是对于 32 位 thiscall 调用约定，只要不仅调用具有相同固定数量参数的 vfgadgets，就很难避免这种情况。但是对于 32 位 thiscall 调用约定，只要不仅调用具有相同固定数量参数的 vfgadgets，就很难避免这种情况。这是一个潜在的弱点，可以将 Windows x86 上的 COOP 攻击暴露给 C ++ - 一个密切关注堆栈指针的不知情的防御者。但是，我们注意到，**可能很难始终将此行为与 cdecl 调用约定中的函数的良性调用区分开来**。

c）细粒度代码随机化

COOP 在概念上对二进制代码的位置的细粒度随机化具有弹性，例如，在函数，基本块或指令级上。只是因为在 COOP 攻击中，除了例如在 ROP 攻击中，知道某些指令序列的确切位置不是必需的，COOP 攻击仅仅需要某些 vtable 的位置。而且，在 COOP 中，攻击者大多滥用现有代码的实际高级语义。因此，**除 ROP gadgets 之外的大多数 vfgadget 类型可能不受语义保留 - 重写二进制代码的影响**。只有用于加载 x64 参数寄存器的 LOAD-R64-G 才能通过这种方式被破坏。但是，在这种情况下，攻击者可能经常会回归到 x86 风格的基于 ML-ARG-G 的 COOP。

（2）C ++语义 – 感知防御技术

我们观察到 COOP 攻击中的控制流和数据流类似于良性 C ++代码（目标 G-2）。但是，C ++感知的防御者可以观察到某些偏差。我们现在讨论几个相应的防御。

a）验证 Vptrs

在基本的 COOP 中，伪造对象的 vptrs 指向现有的 vtable，但不一定指向它们的开头。当应用程序中的所有合法 vcall 站点和 vtable 都已知时，这允许实现针对 COOP 的可行防御，因此可以通过健全性检查来增强每个 vptr 访问。通过静态二进制代码重写可以在不访问源代码的情况下实现这种防御。**虽然这样的防御显着缩小了可用的 vfgadget 空间，但**我们的§V -A1 中的漏洞利用代码表明**仍然可以进行基于 COOP 的攻击**，至少对于大型 C ++目标应用程序而言。

防御者需要知道应用程序中每个 vcall 站点允许的 vtable 集合，以可靠地防止恶意 COOP 控制流（或者至少需要达到足以缩小 vfgadget 空间的近似值）。为此，防御者需要：（i）推断具有虚函数的 c++类的全局层次结构（ii）以确定对应每个 vcall 站点的 C ++类（层次结构内）。当源代码可用时，两者都可以轻松实现。没有源代码，只给出二进制代码和可能的调试符号

或 RTTI 元数据，前者可以以合理的精度实现，而据我们所知，通过静态分析对较大的应用程序来说，后者通常被认为是很难。

b）监控数据流

COOP 还展示了一系列可以在考虑 C++语义时显示出来的数据流模式。可能最重要的是，在基本 COOP 中，从同一个 vcall 站点调用具有不同数量参数的 vfgadgtes。当已知应用程序中每个虚函数所期望的参数数量时，可以检测到这种情况。虽然使用源代码是微不足道的，但从二进制代码中获取此信息可能具有挑战性。通过考虑实际的参数类型，可以创建更强大（但也可能更昂贵）的保护。

在 COOP 攻击中，伪造对象不是由合法的 C++构造函数创建和初始化的，而是由攻击者注入的。此外，重叠对象的概念会产生不寻常的数据流。为了检测这一点，防守者需要感知到 C++中应用程序的对象的生命周期。这需要了解具有虚函数的类（可能是内联的）构造函数和析构函数的行踪。

c）C++数据结构的细粒度随机化

在 COOP 中，每个伪造对象的布局需要与其 vfgadget 的语义字节兼容。因此，**在应用程序启动时随机化 C++对象布局**，例如，通过在 C++对象的字段之间插入随机大小的填充，可能妨碍 COOP。此外，vtable 的位置或结构的细粒度随机化可能是对 COOP 的可行防御。

我们得出结论，COOP 可以通过一系列不需要 C++语义知识的方法来缓解。但是，我们认为通过考虑并执行 C++语义可靠地防止 COOP 是至关重要的。通过静态二进制分析和重写这样做是很有挑战性的，因为 C++代码的编译在大多数情况下是一个有损的过程。例如，在二进制代码中，区分虚函数的调用与恰好存储在只读表中的 C 样式函数指针的调用可能是困难的。因此，之后明确地恢复基本的高级 C++语义可能很难甚至是不可能的。事实上，正如我们在§VII 中更详细地讨论的那样，我们知道没有二进制唯一的 CFI 解决方案，它可以精确地考虑 C++语义，以完全防止 COOP。

B．适用性和图灵完整性

我们已经证明 COOP 适用于不同操作系统和硬件架构上的流行 C++应用程序（目标 G-3）。当然，只有在至少有最小的一组 vfgadgets 可用的情况下才能安装 COOP 攻击。我们没有对 C++应用程序中可用 vfgadgets 的一般频率进行定量分析：以自动方式确定潜在 vfgadget 的实际有用性具有挑战性，我们将其留待将来工作。

给定表 I 中定义的 vfgadget 类型，COOP 具有与不受限制的 ROP 相同的表达能力。因此，它允许基于存储器加载/存储，计算和分支来实现图灵机（目标 G-4）。特别地，在§V 中的 COOP 实施案例表明在现实条件下，复杂语义例如循环可以实施。

第七部分：COOP 和现有防御

A．通用 CFI

我们首先讨论 CFI 方法，这些方法不考虑 C++语义来推导应该强制执行的 CFG。我们发现他们都很容易受到 COOP 的攻击。

Abadi 等人完成了最初的 CFI 工作的基本实现。使用二进制代码，使间接调用只能转到 address-taken 函数（粗粒度 CFI）。最近，该方案和一个密切相关的方案被证明易受高级的基于 ROP 的攻击。Abadi 等人还建议将它们的基本实现与影子调用栈相结合，以防止调用/返回不匹配。此扩展**有效地缓解了这些基于 ROP 的高级攻击**，而如第 VI 节所述，**它不会禁止 COOP**。

Davi 等人描述了嵌入式系统的硬件辅助 CFI 解决方案，它包含了一个影子调用栈和一组特定的运行时启发式算法。但是，间接调用策略仅验证间接调用是否以有效函数 start 为目标。由于 COOP 仅调用整个函数，因此它**可以绕过这种基于硬件的 CFI 机制**。

CCFIR 是 Windows x86 二进制文件的 CFI 方法，它使用随机排列的“springboard”来调度代码模块中的所有间接分支。在基线上，CCFIR 允许间接调用和跳转以定位二进制中的所有 adress-taken 位置，并把返回限制到某些调用前位置。CCFIR 的核心假设之一是攻击者无法“选择性地显示他们选择的 springboard stub addresses”。Gökta§等人最近表明，**springboard 的前期信息泄漏，使 CCFIR 的基于 ROP 的绕过是可能的**。相比之下，COOP 打破 CCFIR 而不违反其假设：**springboard 技术对 COOP 无效**，因为我们不注入代码指针而只注入 vptrs（指向代码指针的指针）。CCFIR 还确保只能通过常量间接分支来调用敏感的 WinAPI 函数（例如，CreateFile（）或 WinExec（））。但是，正如第 VI-A1a 段所述，这项措施并不能防止危险的攻击，也可能在实践中被回避。在任何情况下，COOP 都可以在攻击的第一阶段使用，以选择性地读出 springboard。

Microsoft Windows 10 Technical Preview 中的许多系统模块都是使用 Control Flow Guard（CFG）编译的，这是一种简单的 CFI 形式。**目前形式的微软 CFG 不会阻止 COOP**，但可能也不会阻止基于 ROP 的高级攻击。

Tice 等人最近为 GCC 和 LLVM 编译器套件描述了两种前向 CFI 变体，它们仅用于约束间接调用和跳转但不限制返回。因此，前向 CFI 不会以任何方式阻止 ROP。提出的变体之一是用于 GCC 的 C++-aware virtual table verification（VTV）技术。**它根据 C++类层次结构严格限制每个 vcall 站点的目标，从而防止 COOP**。自版本 4.9.0 起，VTV 可在 mainline GCC 中使用。但是，LLVM 的变体称为间接函数调用检查（IFCC）“[...]不依赖于 C ++或其他高级语言的细节”[52]。相反，每个间接调用站点都与一组有效的目标函数相关联。如果一个目标函数（i）是 address-taken 的并且（ii）其签名与呼叫站点兼容，则目标有效。Tice 等人讨论了 IFCC 功能签名兼容性的两个定义：（i）所有签名都是兼容的，或者（ii）具有相同数量的参数的签名是兼容的。我们观察到**前一种配置不会阻止 COOP，而后者在实践中允许基于 COOP 的强大攻击**，如第 VI-A2b 段所述。

B．C++-aware CFI

当从源代码中考虑精确的 C++语义时，可以可靠地防止 COOP 的控制流。因此，存在各种基于源代码的 CFI 解决方案，其防止 COOP，例如，如上所述的 GCC VTV，Safedispatch 或

WIT。

最近，已经提出了三种用于传统二进制代码的 C++-aware CFI 方法：T-VIP，vfGuard 和 VTint。他们遵循类似的基本方法：

（1）使用启发式和静态数据流分析来识别 vcall 站点和 vtable（仅限 vfGuard 和 VTint）。

（2）vcall 站点的工具来限制允许的 vtable 集。

T-VIP 确保在每个已检测的 vcall 站点上 vptr 指向只读内存。可选地，它还检查相应 vtable 中的随机条目是否指向只读存储器。同样，VTint 将所有已识别的 vtable 复制到一个新的只读部分，并检测每个 vcall 站点以检查 vptr 是否指向该部分。两者都有效地防止了基于伪 vtable 注入的攻击，但是在 COOP 攻击中只引用了实际的 vtable，**它们不会阻止 COOP**。VfGuard 仪器 vcall 站点检查 vptr 是否指向任何已知 vtable 的开头。如第 VI-A2a 段所述，这样的策略显着限制了可用的 vfgadgets 集，但**仍然不能可靠地防止 COOP**。V fGuard 还会在 vcall 站点检查调用约定的兼容性和 this-ptr 的一致性，但**这不会影响 COOP**。**尽管如此，我们认为 vfGuard 是针对 COOP 的最强二元防御之一**。VfGuard 严重限制了攻击者，我们希望它至少在某些攻击场景中是可靠的防御，例如，对于中小型 x86 应用程序。

C．基于启发式的检测

Microsoft EMET 可能是部署最广泛的漏洞利用缓解工具。其中，它实现了不同的基于启发式的 ROP 检测策略。此外，已经提出了几种相关的基于启发式的防御措施，它们利用现代 x86-64 CPU 中可用的某些调试功能。最近所有这些防御都被证明无法检测到更先进的基于 ROP 的攻击。同样，HDROP 防御使用性能监视计数器现代 x86-64 CPU 用于检测基于 ROP 的攻击。该方法依赖于观察到 CPU 的内部分支预测通常在执行公共代码重用攻击期间以异常方式失败。

正如在讨论 § VI-A ，这种启发式不太可能实际应用于 COOP。我们其实可以证实我们的 Internet Explorer 漏洞（第 VA 和第 VB）不被 EMET 版本 5 检测。

D．代码隐藏、打乱或重写

STIR 是一种仅二进制防御方法，可以在每次启动时随机重新排序应用程序中的基本块，以使攻击者不知道小工具的下落 - 即使她可以访问完全相同的二进制文件。正如 § VI -AIc 中所讨论的那样，**这样的方法在概念上不会影响我们的 COOP 攻击**，因为 COOP 仅将整个函数用作 vfgadgets，并且只需要知道 vtable 的下落。这也适用于最近提出的 O-CFI 方法，它将 STIR 概念与粗粒度 CFI 相结合。

Execute-no-Read（XnR）针对所谓的 JIT-ROP 攻击提出的防御措施，它可以防止代码页被读取。我们注意到，根据具体情况，**相应的 JIT-coop 攻击不能总是被这些措施所阻碍**，因为它足以从数据部分读取 vtable 和可能的 RTTI 元数据（包含类的文字名称）并应用模式匹配以识别感兴趣的 vtable 的地址。

G-Free 是 GCC 编译器的扩展。G-Free 生成 x86 本机代码（很大程度上）不包含未对齐的间接分支。此外，它的目的，以防止攻击者滥用对准间接分支：在栈上返回地址加密/解密的函数的入口/出口和一个“cookie”的机制来确保间接跳转/调用指令只能通过达成他们的相应功能的输入。虽然对许多先进的基于 ROP 的攻击有效，但 G-Free **不会影响 COOP**。

E．内存安全

为 C / C ++应用提供内存安全形式的系统**可以构成对一般控制流劫持攻击的强大防御**。由于我们的对手模型明确预见到最初的内存损坏和信息泄漏（参见§III-B），我们不会详细探讨这些系统的防御优势。相反，我们在下面示例性地讨论了两种最近的方法。

代码 - 指针完整性（CPI）：作为 C / C ++的低开销控制流劫持保护。在基线上，CPI 保证了代码指针的空间和时间完整性，并且递归地保证了代码指针的指针。与在 C ++应用程序中一样，通常存在许多指向代码指针的指针（例如每个对象的 vptr），CPI 仍然会在那里施加很大的开销。因此，又提出代码指针分离（CPS）：作为 CPI 的一种较便宜的变体，专门针对 C ++。在 CPS 中，敏感指针不会被递归保护，但仍然强制执行（i）代码指针只能通过代码指针存储指令存储到内存中或在内存中修改，并且（ii）代码指针只能通过代码指针加载指令加载。其中代码指针加载/存储指令在编译时是固定的。库兹涅佐夫等人认为 CPS 提供的保护在实践中可能足够，因为它在概念上阻止了最近基于 ROP 的先进攻击[。我们观察到 **CPS 不会阻止我们的 COOP 攻击**，因为 COOP 不需要注入或操纵代码指针。在存在 CPS 的情况下，尽管很难调用未由应用程序导入的库函数。但我们注意到几乎所有应用程序都导入关键功能。在存在 CPS 的情况下，通过 INV-G 调用库函数也可能是复杂的或不可能的。然而，这不是一个障碍，因为 CPS 不考虑 C ++语义，因此可以很容易地调用导入的库函数，而不需要绕过 INV-G，如方法 W-2 中的第 III-E 节所述。

第八部分：相关工作

证实了几种先进的基于 ROP 的攻击绕过了某些粗粒度 CFI 系统或基于启发式的系统。COOP 对细粒度重写，改组和隐藏可执行代码的容忍度是唯一的。

Sigreturn Oriented Programming（SROP），这是一种滥用 UNIX 信号的独特代码重用攻击方法。SROP 是图灵完整的，与 ROP 相反，它不会链接短的指令序列块。在 SROP 中，对位于堆栈上的攻击者提供的信号帧重复调用 UNIX 系统调用 sigreturn。因此，作为必要条件，攻击者需要控制堆栈，并且需要以能够转移该控制流，使得 sigreturn 被调用。SROP 并非专门设计用于规避现代保护技术，而是作为 ROP 的易用和便携替代品以及实施隐形后门。

Tran 等人证明图灵完整的 return-to-libc 攻击是可能的。从根本上说，他们的方法与我们的方法有相似之处。但是，它在概念上不能用于绕过现代 CFI 系统。

第九部分：结论

在本文中，我们介绍了伪造的面向对象编程（COOP），这是一种新颖的代码重用攻击技术，可绕过几乎所有 CFI 解决方案和许多其他不考虑面向对象 C ++的防御语义。我们讨论了面向对象编程的细节，并解释了 COOP 背后的技术细节。我们相信，我们的研究结果有助于持续研究设计实用且安全的防御控制流劫持攻击，这是一种已经存在了二十多年的严重威胁。我们需要考虑更高级别的编程语言特定语义的基本见解是未来防御的设计和实现的有价值的指南。特别是，我们的结果要求重新考虑仅依赖二进制代码的防御评估。