面向 C++商业软件二进制代码中的类信息恢复技术

杨 晋[1,2], 龚晓锐[1,2*], 吴 炜[1,2], 张伯伦[1,2]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 中国科学院大学 网络空间安全学院 北京 中国 100049

摘要 采用 C++编写的软件一直是二进制逆向分析中的高难度挑战, 二进制代码中不再保留 C++中的类及其继承信息, 尤其是正式发布的软件缺省开启编译优化, 导致残留的信息也被大幅削减, 使得商业软件(Commercial-Off-The-Shelf, COTS)的 C++二进制逆向分析尤其困难。当前已有的研究工作一是没有充分考虑编译优化, 导致编译优化后类及其继承关系的识别率很低, 难以识别虚继承等复杂的类间关系; 二是识别算法执行效率低, 无法满足大型软件的逆向分析。

本文围绕编译优化下的 C++二进制代码中类及其继承关系的识别技术开展研究, 在三个方面做出了改进。第一, 利用过程间静态污点分析从 C++二进制文件中提取对象的内存布局, 有效抵抗编译优化的影响(构造函数内联); 第二, 引入了四种启发式方法, 可从编译优化后的 C++二进制文件中恢复丢失的信息; 第三, 研发了一种自适应 CFG(控制流图)生成算法, 在极小损失的情况下大幅度提高分析的效率。在此基础上实现了一个原型系统 RECLASSIFY, 它可以从 C++二进制代码中有效识别多态类和类继承关系(包括虚继承)。

实验表明, 在 MSVC ABI 和 Itanium ABI 下, RECLASSIFY 均能在较短时间内从优化后二进制文件中识别出大多数多态类、恢复类关系。在由 15 个真实软件中的 C++二进制文件组成的数据集中(O2 编译优化), RECLASSIFY 在 MSVC ABI 下恢复多态类的平均召回率为 84.36%, 而之前最先进的解决方案 OOAnalyzer 恢复多态类的平均召回率仅为 33.76%。除此之外, 与 OOAnalyzer 相比, RECLASSIFY 的分析效率提高了三个数量级。

关键词 二进制分析; 类继承关系恢复; 静态污点分析; 自适应 CFG 生成算法
中图法分类号 TP311.5 DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.12.04

Class Information Recovery Technology for COTS C++ Binary

YANG Jin[1,2], GONG Xiaorui[1,2*], WU Wei[1,2], ZHANG Bolun[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract Software written in C++ has always been a difficult challenge in binary reverse analysis. Binary code no longer retains the classes and their information in C++, especially Commercial-Off-The-Shelf (COTS) enables compiler optimization by default, resulting in significant reduction of residual information. It makes COTS C++ binary reverse analysis particularly difficult. At present, the existing research work does not fully consider compilation optimization, resulting in a low recognition rate on recovering classes and class relationships under compiler optimization , and it is difficult to identify complex relationships such as virtual inheritance. Second, the recognition algorithm has low efficiency and cannot meet the reverse analysis of large-scale software.

This paper conducts research on the identification technology of classes and their inheritance in C++ binary under compiler optimization, and makes achievements in three aspects. First, using the inter-procedural static taint analysis to extract the object memory layout from the C++ binary, effectively resisting the impact of compiler optimization (inline constructors); second, introducing four heuristic methods, which can recover lost information in C++ binary files; third, an adaptive CFG (control flow graph) generation algorithm has been developed to greatly improve the efficiency with minimal loss. On this basis, a prototype system RECLASSIFY is implemented, which can effectively identify polymorphic classes and class relationships (including virtual inheritance) from C++ binary.

Experiments show that under both MSVC ABI and Itanium ABI, RECLASSIFY can identify most polymorphic class and recovery class relationships from the optimized binary in a short time. In a data set composed of 15 C++ binaries in real software (O2 compiler optimization), the average recall rate of RECLASSIFY recovering polymorphic classes under MSVC ABI is 84.36%, while the average recall rate of most advanced solution OOAnalyzer is only 33.76%. In addition,

通讯作者: 龚晓锐, 硕士, 正高级工程师, Email: gongxiaorui@iie.ac.cn。
本课题得到北京市科技计划网络空间攻防特殊技能人才培养及支撑平台建设课题(No. Z181100002718002)资助。
收稿日期: 2020-06-05; 修改日期: 2020-06-12; 定稿日期: 2022-12-07

compared with OOAnalyzer, the analysis efficiency of RECLASSIFY is improved by three orders of magnitude.

Key words binary analysis; class inheritance recovery; static taint analysis; adaptive CFG generation algorithm

1 引言

C++是最受欢迎的编程语言之一, 这是由于封装, 继承和多态的抽象特征以及执行效率的原因, 它提高了代码重用率, 可维护性和可伸缩性。因此, 开发人员通常会选择 C++来开发大型商业软件, 例如 Office[1], Adobe Reader[2], Windows [3], Chrome[4], Firefox[5], MySQL[6]等。

然而, 逆向 C++程序是一件很困难的事情。在二进制层次, 由于丢失了高级抽象信息, 许多分析人员很难理解程序逻辑。例如, 开发人员一旦将源代码编译为禁用了 RTTI(运行时类型信息)[7]的二进制文件, 所有冗余信息都将被去除。 此外, 在 C++的封装, 继承和多态的三个抽象特征中, 多态的应用最为复杂。

多态因为可以显著提高代码利用率, 所以它在面向对象软件设计中起着至关重要的作用。虚函数机制的实现为 C++提供了丰富的类层次结构, 使我们的代码更具可伸缩性。然而, 商业软件一般都经过编译优化, 这对于理解程序带来了巨大的挑战。虚函数表的实现使类之间的关系更加模糊, 因此难以通过简单的静态分析获得这些高级抽象信息。

尽管具有挑战性, 但恢复类信息对于商业软件的安全审计具有重要的意义。棱镜计划[8]的曝光表明目前大型的商业软件普遍缺乏安全审计, 因此我们日常生活中的大多数商业软件在二进制层次都是不受信任的。恢复类信息的一个重要应用是为安全分析人员提供高级抽象信息, 以便他们可以更有效地理解 C++二进制文件, 并进一步执行更复杂的漏洞分析, 例如在浏览器中识别类型混淆漏洞并评估其可利用性。然而, 许多复杂的 C++程序大量使用多态的特性, 这使安全分析人员很难理解二进制文件并执行各种与安全相关的任务, 并且非常耗时。

已有的研究工作[9-20]在恢复类信息方面取得了一些进展, 但是在恢复多态性方面仍然存在三个问题: 处理复杂编译优化的能力, 虚继承的识别和分析效率。这些未解决的问题使已有的研究工作不能处理实际的大型商业软件, 这是因为商业软件通常使用编译器优化和虚继承来减小代码大小, 提高执行效率并增加可伸缩性。除此之外, 低效的分析效率使得已有的研究工作不能在可接受时间内分析大型二进制文件。

1) 缺乏处理复杂编译优化的能力会导致恢复类的召回率较低。已有的研究工作无法处理编译优化, 或仅考虑简单的编译优化情况, 因此无法正确还原大量的类。例如, OOAnalyzer[17]在非编译优化下取得了成功, 但是在复杂编译优化下却表现不佳。根据我们的实验, 在 O2 编译优化下 OOAnalyzer 恢复类的召回率不到 40%, 这意味着安全分析人员仍然必须手动恢复一半以上的类信息。

2) 现有解决方案无法从经过编译优化的二进制文件中恢复虚继承。分析对象内存上的读写操作是恢复类继承关系的关键。然而, 当处理虚继承时, 对对象内存的读写操作分布在不同的函数中。 已有的研究工作仅对对象的内存布局执行过程内分析, 因此无法处理虚继承。例如 OOAnalyzer[17]、Marx[16]等。VirtAnalyzer[9]在恢复虚继承方面取得了一些进展,它通过提取 *VTT*(*Virtual Table Table*)并跟踪有关基类构造函数的“*call*”指令和参数来恢复虚继承。然而, 当构造函数内联到其他函数中时, 则不存在与虚基类相关的构造函数的“*call*”指令, 因此 VirtAnalyzer 无法恢复虚继承。实际上, 由于虚继承已在大型商业软件中广泛使用, 因此缺乏处理虚继承的能力会降低恢复类和类关系的能力。

3) 已有的研究工作分析效率较低。例如, OOAnalyzer 可能花费超过 102 个小时来分析 libz3.dll(这是一个广泛使用的定理证明程序)。此外, 即使对于小型程序, 例如 opencv_dnn.dll(5024 KB), OOAnalyzer 仍需要 20 多个小时。对于辅助分析工具来说, 如此巨大的时间开销是无法接受的。

针对上述挑战, 本文提出了启发式推理的方法, 研发了过程间静态污点分析、自适应 CFG 生成算法等技术, 能够有效解决从经过编译优化的商业软件的二进制代码中恢复类信息的问题。实验结果表明, 在 O2 编译优化情况下, RECLASSIFY 的类的平均召回率 (84.36%) 远高于 OOAnalyzer (33.76%), 且 RECLASSIFY 的 F score(0.90)也是 OOAnalyzer 的两倍。除此之外, RECLASSIFY 的分析效率相比当前最好的工具 OOAnalyzer 提升了三个数量级。

本文的贡献总结为以下三点:

- 本文定义了四个与编译优化无关的启发式规则, 可以有效的从经过编译优化的二进制文件中恢复类和类关系(包括虚继承)。
- 本文提出了一种自适应 CFG 生成算法, 在极小

损失的情况下大幅度提高了分析的效率。

- 本文实现了一个原型系统——RECLASSIFY, 能够从大型的 C++二进制文件中恢复类和类关系, 为分析大型二进制文件提供了可能。

2 背景与挑战

本章主要介绍 C++二进制类继承关系识别的相关基础知识。由于类内存布局的复杂性, 具体实现交给了编译器负责, 目前主要遵循两套标准, 一套是以 GCC/Clang 为首的 Itanium ABI[21]标准, 另一套是以 MSVC 为首的 MSVC ABI[22]标准。在现实世界中, 大型的商业软件大多是在 Windows 平台上开发, 而 Windows 平台上大多是使用 MSVC 开发、编译的, 所以后面的内容主要以 MSVC ABI 为主。

2.1 重要的基本概念

多态类与虚函数表: 多态是 C++中最为复杂的一个特性, 尽管它极大地提高了代码复用率和代码可扩展性, 但是也为逆向工程带来了极大的挑战。多态类从二进制角度来讲, 就是具有虚函数的类, 虚函数指具有 virtual 修饰符的函数。具有虚函数的多态类会在实例化的对象内存中会存在虚函数表指针(*vftptr*)的变量, 这个虚函数表指针会指向虚函数表(*vftable*), 虚函数表里记录着这个类的所有虚函数的地址。因为虚函数表的存在, 多态类调用函数的方式由直接调用改为间接调用, 进而增加了逆向的难度。

二进制代码中常见的类关系: 二进制代码中常见的类关系有单一继承、多重继承、虚继承和对象成员。单一继承, 指一个派生类只继承了一个基类。多重继承, 指一个派生类继承了多个基类。虚继承, 是一种特殊的继承关系, 它为了防止派生类数据重复。若没有虚继承, 同样的数据可能在派生类中存在多份, 这样在数据引用的时候便会引发错误。虚继承的存在消除了冗余的数据, 保证了数据的一致性。对象成员, 指一个类的对象是另一个类的成员变量, 这两个类之间的关系也叫做联系。

构造函数与析构函数: 构造函数是一个类的对象实例化时进行的初始化函数, 一个构造函数一般会执行以下动作:

1) 若存在虚继承关系, 初始化对象的虚基类表指针(*vbtptr*)。

2) 调用基类的构造函数。

3) 初始化对象的虚函数表指针(*vftptr*)。

4) 初始化对象的内部成员变量, 如普通变量, 对象成员等等。

5) 调用其他的初始化函数。

在编译优化的情况下, 构造函数可能会被内联进其他函数中, 后面会进行详细的分析。

析构函数是一个类的对象在销毁时执行的函数, 与构造函数类似, 它执行的顺序大体是相反的, 执行的动作如下:

1) 将派生类的虚函数表指针(*vftptr*)写入对应的对象内存区域。

2) 销毁对象的内部成员变量, 如普通变量, 对象成员等等。

3) 执行基类的析构函数。

与构造函数不同的地方是, 析构函数一般不会被函数内联, 这是因为为了防止内存泄露, 析构函数一般被设置为虚函数, 而虚函数是不会被函数内联的。

虚基类表: 在 MSVC 编译器下, 具有虚继承关系的类的对象内存中还包含虚基类表(*vbtable*)这个变量, 虚基类表的主要作用是用来寻找当前 this 指针与虚基类之间的相对偏移。在执行构造函数时, 虚基类表(*vbtable*)的初始化要先于虚函数表(*vftable*)的初始化。

RTTI: RTTI[7]指运行时类型信息, 是编译器为了支持 C++中 dynamic_cast 和 typeid 以及异常处理而产生的额外字段, 当有多态类存在, 即存在虚函数表时, 这个类对应的 RTTI 也会出现。RTTI 里记录了类的具体相关高层抽象信息, 比如类名, 类关系, 内部成员变量信息等等。已有的研究工作[20]中就有使用 RTTI 信息来恢复类的层次结构的方法, 但是商业软件一般会把 RTTI 信息去掉, 这就使得该方法失效。

this 指针传递准则: this 指针记录了对象的内存基址, 在类方法中, this 指针的传递是隐式的。在 MSVC 编译器下, this 指针一般通过 rcx 寄存器(64 位程序)或者 ecx 寄存器(32 位程序)传递。

2.2 类对象实例内存布局过程

结合前面的基础知识, 下面给出一个具体的例子来进行讲类对象实例内存布局过程。

如图 1 所示, 是一个简单的 C++ 程序的源码, 一共有 5 个类 A、B、C、D、E, 其中 D 多重继承了 B 和 C, B 和 C 同时虚继承了 A, 并且 E 是 D 的对象成员。在无函数内联的情况下, 其汇编代码如图 2 所示。

如图 3 所示, 该图展示了派生类 D 的对象内存布局与对象初始化的过程。对象初始化过程如下:

(1) 在执行 D 的构造函数时, 因为有虚继承的存在, 首先会初始化 D 的两个 *vbtable* 的指针, 将其写

```
Class A{
  public:
    int a;
    A(){cout<<"A"<<endl;}
    virtual ~A(){cout<<"~A"<<endl;}
    virtual void funcA(){...}
};
Class B: virtual public A{
  public:
    int b;
    B(){cout<<"B"<<endl;}
    virtual ~B(){cout<<"~B"<<endl;}
    virtual void funcB(){...}
};
Class C: virtual public A{
  public:
    int c;
    C(){cout<<"C"<<endl;}
    virtual ~C(){cout<<"~C"<<endl;}
    virtual void funcC(){...}
};
Class E{
  public:
    int e;
    E(){cout<<"E"<<endl;}
    virtual ~E(){cout<<"~E"<<endl;}
    virtual void funcE(){...}
};
Class D: public B, public C{
  public:
    int d;
    E object_member;
    D(){cout<<"D"<<endl;}
    virtual ~D(){cout<<"~D"<<endl;}
    virtual void funcD(){...}
};
void main(){
  D *d= new D();
  d->funcD();
  delete d;
  ...
}
```

图 1　五个类的 C++源码

Figure 1　The C++ source code of five classes

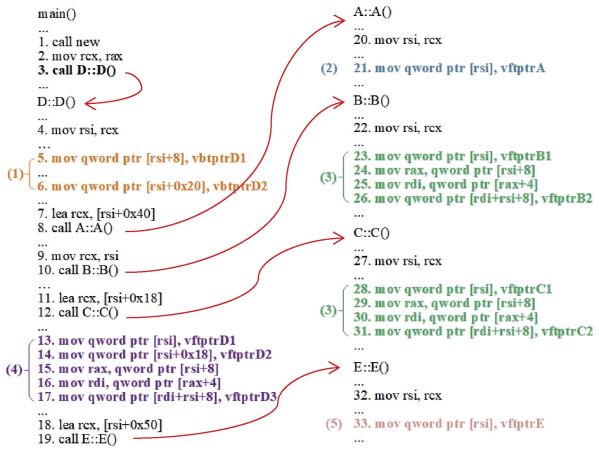

图 2　类 D 初始化过程的汇编代码

Figure 2　Assembly code for class D initialization

入对应的内存偏移处, 如图 2 和图 3 中的(1)所示。这里面 *vbtable* 就是用来记录当前 this 指针与虚基类的偏移。

(2) 执行虚基类 A 的构造函数, 对 A 的 *vftable* 指针进行初始化, 如图 2 和图 3 中的(2)所示, 将其写入对应内存偏移 0x40 处。

(3) 执行基类 B 和 C 的构造函数, 将 B 和 C 的 *vftable* 指针初始化。如图 2 和图 3 中的(3)所示, 此时 0x40 偏移处又被写入了两次 *vftable* 指针, 这种 *vftable* 指针覆盖的写入操作叫做覆写操作。

(4) 对派生类 D 的 *vftable* 指针初始化, 如图 2 和图 3 中的(4)所示, 覆盖了其基类(0x0, 0x18)和虚基类(0x40)的 *vftable* 指针。

(5) 完成对象成员 E 的 *vftable* 指针的初始化, 如图 2 和图 3 中的(5)所示。

2.3　挑战

C1. 虚继承的识别: 已有的研究工作在识别类关系上主要分为两大类方法, 一类方法是通过构造函数调用关系来推理类之间的继承关系(基于控制流的方法), 另一类方法是通过覆写分析和偏移量记录的方法来识别类关系(基于数据流的方法), 但是这两类方法都没有对虚继承有深入的研究, 所以都无法正确识别虚继承, 下面详细解释一下原因:

- 通过构造函数调用关系来推理类之间的继承关系在编译优化的情况下因为构造函数内联而导致方法失效; 同时, 即便是在非编译优化的情况下, 该方法也无法正确识别虚继承。对于图 1 的示例来说, 已有的研究工作由调用图推理得到的类关系要么是 D 多重继承了 A、B、C, 要么是程序崩溃(没有处理与虚继承相关的指令), 这两种情况都不能正确的识别虚继承关系。
- 通过覆写分析和偏移量记录的方法来识别类关系在识别虚继承上有两个问题, 一个问题是仅使用偏移量会导致错误识别虚继承, 如图 3 所示, D 有三张 *vftable* 表, 对应的内存偏移分别是 0、0x18、0x40, 按照已有的研究工作, 他们会将 0 偏移处的识别为主基类, 其他的与派生类 D 覆写相关的类都是辅助基类, 虽然已有的研究工作已经能够处理对象成员, 但是因为没有对虚基类和虚继承做特殊处理, 最后识别的结果也是 D 多重继承了 A、B、C 或者程序崩溃。

VirtAnalyzer[9]在恢复虚继承上取得了一定的进步, 它通过 VTT 映射与控制流追踪来恢复虚继承。然而, VirtAnalyzer 没有考虑构造函数内联的情况(挑战 C3), 在构造函数内联时, 虚基类相关的控制流信息是无法获取到的, 进而导致该方法失效。

除此之外, 已有的研究工作对于 *vftable* 的分析都是过程内的, 它无法处理数据间接引用的问题。如图 3 所示, 在 B, C 将 *vftable* 的指针写入虚基类的内存偏移处时, 程序首先要先从内存中读取 *vbtable*, 找到对应的虚基类偏移, 并计算出真正的内存偏移, 然后才能进行覆写操作, 而过程内的方法是无法读取 *vbtable* 里的数据的。

C2. 地址优化: 当一个类拥有多重继承关系或者虚继承关系时, 该类拥有多张 *vftable*, 所以在恢复多态类的时候需要进行 *vftable* 的合并操作, 即将多张 *vftable* 表识别为一个类。已有的研究工作[9,11,16]假设 *vftable* 分布在连续的地址空间上, 在 GCC 下直接依据 *OffsetToTop* 字段进行同一个类的 *vftable* 的合并, 并且一个 *vftable* 只与一个类绑定, 然而这在复杂的编译优化的环境下并不适用。

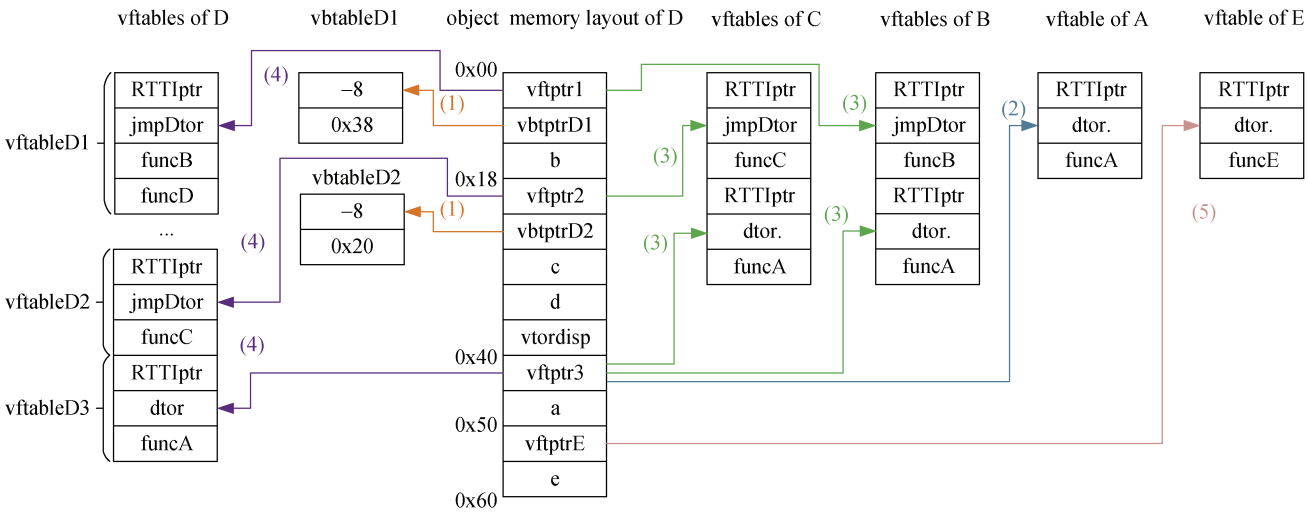

图 3　派生类 D 对象内存布局与对象初始化过程

Figure 3　The object memory layout and object initialization of derived class D

首先，在复杂的编译优化的环境下，同一个类的不同 *vftable* 可能分布在不连续的地址空间上，这样已有的研究工作中根据连续地址合并 *vftable* 的方法就都失效了。其次，在复杂的编译优化的环境下，不同的类可能共享相同的 *vftable*，使用已有的研究工作会导致类的缺失。所以，要想解决地址优化的问题，需要一种更加通用的方法。

C3. 构造函数完全内联：已有的研究工作[15-17]仅考虑内联函数的简化形式。例如，基类的构造函数被内联到派生类的构造函数中。但是，在复杂的编译优化的情况下，派生类和基类的构造函数都内联到程序中的另一个函数中。例如，如图 2 所示，如果 A()，B()，C()，D()，E() 都内联到函数 main() 中，则 OOAnalyzer[17]无法识别任何构造函数，因为它认为在 new()之后调用的第一个函数是构造函数。

C4. 分析效率：因为已有的研究工作的解决方案会频繁对所有函数进行检查和操作，尤其是在 CFG 生成阶段，因此他们都无法有效地分析大型的二进制文件，比如 OOAnalyzer[17]。在从二进制文件中恢复类和类关系领域中，为所有函数生成 CFG 会造成很多的无效时间开销，这是因为涉及类和类关系的函数并不是很多，不需要分析所有的函数。除此之外，符号执行的路径爆炸问题也对分析效率造成了很大的影响，进而导致已有的研究工作[14-15,17]分析效率低下。

3　解决方案设计思想

本章主要介绍解决方案的设计思想。首先，本章介绍本文的设计目标与环境假设，然后对每一个方法或者技术的产生原因和设计思路进行详细的分析。

3.1　设计目标与假设

本文的解决方案满足以下的设计目标:

- 该解决方案必须能够分析商业软件，即能够抵抗编译优化的干扰，其中包括函数内联问题。
- 该解决方案必须能够从 C++二进制文件中恢复类和类关系，其中类关系包括单一继承、多重继承、虚继承和对象成员。
- 该解决方案必须能够分析大型的二进制文件，所以它需要具有较高的分析效率。

本文的解决方案基于以下的假设:

- 该解决方案假设析构函数是虚函数，如章节 2 所述，为了防止内存泄露析构函数一般被设置成虚函数，所以该假设是合理的。
- 该解决方案假设被分析的二进制文件没有经过混淆，如代码虚拟化混淆等技术。

3.2　整体架构

如图 4 所示，是 RECLASSIFY 的整体架构，这里输入是一个二进制文件，经过信息提取、析构函数与构造函数分析、过程间静态污点分析和启发式推理四大步骤。输出的类信息有类、类关系和类方法等信息，其中类关系包括单一继承、多重继承、虚继承和对象成员。RECLASSIFY 分析的四大步骤如下所示:

1) 从经过编译优化的二进制文件中提取 *vftable*、*vbtable* 和符号表等信息。

2) 寻找每个 *vftable* 对应的析构函数，并通过 *vftable* 和析构函数来进行启发式搜索构造函数。

3) 将构造函数和析构函数的地址作为输入，运

行自适应 CFG 生成算法，生成相关的 CFG，然后对构造函数或析构函数执行静态污点分析，构造出对象内存布局。

4) 利用对象内存布局和四个与编译优化无关的启发式来恢复类信息。

3.3 高层设计思想

使用析构函数合并 *vftable*: 使用析构函数合并 *vftable* 可以有效抵抗地址优化带来的影响。如章节 2 所述，因为为了避免内存泄露的问题，程序员都会将析构函数设置为虚函数，这样可以通过寻找 *vftable* 中的析构函数是否相同来进行合并。

然而，在真实的二进制文件中，同一个类的不同 *vftable* 只有一个记录着析构函数，其他的 *vftable* 里不存在析构函数。但是，我们通过仔细分析和观察可以发现，如图 5 所示，不存在析构函数的 *vftable* 中都存在一种跳转析构函数的包装函数(jmpDtor)，该函数主要的作用是修改 this 指针，然后再跳转到真正的析构函数。因此，我们通过探测语义来寻找语义上相同的析构函数作为启发式，以此来合并同一个类的不同 *vftable*，这样便可解决挑战 C2。

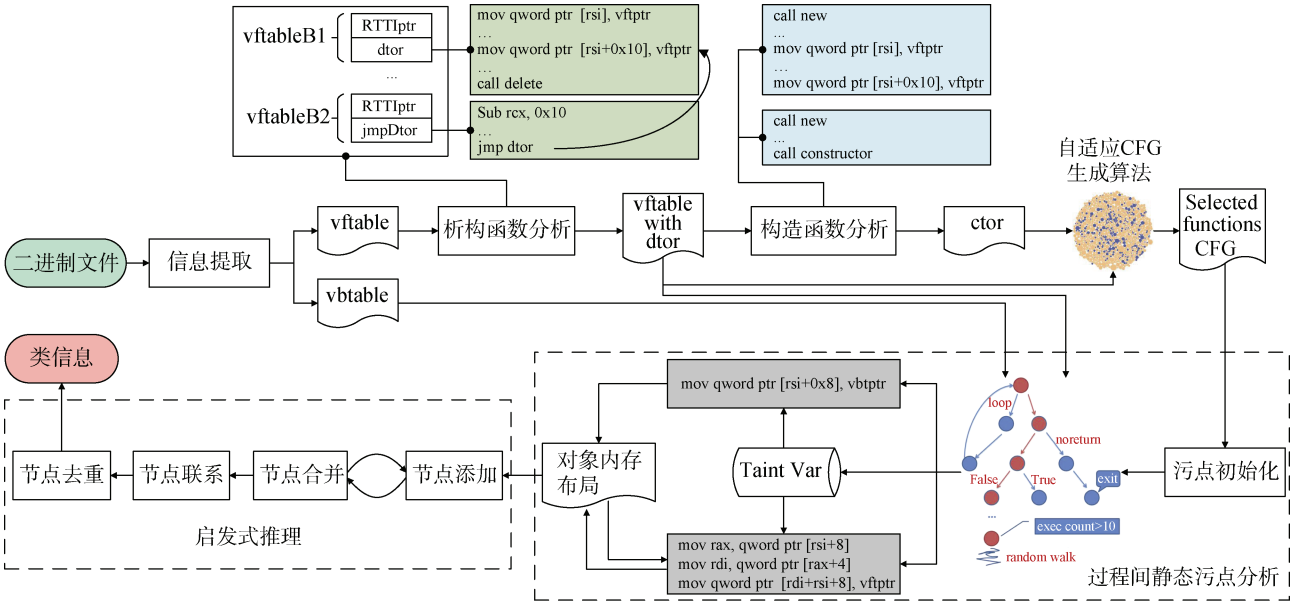

图 4 RECLASSIFY 架构图

Figure 4 RECLASSIFY architecture diagram

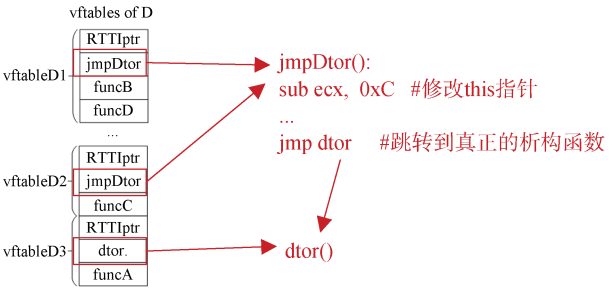

图 5 跳转析构函数

Figure 5 jmpDtor

使用过程间的静态污点分析构建对象内存布局: 对于挑战 C3，我们不必区分特定的构造函数，而是将所有函数考虑在一起，对整个函数集合进行分析，使用过程间的方法来构造对象内存布局，从数据流分析角度恢复类和类关系，这样可以抵抗函数内联的干扰。除此之外，过程间的方法可以处理虚继承的识别，如章节 2 所述，过程内的方法是无法处理数据间接引用的问题。

已有的研究工作中有一些方法使用了符号执行[14-15,17]来追踪数据流，但是符号执行的路径爆炸问题在面对分析大型二进制文件时尤为突出，会导致分析时间大幅度增加。由于类继承关系的特征性，只需要专注与 *vftable* 和 *vbtable* 相关的操作即可，不需要追踪所有的变量。除此之外，不同时间对同一内存偏移写入 *vftable* 会对类继承关系产生不同的影响，所以也要考虑时间顺序问题。基于以上原因，本文考虑使用污点分析技术，并在类继承关系识别上首次提出静态污点分析技术的思想，既能避免构造函数内联造成的影响，又能避免路径爆炸，有效地提高了分析的效率。

常规的二进制污点分析技术大多是动态的，而动态分析方法具有代码覆盖率低的缺点。使用动态污点分析技术恢复类和类关系，其识别类的数量取决于测试用例可以触发的类方法的数量，例如

Srinivasah 等人的工作[19]。解决此问题的方法是构建多个测试样本或从多个起点运行，但是这两种方法都会带来新的挑战。构造多个测试样本需要解决如何自动构造可以触发所有类方法的多个测试用例的问题；从多个起点运行需要解决如何确定每个起点的上下文的问题。这些问题在当前的研究进展中还没有得到很好的解决，因此动态分析方法在这里并不适用。最新的二进制静态污点分析技术可以避免上述问题，并在专业领域中取得良好的效果。例如，Bintaint[23]使用静态污点分析技术来缓解符号执行的路径爆炸问题，从而可以更有效地进行漏洞挖掘。同样的，在二进制类继承关系识别的分析中，静态污点分析技术也非常有帮助。它可以从每个起始分析函数开始，而无需考虑其上下文。在分析过程中通过近似程序的执行流程来选择执行的分支，并且焦点集中在 *vftable* 和 *vbtable* 的相关的内存读写操作上，以构造细粒度的对象内存布局。

静态污点分析技术可以有效地避免构造函数内联对分析的影响，因为该方法不需要区分特定的构造函数，只需要找到具有构造函数行为的起点，就可以建立对象的内存布局。除此之外，已有的研究工作仅描述了基类构造函数与派生类构造函数之间的继承现象，或者 *vftable* 覆写操作具有继承关系的浅层指令级现象。本文从内存布局的角度，更深入地揭示了继承的本质：对于一个对象，在相同的内存偏移处多次写入多个 *vftable*，这些 *vftable* 所属的类之间存在继承关系。

对象内存布局是把每个对象的 *vftable* 与 *vbtable* 的空间位置与时间顺序都记录了下来，即将图 3 中的信息转换成如图 6 里的形式。纵向是空间维度，记录内存的偏移和内容的属性；横向是时间维度，记录着 *vftable* 或者 *vbtable*,以及它们的覆写顺序。使用对象内存布局来进行启发式推理，进而恢复类信息，这样可以有效地抵抗编译优化带来的影响。

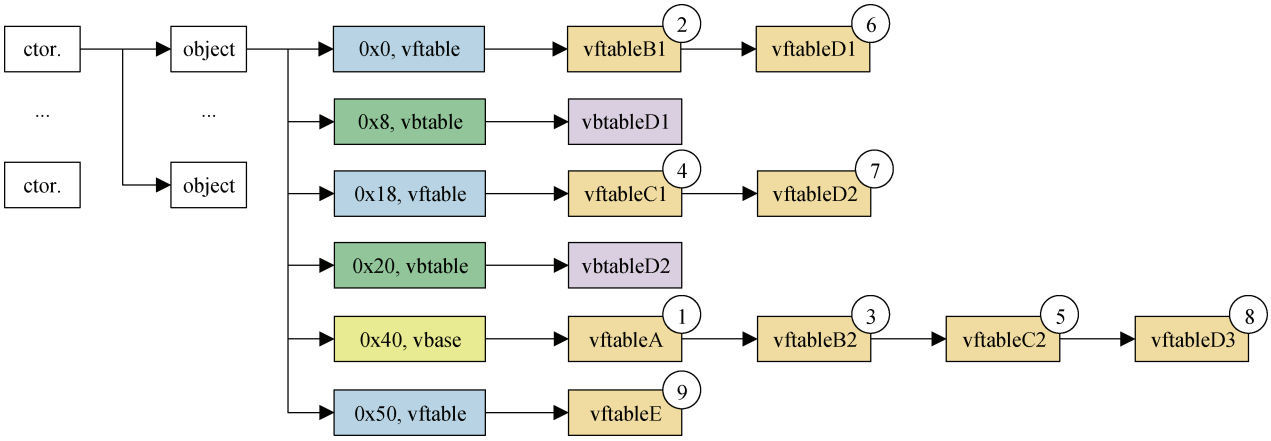

图 6　对象内存布局

Figure 6　Object memory layout

使用启发式推理来恢复类信息：启发式推理使用静态污点分析产生的对象内存布局来进行推断，在推断过程中使用了四个与编译优化无关的启发式，下面逐一进行讲解：

- 本文没有采用构造函数与 *vftable* 配对的方式来进行类继承关系的恢复，如挑战 C1 所述，这种方法在编译优化的情况下是无效的。我们通过静态污点分析技术记录了每个对象关于 *vftable* 和 *vbtable* 的内存布局，由 C++ ABI 原理可知，相同内存偏移处的 *vftable* 所属的类之间具有继承关系。
- 如挑战 C2 所述，我们通过逆向分析发现了同一个类的多张 *vftable* 具有语义相同的析构函数，如一个 *vftable* 具有析构函数，其他的 *vftable* 具有跳转析构函数(跳转析构函数指仅改变一下 this 指针，然后跳转到析构函数)。通过这个启发式可以将多个 *vftable* 合并为一个类。
- 系统在识别虚基类的时候，会存在数据间接引用的操作，如章节 C1 所述，已有的研究工作无法处理这个问题，使用过程间的静态污点分析技术可以准确处理该问题。过程间的静态污点分析能计算出正确的内存偏移，将 *vftable* 和对应的覆写顺序记录下来，同时会在该内存偏移处标上 *vbase* 标签。该内存偏移处的所有 *vftable* 所属的类具有虚继承关系，通过逆向经验可知，第一次写入该内存偏移处的 *vftable* 所属的类是虚基类。
- 对象成员的识别是通过覆写顺序来判别的，覆

写顺序指在构造函数中具有*vftable*覆写操作的时间顺序。如章节 2 所述, 在构造函数执行流程中, 对象成员的初始化是在完成该类的*vftptr* 初始化之后进行的, 所以对象成员的覆写顺序是大于对象成员所属类的覆写顺序的, 这与已有的研究工作[11,14]很近似。除此之外, 因为 RECLASSIFY 识别的时候是遍历以*vftable* 为集合的节点, 而 *vftable* 没有与构造函数配对, 所以我们在这里使用了析构函数, 通过检测一个 *vftable* 是否在某个析构函数内, 并结合覆写顺序共同判断, 以此来恢复对象成员, 这与已有的研究工作不同。

提高分析效率的方法: 已有的研究工作因为分析效率低下而不能分析大型的二进制文件, 这里主要介绍提高分析效率的三种方法:

- *启发式搜索构造函数*。已有的研究工作[10-14,17,20]通过遍历所有函数来寻找构造函数, 这在分析大型的二进制文件时是非常耗时的, 为了提高分析效率, 本文使用了启发式搜索构造函数的方法, 即通过遍历每一个 *vftable* 的写入操作的交叉引用来寻找构造函数。因为具有 *vftable* 覆写操作的函数只有构造函数和析构函数, 而析构函数提前被识别出来了, 所以剩下的函数就是构造函数。使用该方法在分析大型的二进制程序时可以大幅度提升分析的效率, 虽然可能会产生一些误报, 但是通过实验证明造成的影响不大。
- *自适应 CFG 生成算法*。 在进行静态污点分析之前必须先生成 CFG, 之前所有的工作都是对整个二进制文件生成 CFG, 而这在分析大型的二进制文件时是非常耗时的。除此之外, CFG 生成所消耗的时间在整个分析过程中占有很大比例, 然而, 在面对特定需求时, 往往并不需要分析整个二进制文件或者所有函数。比如在分析类的继承关系时, 仅需要关注构造函数和析构函数及其相关的函数(相关函数为构造函数或析构函数中直接或间接调用的函数), 而这些函数一般只占总数的 20%左右(数据来源可见章节 5), 所以对整个二进制文件生成 CFG 便会造成很多分析时间上的浪费。基于以上原因, 本文提出了一种新的生成 CFG 的思路, 那便是仅对关心的函数(构造函数和析构函数及其相关的函数)进行 CFG 生成, 即将构造函数和析构函数的地址作为输入进行 CFG 的生成, 这样可以大幅度地提高分析的效率。
- *静态污点分析*。如前面所述, 已有的研究工作使用符号执行来恢复类和类关系, 然而这会造成很大的时间开销。使用静态污点分析便可避免路径爆炸的问题, 同时也能避免动态分析产生的代码覆盖率低的问题。

4 RECLASSIFY 的实现

本章主要介绍 RECLASSIFY 的具体实现细节, 详细阐述了执行流程中的每一步内容, 并给出了扩展 Itanium ABI 的方法。除此之外, 本章针对一些实现过程中遇到的问题进行详细的分析和阐述, 并给出解决的方法。

4.1 信息提取

首先, 我们使用二进制逆向分析软件 IDA Pro[24], 编写 IDA Python[25]脚本对二进制文件进行信息提取, 提取的信息有 *vftable* 表、*vbtable* 表和符号表等信息。其中提取 *vftbale* 可以依据以下特征:

- 位置在只读段上, 如.rdata。
- 存储的地址在代码段上, 如.text。
- 第一项有交叉引用, 将 *vftable* 赋值给 *vftptr*。
- 若有 RTTI,则 *RTTIptr* 在 *vftable* 前面。
- 提取 *vbtable* 可以依据以下特征:
- 位置在只读段上, 如.rdata。
- 固定两个字段, 且字段长固定为 4 字节, 第一个字段为与对应的 *vftable* 的相对偏移, 因为 *vbtable* 与对应的 *vftable* 在对象内存中是紧挨着的, 所以偏移固定为–4(32 位程序)或–8(64 位程序), 第二个字段为虚基类偏移。
- 第一项有交叉引用, 将 *vbtable* 赋值给 *vbtptr*。

4.2 析构函数与构造函数分析

得到 *vftable* 之后, 我们先对析构函数进行分析, 遍历每个 *vftable* 中的函数, 分析语义找出其中的析构函数, 并将析构函数与 *vftable* 配对。这里的语义主要是以下两个:

- 覆写操作, 例如 *mov qword [rsi + 0x8], vftptr*。
- delete 操作, 例如 *call delete()*。

之后, 我们利用与析构函数配对后的 *vftable* 进行启发式搜索构造函数分析, 对每个 *vftable* 进行交叉引用查询, 排除其中的析构函数, 剩下的函数进行语义查询来寻找构造函数。这种方法可避免遍历所有函数, 提高分析效率。这里的语义主要是以下来两个:

- 覆写操作, 例如 *mov qword [rsi + 0x8], vftptr*。
- new 操作, 例如 *call new()*。

除此之外, 为了方便后续的静态污点分析(this

指针的污点初始化), 我们在这里需要将识别的构造函数分为两类, 一类是构造函数完全内联的情况, 另一类是其他情况。

4.3 过程间静态污点分析

自适应 CFG 生成算法: 在进行过程间静态污点分析之前, 我们需要先对要分析的函数生成 CFG。这里并没有对二进制文件的所有函数生成 CFG, 而是只针对构造函数、析构函数及其相关的函数生成 CFG, 这些函数一般只占总数的 20%左右(数据来源可见章节 5), 可以极大的提高分析的效率。自适应 CFG 生成算法具体包括以下步骤:

1) 将构造函数和析构函数的地址放在同一集合中, 遍历该集合, 并对每个函数以跳转语句进行基本块的划分, 每个函数以函数起始地址所在的基本块为起始点。

2) 对于每个基本块, 若是直接跳转语句或者条件跳转语句, 则将跳转目标所在的基本块与该基本块相连接。

3) 对于每个基本块, 若存在 call 指令, 通过符号表模式匹配的方式检验是否为系统调用, 若是则直接丢弃, 否则将 call 指令指向的函数地址所在的基本块与该基本块相连接, 若 call 指令指向的函数地址不存在集合中, 则将其添加进集合中。

4) 在处理跳转语句时, 若跳转目标地址在已经分析过的基本块中, 且不是该基本块的起始地址和结束地址, 则将该基本块对于跳转目标地址进行分割。

5) 若遇到循环结构时, 将走向循环路径的分支标记为 *loop*, 为智能路径选择策略提供判断依据。

6) 若遇到无返回的路径结构(*noreturn*)时, 将走向 *noreturn* 路径的分支标记为 *noreturn*, 为智能路径选择策略提供判断依据。

智能化路径选择策略: 为了避免路径爆炸, 提高效率, 我们不会遍历所有的路径, 会对执行的路径进行一些筛选。如图 7 所示, 有如下三种路径选择策略:

- 在 CFG 生成阶段识别循环和 *noreturn* 分支做好标记, 在遇到分支选择时会避免走这两种分支。
- 若两条分支既不是循环分支也不是 *noreturn* 分支, 依据编译器原理和逆向经验, 走 False 分支可覆盖覆写操作相关指令。
- 为避免陷入过程间循环等复杂情况的无限循环中, 设置指令追踪, 同一条分支指令执行超过 10 次后开始采用随机游走的策略。

污点初始化与传播规则: 以每个识别的构造函数/析构函数为起始点, 结束点为函数结束, 初始污点为 this 指针。若是构造函数完全内联的情况则 this 指针为 rax 寄存器, 起始指令为 new 函数的下一条指令, 其他情况 this 指针为 rcx 寄存器, 起始指令为函数第一条指令。

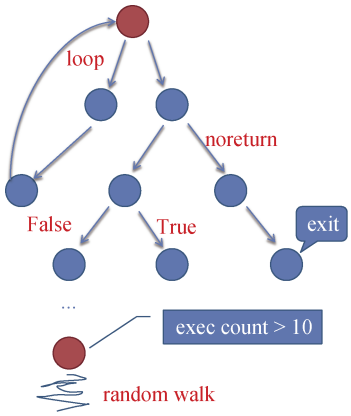

图 7 智能化路径选择策略

Figure 7 Intelligent path selection strategy

表 1 展示了静态污点分析的污点传播规则与检测点。当执行到 *vbtable* 写操作相关指令时(如 *mov qword [rsi + 0x8], vbtptr*), 我们通过污点找到对应的内存偏移, 将 *vbtable* 记录下来。当执行到 *vftable* 写操作相关指令时(如 *mov qword [rsi + 0x8], vftptr*), 我们通过污点找到对应的内存偏移, 将 *vftable* 记录下来, 同时将覆写顺序(构造函数中的覆写顺序)也记录下来。其中若涉及 *vbtable* 读操作相关指令时, 我们会通过污点从对象内存布局中寻找对应的 *vbtable*, 并计算出真正的内存偏移, 将 *vftable* 及其覆写顺序记录下来, 并将该内存偏移标记为虚基类(*vbase*), 其中第一个写入该内存偏移处的 *vftable* 所属的类即为虚基类。完成对象的静态污点分析之后, 我们便构建出了对象内存布局, 其整个过程具体包括以下步骤:

1) 对构造函数和析构函数及其相关的函数生成 CFG。

2) 对每个构造函数/析构函数执行污点分析, 采用智能化路径选择策略选择执行路径。

3) 将 this 指针标记为初始污点, 利用污点传播规则进行关于 *vftable* 和 *vbtable* 的污点的传播和消除, 执行完后构造出对象内存布局。

4.4 启发式推理

在得到对象内存布局之后, 我们对每一个对象的对象内存布局进行启发式推理分析, 循环进行节点添加和节点合并的过程, 然后对孤立的节点或节点树进行对象成员的分析。这里我们主要应用了 4 个与编译优化无关的启发式:

表 1　污点传播规则与检测点

Table 1　Taint propagation and sink check

	传播规则		检测点		
操作指令	源操作数为污点数据	源操作数为清白数据且目的操作数为污点数据	源操作数为污点数据且为对象内存	源操作数为 vftptr 且目的操作数为污点数据	源操作数为 vbtptr 且目的操作数为污点数据
移数指令	目的操作数标记为污点数据	目的操作数标记为清白数据	目的操作数标记为污点数据, 且内容为从对象内存布局中取出来的 vbtable 的地址	将 vftable 记录进对象内存布局中,若涉及 vbtable 读取则对应偏移内容标记为“vbase”	将 vbtable 记录进对象内存布局中
算术指令	目的操作数标记为污点数据且更新为计算结果	目的操作数标记为清白数据	—	—	—
压栈指令	将污点数据保存进栈污点列表中	—	—	—	—
出栈指令	从栈污点列表中弹出, 并将对应寄存器标记为污点数据	—	—	—	—

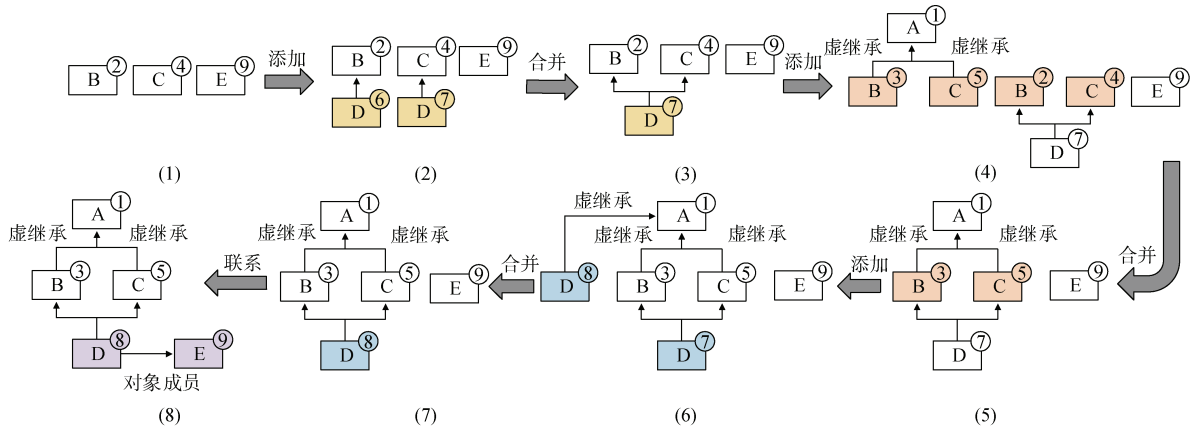

图 8　节点生成过程

Figure 8　The process of the node generation

H-1: 同一内存偏移处的 *vftable* 所属的类存在继承关系, 负责类继承关系的恢复。

H-2: 同一个类的不同的 *vftable* 具有相同的析构函数, 负责节点合并。

H-3: 带有 *vbase* 的内存偏移中的第一个 *vftable* 所属的类为虚基类, 后面的类都具有虚继承关系, 负责虚继承关系和虚基类的恢复。

H-4: 对象成员的 *vftable* 的相关覆写指令存在与其所属类的析构函数中, 且对象成员的覆写顺序大于所属类的覆写顺序, 负责对象成员的恢复。

对于每一个对象内存布局, 节点生成具体包括以下步骤:

1) **单一继承与多重继承分析:** 循环提取每个具有 *vftable* 属性的内存偏移中的 *vftable*, 每次循环从每个内存偏移处提取一个 *vftable*, 将该 *vftable* 从对象内存布局中删除, 独立成为一个节点, 并与上一个 *vftable* 所属的节点建立继承关系。在每次循环中, 检测各 *vftable* 之间是否存在相同的析构函数, 若存在则进行节点的合并, 合并时覆写顺序保留数值大的。

2) **虚继承分析:** 对于每个具有 *vbase* 属性的内存偏移, 从第二个 *vftable* 开始依次提取, 提取的 *vftable* 从对象内存布局中删除, 独立成为一个节点, 并与第一个 *vftable* 所属的节点(虚基类)建立虚继承关系, 其中提取第二个 *vftable* 时会将第一个 *vftable* 也提取。与此同时, 遍历所有节点, 寻找与该 *vftable* 具有相同析构函数的节点, 并将其进行节点合并, 合并时覆写顺序保留数值大的。若寻找到的节点的父节点或者父节点以上的节点具有虚继承关系, 则删除该 *vftable* 所属节点与虚基类的虚继承关系。

3) **对象成员分析:** 若存在孤立的节点或者节点树, 记录其最下面的子节点的 *vftable*, 覆写顺序, 以

代表该孤立节点或者节点树, 且该子节点为记录节点。遍历其他节点树, 分析其他节点树中的节点的析构函数是否包含记录节点的 *vftable*, 若包含且其节点覆写顺序小于记录节点的覆写顺序, 则将记录节点所属的节点或者节点树(对象成员)归到对应节点上, 建立对象成员关系。

对于图 1 中的 D, 其节点生成过程如图 8 所示。

因为不同的对象可能包含相同的类节点, 所以我们需要再把所有节点进行聚合和去重操作。最后, 我们便得到了恢复的类信息, 其中包括类、类关系、类方法等信息。

4.5 对 Itanium ABI 的支持

本文的解决方案是一种通用的解决方案, 对于其他平台, 只需要依据其平台特性进行一些工程化的开发和修改, 便可适用。下面给出支持 Itanium ABI 需要修改的地方。

Itanium ABI 的实现与 MSVC ABI 的实现有轻微的不同。其中, *vftable* 在 RTTI 上面多了 *OffsetToTop* 和 *OffsetToVbase* 两个字段。*OffsetToVbase* 记录了 this 指针到虚继承的偏移, *OffsetToTop* 记录了 this 指针到对象内存中第一个 *vftptr* 的偏移。除此之外, Itanium ABI 中不存在 *vbtable*, 取而代之的是 *VTT* (*Virtual Table Table*)。*VTT* 记录了虚继承关系的派生类和中间基类的 *vftable* 的地址。

Itanium ABI 处理虚继承的过程类似于 MSVC ABI, 也涉及了 *VTT* 的间接引用。当程序执行到带有虚继承的基类的构造函数时, 隐藏的参数不止有 this 指针, 还有 *VTT* 的地址。另外, 相关 *vftable* 的内存写操作也没有使用立即数, 而是通过将 *VTT* 中存储的 *vftable* 地址写入对应内存中。虚基类的偏移的计算也依靠 *VTT*, 先从 *VTT* 中取对应的 *vftable* 的地址, 然后取到这个 *vftable* 中的 *OffsetToVbase* 字段, 进而计算出虚基类的偏移。

对于 RECLASSIFY, 主要需要改变的就是 *VTT* 获取。在提取 *VTT* 之前, 我们需要先识别每个 *vftable* 对应的析构函数, 通过析构函数可以界定 *VTT* 的边界。每个派生类的 *VTT* 的 *vftable* 的地方都分布在 *VTT* 的开头和结尾。另外, *VTT* 也存在二进制文件中的只读段中。根据以上两条启发式便可提取出 *VTT*。

当从 Itanium ABI 二进制文件中恢复类和类关系时, 首先提取 *vftable*, 这一步与 MSVC ABI 相同。然后为每个 *vftable* 找到对应的析构函数, 并且利用析构函数来提取 *VTT*, 这一步与 MSVC ABI 不同。在那之后, 利用 *vftable* 搜索构造函数并为构造函数和析构函数生成 CFG。之后执行静态污点分析, 这里与 MSVC ABI 有一点轻微的不同。当处理虚基类时, *VTT* 不从对象内存中获取, 而是直接从只读段中获取, 这一点实现起来要比 MSVC ABI 容易。当程序通过派生类或者非虚基类的 *vftable* 获取 *OffsetToVbase* 字段时(例如, *mov vftptr, [VTT + offset]; sub vftptr, 0x20;*), 污点系统会捕捉这些操作并计算出虚基类的偏移, 然后将对象内存布局中对应的内存偏移的内容标记成“*vbase*”。最后使用对象内存布局来恢复类信息, 这步与 MSVC ABI 相同。另外, 存在一种特殊情况, 当派生类直接虚继承虚基类, 并且派生类没有任何子类时, 程序会使用立即数来获取虚基类的偏移, 而不是通过 *VTT* 和 *OffsetToVbase*。我们通过在静态污点分析前提取每个 *vftable* 的 *OffsetToVbase* 来解决这种情况。在执行完静态污点分析后, 我们检查每个对象内存布局中内存偏移为 0 的最后一个 *vftable*(派生类的 *vftable*), 如果它包含 *OffsetToVbase*, 则对应的内存偏移的内容被标记为“*vbase*”。

4.6 细节问题与解决方法

RECLASSIFY 是一种通用的解决方案, 其中, 信息提取和析构函数与构造函数分析使用了 IDA Python[25]进行提取。在这两个分析过程中, 因为我们没有使用 CFG, 所以这里的搜索精度较粗, 但是这并不影响后续的分析结果, 而且还能提高分析的效率。为了提高 RECLASSIFY 的可扩展性, 使得 RECLASSIFY 可以分析其他平台的二进制文件, 后续的分析都在中间语言 VEX IR[26]上进行分析。在进行静态污点分析之前, 我们将二进制文件的汇编代码转换成中间语言 VEX IR, 这里借助了 angr[27]的帮助。

除此之外, 我们在实现过程中遇到一些比较棘手的细节问题, 下面进行详细的阐述, 并给出对应的解决方法。

具有虚继承关系的非多态类: 在真实世界中的二进制文件中, 有些具有虚继承关系的派生类不是多态类, 在这种情况下, 其 *vbtable* 的第一个字段是 0。虽然本文重点关注多态类的识别, 但是这种情况的类只要进行一些细微的调整也可进行识别。其方法是在信息提取阶段将 *vbtable* 的提取条件限制放宽, 变成第一个字段是 0/-4/-8。这样虽然会造成很多的误报, 识别出了很多不是 *vbtable* 的数据, 但是因为只有真正的 *vbtable* 才会在后续分析中被使用, 所以这些冗余的信息不会对分析结果造成影响。

异常处理对分析的干扰: 类的异常处理也会调用析构函数, 在 IDA Pro 中, 异常处理与它对应的函数放在了一个集合中, 这会导致对启发式搜索构造函数造成干扰。因为启发式搜索构造函数是基于指

令遍历的, 不是基于 CFG 的, 而异常处理中的析构函数中也存在 *vftable* 的写操作, 若存在构造函数完全内联的情况, 如章节 2 所述, 则异常处理也会满足存在 *vftable* 写操作和在异常处理前有 new 操作的情况, RECLASSIFY 将误判异常处理为构造函数, 对后续的分析造成了影响。我们通过观察可以发现, 经过编译器编译以后, 异常处理的代码的地址空间与相对应的函数的地址空间相差很大, 根据这个启发式, 我们将异常处理的代码排除掉, 便可进行正常的分析。

不常见的操作指令: angr 在将汇编代码转换中间语言 VEX IR 时, 有一些不常见的指令因为无法进行处理便被忽略掉了, 比如条件跳转指令(*jrcxz*), 这会导致控制流的缺失, 从而影响 RECLASSIFY 后续的分析。这属于 angr 自身的 bug, 我们通过额外添加处理这些不常见的操作指令的代码来解决这个问题。

多个对象的构造函数内联进同一个函数的情况: 在 O2 编译优化下, 有时会出现多个对象的构造函数内联进同一个函数的情况, 这会导致 RECLASSIFY 在静态污点分析阶段计算内存偏移时产生错误, 这是因为 RECLASSIFY 没有对每个对象划分界限, 将多个对象误认为成一个对象。解决的方法是, 若检测到这种情况时, 我们将该分析的起点函数做上标记, 并在后续重新分析。重新分析的候, 我们将每一个 new 操作作为一个分割点, 以此来划分各个对象的界限。然后, 我们对每个对象进行独立的静态污点分析, 以便降低 RECLASSIFY 漏报或误报的可能性。

未初始化的多态类: 在真实世界中, 有些多态类在源代码中定义了, 但却没有使用, 经过编译之后, 其二进制文件中不存在这些类的构造函数, 但是这些类可能产生新的攻击面, 所以具有很大的恢复价值。然而, 如果只对构造函数进行静态污点分析可能会漏掉很多未使用的多态类。其解决方法是分析他们的析构函数, 如章节 2 所述, 因为为了防止内存泄露, 析构函数一般都是虚函数。但是却不能只分析析构函数而不分析构造函数, 因为析构函数中关于 *vftable* 和 *vbtable* 的操作指令可能会有缺失, 而构造函数中所包含的信息会更加全面一些。所以最佳的解决方案是我们先使用构造函数进行静态污点分析, 然后对于没有分析到的 *vftable*, 对它们的析构函数进行静态污点分析。这样的组合分析策略既可降低 RECLASSIFY 的漏报率, 又可降低 RECLASSIFY 的误报率。

静态对象: 静态对象不会使用 *new*(), 所以我们不能识别静态对象的构造函数。然而, 为了防止内存泄露, 析构函数一般是虚函数所以不会被函数内联, 并且析构函数中也存在 *vftable* 的内存写操作, 所以我们可以通过分析析构函数来处理静态对象。

5 实验评估

我们使用真实的 C ++程序的数据集来评估 RECLASSIFY 的有效性, 并将 RECLASSIFY 与 OOAnalyzer 进行比较。在已发表的论文中, 尽管一些研究工作取得了一定的成就, 但是我们无法获得这些工作的源代码, 并且这些工作提供的详细信息还不足以复现。因此, 可以代表最前沿的当前工作是 OOAnalyzer。

5.1 数据集的选取与参考标准的提取

因为已有的研究工作的数据集中的二进制文件大小普遍偏小, 而且所涉及的领域不够全面, 不能很好的代表真实世界的二进制文件的标准, 所以我们没有使用它们的数据集, 而是自己构造了一个数据集[28]。为了能全面地测试 RECLASSIFY 的性能, 我们从 Github 上热门项目排名前 1000 的 C++项目中选取 9 个项目, 15 个二进制文件, 涉及领域有密码学, 科学计算, 数据分析, 区块链, 金融, 图像处理等。然而, 从源码中提取参考标准并不容易, 其难点如下:

- 一个项目中有多个二进制文件, 每个二进制文件对应的源码不一样, 并且有的类声明了, 但没有被使用, 便不会被编译进二进制文件中。因此, 从源码中自动化提取参考标准是困难的。
- 一个项目一般会用很多系统库或者第三方库中的类(如 Boost 等), 这加大了从源码中提取参考标准的难度。

针对这个难题, 我们采用了通过 PDB 文件来进行提取的方法去解决。这里使用了微软公布的 Debug Interface Access Software Development Kit(DIA SDK)[29]来进行编程, 可以读取二进制文件对应的 PDB 文件中类信息, 进而可以提取出参考标准。除此之外, 因为 OOAnalyzer 可以恢复多态类和非多态类, 为了进行公平的比较, 我们将 OOAnalyzer 识别的非多态类移除。

对于 Itanium ABI, 因为其编译器生成的二进制文件没有对应的 PDB 文件, 所以我们通过分析 Itanium ABI 二进制文件中的 RTTI 结构来提取参考标准。

本文所有编译的二进制文件皆为 O2 编译优化等级, 其中 MSVC ABI 下是使用 VS2017 进行编译的, Itanium ABI 下是使用 GCC5.4.0 进行编译的。

5.2 类的识别

如表 2 所示, 这是 RECLASSIFY 和 OOAnalyzer 在识别类上面的对比。其中, GT 表示参考标准(Ground Truth)。F score 是召回率和准确率的加权调和平均值, 能够更加客观全面的反应一个工具的好坏。召回率中的 X/Y 表示 RECLASSIFY 识别出来的正确的类的数量与数据集参考标准的类的数量的比值, 准确率中的 X/Y 表示 RECLASSIFY 识别出来的正确的类的数量与 RECLASSIFY 识别出来类的总数的比值。结果显示, RECLASSIFY 平均召回率为 84.36%, 相比 OOAnalyzer 高了 50%左右, 而且 F score 也是它的二倍多, 可以证明 RECLASSIFY 在恢复多态类上的优越性。除此之外, RECLASSIFY 也具有较高的准确率(97.17%)。

如表 3 所示, 这是 RECLASSIFY 在 Itanium ABI 下识别类的评估结果。结果显示, RECLASSIFY 在 Itanium ABI 下识别类的平均召回率和准确率分别为 80.06%和 97.54%, F score 为 0.88, 这与 MSVC ABI

表 2 MSVC ABI 类的识别

Table 2 Class recovery under MSVC ABI

二进制文件	大小 (KB)	GT	RECLASSIFY				OOAnalyzer			
			召回率	准确率	F score	时间 (h:m:s)	召回率	准确率	F score	时间 (h:m:s)
Cryptopp.dll	2041	287	213/287	213/228	0.83	00:01:09	131/287	131/131	0.63	02:25:09
muparser.dll	379	15	14/15	14/14	0.97	00:00:04	10/15	10/10	0.80	00:15:53
libz3.dll	17065	1130	693/1130	693/694	0.76	00:18:50	545/1130	545/547	0.65	102:00:35
solc.exe	7273	1002	867/1002	867/947	0.95	00:02:39	0/1002	×	0	35:25:23
yulrun.exe	3244	629	572/629	572/584	0.94	00:01:30	0/629	×	0	09:46:30
yulopti.exe	2113	456	413/456	413//426	0.94	00:00:45	0/456	×	0	04:31:53
MarkerMod-els.exe	3907	125	95/125	95/99	0.85	00:01:27	48/125	48/48	0.55	01:22:52
Repo.exe	522	132	119/132	119/121	0.94	00:00:11	34/132	34/34	0.41	00:25:31
QuantLib.dll	22185	1954	1468/1954	1468/1720	0.80	00:04:18	274/1954	274/276	0.25	50:59:02
mysql.exe	4127	74	62/74	62/62	0.91	00:00:11	34/74	34/34	0.63	01:01:10
libmysql.dll	3995	74	61/74	61/61	0.90	00:00:10	32/74	32/32	0.60	00:53:34
protobuf.dll	1345	86	79/86	79/79	0.96	00:00:06	67/86	67/67	0.88	02:58:35
opencv_core.dll	3956	106	89/106	89/91	0.90	00:01:27	58/106	58/58	0.71	11:16:53
opencv_dnn.dll	5024	444	358/444	358/358	0.89	00:04:23	0/444	×	0	21:45:34
libzmq.dll	623	98	97/98	97/102	0.97	00:00:07	45/98	45/95	0.47	00:56:38
平均值			**84.36%**	**97.17%**	**0.90**		**33.76%**	**95.12%**	**0.44**	

表 3 Itanium ABI 类的识别

Table 3 Class recovery under Itanium ABI

二进制文件	大小(KB)	GT	召回率	准确率	F score	时间(h:m:s)
libcrytopp.so	37142	656	406/656	406/422	0.75	00:07:24
libmuparser.so	408	6	6/6	6/6	1	00:00:30
libz3.so	23585	1185	1009/1185	1009/1025	0.91	00:11:18
solc	8294	494	419/494	419/425	0.91	00:07:54
yulrun	4029	236	182/236	182/185	0.86	00:02:15
yulopti	2649	190	149/190	149/151	0.87	00:01:08
MarketModels	1965	19	14/19	14/14	0.85	00:00:02
Repo	1111	18	14/18	14/14	0.88	00:00:02
libQuantLib.so	450101	1979	1460/1979	1460/1705	0.79	00:42:03
mysql	10975	65	53/65	53/53	0.90	00:00:18
libmysqlclient.so	9581	65	53/65	53/53	0.90	00:00:14
libprotubuf.so	9596	83	60/83	60/60	0.84	00:00:20
libopencv_core.so	5222	89	69/89	69/71	0.86	00:02:00
libopencv_dnn.so	5201	370	324/370	324/324	0.93	00:01:16
libzmq.so	1155	82	72/82	72/80	0.89	00:00:19
平均值			**80.06%**	**97.54%**	**0.88**	

下的评估结果相似。

5.3 类关系的识别

类关系的评估相对复杂, 它包含了类关系类型的正确性与类关系方向的正确性。已有的研究工作在这方面没有一致的评估标准, 我们这里采用了一种相对严苛的评估标准来避免高估和低估, 即统计类关系的数量而不是不同类关系的类的数量, 并且同时考虑类关系的方向性。举个例子, 假设有一个拥有三个直接基类和一个对象成员的派生类, 这里类关系的数量的参考标准就是 4。如果 RECLASSIFY 恢复派生类与 2 个基类之间的关系, 对象成员的类关系方向识别反了, 那么 RECLASSIFY 识别类关系的召回率和准确率分别是 2/4(50%)和 2/3(66.67%)。

表 4 类关系的识别

Table 4 Class relationship recovery

MSVC ABI						Itanium ABI			
二进制文件	GT	RECLASSIFY		OOAnalyzer		二进制文件	GT	RECLASSIFY	
		召回率	准确率	召回率	准确率			召回率	准确率
Cryptopp.dll	168	32/168	32/67	0/168	×	libcrytopp.so	708	43/708	43/110
muparser.dll	8	3/8	3/4	0/8	×	libmuparser.so	2	2/2	2/2
libz3.dll	1094	47/1094	47/195	2/1094	2/4	libz3.so	1147	183/1147	183/466
solc.exe	209	68/209	68/155	0/209	×	solc	593	47/593	47/113
yulrun.exe	106	18/106	18/49	0/106	×	yulrun	305	17/305	17/29
yulopti.exe	91	18/91	18/51	0/91	×	yulopti	225	16/225	16/28
MarkerModels.exe	73	8/73	8/21	0/73	×	MarketModels	17	2/17	2/2
Repo.exe	45	8/45	8/23	0/45	×	Repo	22	2/22	2/2
QuantLib.dll	1047	104/1047	104/364	5/1047	5/8	libQuantLib.so	2078	89/2078	89/200
mysql.exe	64	3/64	3/13	0/64	×	mysql	66	9/66	9/10
libmysql.dll	57	4/57	4/13	0/57	×	libmysqlclient.so	66	9/66	9/10
protobuf.dll	72	1/72	1/24	0/72	×	libprotubuf.so	62	1/62	1/1
opencv_core.dll	48	9/48	9/43	0/48	×	libopencv_core.so	81	6/81	6/10
opencv_dnn.dll	123	21/123	21/207	0/123	×	libopencv_dnn.so	345	8/345	8/19
libzmq.dll	89	53/89	53/65	0/89	×	libzmq.so	90	15/90	15/19
平均值		**18.49%**	**35.67%**	**0.04%**	×			**14.87%**	**69.42%**

另外, 如果 RECLASSIFY 识别错了类关系的类型, 比如将虚继承或者对象成员识别成普通的继承关系, 这也被判定为是一种假阳性的情况。

如表 4 所示, 这是在 MSVC ABI 和 Itanium ABI 下类关系的评估结果。其中, GT 表示参考标准(Ground Truth)。召回率中的 X/Y 表示 RECLASSIFY 识别出来的正确的类关系的数量与数据集参考标准的类关系的数量的比值, 准确率中的 X/Y 表示 RECLASSIFY 识别出来的正确的类关系的数量与 RECLASSIFY 识别出来类关系的总数的比值。结果显示, 在 MSVC ABI 下, 虽然 RECLASSIFY 恢复的并不好, 召回率只有 18%左右, 但是相比于 OOAnalyzer 的 0.04% 依然提升了不少, 也从侧面证明了 RECLASSIFY 在类关系识别的优越性, 并且这方面在后续的工作中可以继续深入研究。

在 Itanium ABI 下, 结果显示, RECLASSIFY 恢复类关系的平均召回率为 14.87%, 而 RECLASSIFY 恢复类关系的平均准确率为 69.42%, 远高于 RECLASSIY 在 MSVC ABI 下恢复类关系的平均准确率(35.67%)。

如图 9 所示, 是 RECLASSIFY 分析 libQuantLib.so 从而恢复出来的类结构图局部。结果显示, 该类结构图的最大深度为 7, 最大宽度为 21(参考标准的最大深度为 7, 最大宽度为 109)。研究人员可以从

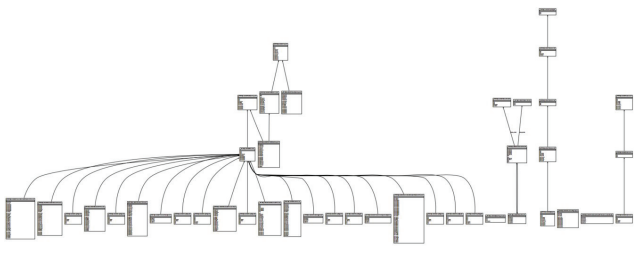

图 9 RECLASSIFY 恢复的类结构图局部

Figure 9 Partial class structure diagram by RECLASSIFY

RECLASSIFY 恢复的类结构图中较好的了解程序的高层抽象信息和整体架构, 对于快速理解程序具有重要的意义, 证明了 RECLASSIFY 的实用价值。

5.4 分析效率

实验环境为 Ubuntu 16.04 LTS, Intel Core i7-6700HQ CPU@2.60GHz, 12GB RAM。如表 5 所示, 这分别是在 MSVC ABI 和 Itanium ABI 下每个二进制文件中总函数的数量和分析函数的数量与占比。我们发现, 在实际分析中, 分析的函数平均占总体函数的 20%左右, 这证明了使用自适应 CFG 生成算法的有效性。

如表2和图10所示, 评估结果显示了RECLASSIFY和 OOAnalyzer 在时间开销上的对比, 可以看出 RECLASSIFY 相比 OOAnalyzer 分析效率提升了 3 个数量级, 证明了 RECLASSIFY 的优越性和实用性。

表 5 数据集函数统计

Table 5 Function statistics for data set

MSVC ABI					Itanium ABI				
二进制文件	大小 (KB)	总函数数量	分析函数数量	占比(%)	二进制文件	大小(KB)	总函数数量	分析函数数量	占比(%)
Cryptopp.dll	2041	4736	1357	28.65	libcrytopp.so	37142	8761	2102	23.99
muparser.dll	379	1069	267	24.98	libmuparser.so	408	898	52	5.79
libz3.dll	17065	38238	6237	16.31	libz3.so	23585	31254	10360	33.15
solc.exe	7273	20513	6276	30.60	solc	8294	9856	3764	38.19
yulrun.exe	3244	10772	3029	28.12	yulrun	4029	4883	1769	36.23
yulopti.exe	2113	7806	2086	26.72	yulopti	2649	3466	1196	34.51
MarkerModels.exe	3907	2154	450	20.89	MarketModels	1965	418	41	9.81
Repo.exe	522	1515	483	31.88	Repo	1111	316	24	7.59
QuantLib.dll	22185	44161	7418	16.80	libQuantLib.so	450101	24969	6042	24.20
mysql.exe	4127	3454	500	14.48	mysql	10975	3581	328	9.16
libmysql.dll	3995	3248	500	15.39	libmysqlclient.so	9581	2865	328	11.45
protobuf.dll	1345	5777	454	7.86	libprotubuf.so	9596	3079	540	17.54
opencv_core.dll	3956	7042	597	8.48	libopencv_core.so	5222	5331	442	8.29
opencv_dnn.dll	5024	12621	2560	20.28	libopencv_dnn.so	5201	7090	1671	23.57
libzmq.dll	623	2272	411	18.09	libzmq.so	1155	2991	257	8.59
平均值				**20.64**					**19.47**

如表 3 所示, 评估结果显示了 RECLASSIFY 在 Itanium ABI 下的时间开销。其中, RECLASSIFY 仅用了 42 min 就分析完了 440M 大小的二进制文件 (libQuantLib.so), 这说明了 RECLASSIFY 可以分析大型的二进制文件。

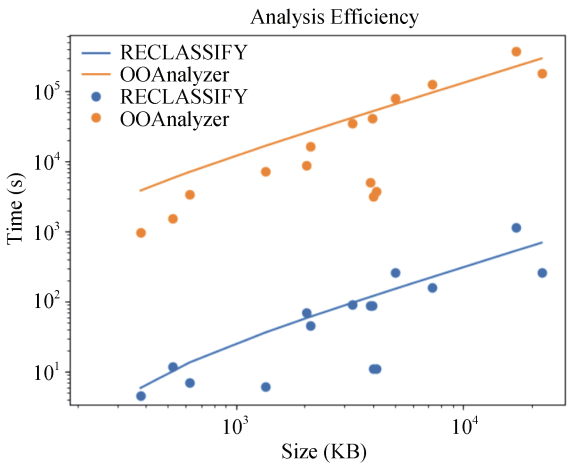

图 10 分析效率

Figure 10 Analysis efficiency

6 讨论

本章主要讨论本文工作的一些不足之处, 并给出未来可能解决的方法。

不能处理的 this 指针传递模式: 在静态污点分析过程中, 在进入基类或者对象成员的构造函数之前会将 this 指针传递到 rcx 寄存器中, 但是在一些二进制文件中会使用 rbp 寄存器获取上层函数的变量并传递给 rcx 寄存器(比如 *lea rcx,[rbp + offset]; call constructor*)。当上层函数在分析的起始函数之上的时候, 这会导致污点的缺失, 这是因为无法获取上层函数的变量, 也就无法将变量的值传递给 rcx 寄存器。因此 RECLASSIFY 无法识别与之相关的 *vftable*

的内存写操作, 进而影响了其识别类和类关系的召回率和准确率。其可能的解决方法是在构建 CFG 的时候对初始函数使用后向切片来进行寻找, 然而可能存在多个函数调用初始函数的情况, 而程序实际运行时只有一个函数是正确的, 所以这仍然是一个待解决的问题, 未来的研究可以继续深入。

间接调用 *new*()和 *delete*(): 为了高效的识别类和类关系, 识别调用 *new*()和 *delete*()的位置是很重要的。然而, RECLASSIFY 使用了模式匹配的方法来识别 *new*()和 *delete*()的函数符号(如 *malloc*, *free* 等), 所以这种方法是无法识别间接调用的, 进而产生识别类的假阴性(例如 Cryptopp.dll 中的 pcharNode 类)和假阳性(例如 Repo.exe 中的 clone_impl 类)。其中, 有些由编译器生成的间接函数调用依旧可以通过其特征来进行模式匹配, 比如 *call cs:__guard_dispatch_icall_fptr*, 而其他的情况则无法识别, 如 *call rax*。其解决方法是解决间接函数调用识别的问题, 这是一个挑战, 未来可以进行深入的研究。除此之外, 另一个解决方法则是换一个思路, 因为识别 *new*()和 *delete*()的目的是划分对象之间的界限, 所以可以寻找其他的划分对象界限的方法, 比如通过对象成员的访问操作和虚函数调用操作来推测对象的大小, 这也可以作为将来的一个研究方向。

智能化路径选择策略的缺陷: 智能化路径选择策略在静态污点分析中起到了很重要的作用, 然而因为它只是模拟程序执行流程, 所以选择的路径可能不是程序运行时的真实路径, 这可能会导致不相关的类之间建立了类关系, 进而产生了识别类关系的假阳性(例如 opencv_dnn.dll 中的 SoftmaxLayer 类和 ectMapConflicts 类)。比如, RECLASSIFY 可能将一个类与它的异常处理相关的类建立了类关系, 而他们之间本来是不存在联系的。一种解决思路是使用混合符号执行, 利用约束求解或者实际运行的值来进行路径的选择, 这种方法虽然能极大地提高准确率, 但是因为符号执行本身固有的路径爆炸问题而会导致很多的时间开销。另一种解决思路是使用剪枝算法, 提前将无关的路径去除掉, 不过它的效果取决于具体剪枝的程度, 会存在误报和漏报的情况。这两种解决思路在未来都可以进行深入的研究。

源码中缺少析构函数定义: 在真实世界中, 有些类的析构函数没有在源码中定义, 编译器自动生成的析构函数是没有对应 *vftable* 的内存写操作的, 进而造成了 RECLASSIFY 识别类关系的假阴性。例如, 在C++标准库中, bad_array_new_length类继承了bad_alloc类, bad_alloc类继承了exception类, 其中只有 exception 类定义了虚析构函数, 其他两个类都没有定义析构函数。所以在二进制文件中, bad_array_new_length 类的析构函数中只存在 exception 类的 vftable 的内存写操作。解决方案是可以利用析构函数中的控制流信息来恢复类继承关系, 因为析构函数一般是虚函数, 不会被内联, 其控制流信息可以获取到。

非虚析构函数: 本文假设了析构函数都是虚函数, 并且这是合理的。然而在实际的商业软件中, 由于程序员的代码编写不规范, 依然存在非虚析构函数的情况。这会造成 RECLASSIFY 无法合并 *vftable*, 可能将具有多继承关系的类识别为多个类, 进而在识别类关系上产生假阳性(例如, QuantLib.dll 中的 Gaussian1dModel 类)。另外, 如果在非虚析构函数中存在 *new* 操作, RECLASSIFY 可能将其误认为构造函数, 造成其识别的类继承关系的方向相反, 进而在识别类关系上产生假阴性(例如, mysql.exe 中的 BulkCipher 类和 DES 类)。可能的解决方案是改变合并 *vftable* 的方法和识别构造函数的方法, 这方面可以在未来进行探索。

7 相关工作

我们的工作重点是从商业软件中恢复类信息, 因此需要解决两个问题, 一个是处理复杂编译优化的情况(构造函数内联与地址优化), 另一个是提高分析效率。与我们最相关的研究是类关系、类成员结构和虚函数调用检测。其中, 一些研究工作[30-37]检测虚函数调用逃逸漏洞并通过获取 *vftable* 制定相应的控制流完整性(CFI)策略。除此之外, 一些研究工作[9-20,38]则会恢复类信息, 例如类, 类方法, 类成员和类关系等。如图 11 所示, 我们对近几年的解决方案在多个方面进行了详细的分析, 以前的解决方案在不同方面取得了一定的成就。根据实验结果, RECLASSIFY 可以从复杂编译优化下恢复类信息, 并且它首次提出了自适应 CFG 生成算法的思想, 极大地提高了分析效率, 这使得分析大型二进制文件成为了可能。下面对已有的研究工作进行介绍。

Lego[19]和 OBJDigger[14]都聚焦在恢复单一继承和对象成员。Lego 通过监视动态对象的生命周期及其调用的方法来恢复继承关系, 它通过分析析构函数来恢复继承方向。然而, Lego 的恢复效果取决于测试用例触发了多少个类方法。OBJDigger 利用符号执行和过程间数据流分析来通过跟踪跨函数的指针来恢复对象实例, 数据成员和类的方法。

		VCI	Marx	OOAnalyzer	DeClassifier	VirtAnalyzer	RECLASSIFY
前提	假设	存在0偏移处的vftptr初始化的函数是构造函数	vftable地址连续分布	new()的下一个函数是构造函数	vftable地址连续分布	vftable地址连续分布	析构函数是虚函数
	关注点	虚函数调用	虚函数调用	多态类，非多态类	内联构造函数	虚继承	复杂编译优化，分析效率
分析	涉及信息	vftable, 调用构造函数的指令信息, class layout	vftable, vftptr 初始化信息	vftable, new()的下一个函数, entity facts	vftable, minimum object size, vftable size	vftable, VTT, OffsetToVbase, 调用虚基类的指令信息	vftable, vbtable/VTT, object memory layout
	抵抗内联优化	无	弱 (overwrite analysis)	中 (forward reasoning, hypothetical reasoning)	中 (object layout analysis)	无	强 (inter-procedural static taint analysis)
	抵抗地址优化	弱 (基于构造函数合并)	无	中 (基于虚函数调用合并)	无	无	强 (基于析构函数合并)
	分析效率	低	低	低	低	低	高
	关键技术	ctor analysis	overwrite analysis	symbolic execution, hypothetical reasoning	ctor-dtor analysis, object layout analysis	ctor-dtor analysis, VTTMap	static taint analysis, heuristic reasoning, adaptive CFG
结果	实验平台	Itanium	Itanium	MSVC	Itanium	Itanium	MSVC, Itanium
	实体	多态类	多态类	多态类，非多态类	多态类	多态类	多态类
	类关系	单一继承，多重继承	单一继承，多重继承 (无方向)	单一继承，多重继承，对象成员	单一继承，多重继承，对象成员	单一继承，多重继承，虚继承	单一继承，多重继承，对象成员，虚继承

图 11　解决方案分析

Figure 11　Solution analysis

OOAnalyzer[17]是 OBJDigger 的改进版本。它可以恢复多态和非多态类。OOAnalyzer 通过轻量级的符号执行提取基本信息, 然后使用假设的推理引擎推断该信息以捕获更高的抽象。

VCI[10], Hex Rays[18,37], SmartDec[12-13], Yoo 等人[20]和 Marx[16]都聚焦在恢复单一继承和多重继承。VCI 通过捕获与 *vftable* 相关的内存写操作来识别构造函数, 并将*vftable*与相关的构造函数配对, 然后通过构造函数的调用顺序恢复类关系。Hex Rays 与 VCI 相似, 它将 *vftable* 与相关的构造函数配对以恢复继承。SmartDec 不仅可以恢复类和继承, 还可以恢复异常处理结构。它通过分析 *vftable* 的大小和使用纯虚函数来恢复继承。Yoo 等人通过 RTTI 恢复类和继承关系。但是, 商业软件通常会禁用 RTTI, 因此这种方法会失效。Marx 主要依靠 *vftable* 分析和覆盖分析, 虽然可以抵抗函数内联的影响, 但是不能恢复继承的方向。

DeClassifier[11]聚焦在构造函数内联。它通过构造函数/析构函数分析, 覆盖分析和对象布局分析来恢复类关系。其中它通过最小对象大小和 *vftable* 大小来推断继承的方向。

VirtAnalyzer[9]是 DeClassifier 的改进版本。它聚焦在恢复虚继承。VirtAnalyzer 从二进制文件中提取 *VTT* 和 *vftable*, 并将 *OffsetToVbase* 映射到类。然后, VirtAnalyzer 监视有关虚基类的构造函数的“*call*”指令和参数以恢复虚继承。

Katz O 等人[15]的方法是一种新颖的解决方案。它采用机器学习来构建预测模型, 以预测虚拟函数调用所属的类。但是, 在复杂编译优化环境下, 其训练集中的标签数量太少, 即已知类的数量远远少于基本事实, 这极大地限制了预测模型的能力。

8　结论

本文提出了一种实用且有效的解决方案, 它可以从经过编译优化的 C ++二进制文件中恢复多态类和类关系(包括虚拟继承)。我们使用真实的商业软件来评估我们的解决方案。实验结果证明, 该解决方案可以在复杂编译优化环境下从二进制文件中恢复类和类关系, 并且分析效率很高。因此, 我们的解决方案可以分析大型的二进制文件, 而已有的研究工作

则不能。

本文介绍了一个名为 RECLASSIFY 的系统的设计和实现, 并根据从 9 个真实的 C ++项目(O2 编译优化)编译的 15 个二进制文件对该系统进行了评估, 该二进制文件涵盖了各个应用领域。结果表明, 在 MSVC ABI 下, RECLASSIFY 的平均类召回率达到 84.36%, 远高于目前最好的解决方案(OOAnalyzer)的 33.76%。除此之外, 恢复类中 RECLASSIFY 的平均 F score 是 0.90(是以前工具的两倍)。此外, 与已有的研究工作相比(提升三个数量级), RECLASSIFY 的分析效率已大大提升, 可用于分析大型二进制文件。

参考文献

[1] Office, Microsoft, https://www.office.com/, 2020.

[2] Adobe Reader, Adobe, https://acrobat.adobe.com/us/en/acrobat/pdf-reader.html, 2020.

[3] Windows Defender, https://en.wikipedia.org/wiki/Windows_ Defender, 2020.

[4] Chrome, Google, https://en.wikipedia.org/wiki/Google_Chrome, 2019.

[5] Firefox, Mozilla, https://www.mozilla.org/en-US/, 2020.

[6] MySQL, MySQL, https://www.mysql.com/cn/, 2019.

[7] RTTI, wikipedia, https://en.wikipedia.org/wiki/Run-time_type_information, 2019.

[8] PRISM, wikipedia, https://en.wikipedia.org/wiki/PRISM_ (surveillance_program), 2019.

[9] Devil is Virtual: Reversing Virtual Inheritance in C++ Binaries, Rukayat Ayomide Erinfolami, A.P, https://arxiv.org/abs/2003.05039, 2020.

[10] Elsabagh M, Fleck D, Stavrou A. Strict Virtual Call Integrity Checking for C++ Binaries[C]. *The 2017 ACM on Asia Conference on Computer and Communications Security*, 2017: 140-154.

[11] Erinfolami R A, Prakash A. DeClassifier: Class-Inheritance Inference Engine for Optimized C++ Binaries[C]. *The 2019 ACM Asia Conference on Computer and Communications Security*, 2019: 28-40.

[12] Fokin A, Derevenetc E, Chernov A, et al. SmartDec: approaching C++ decompilation[C]. *2011 18th Working Conference on Reverse Engineering*, 2011: 347-356.

[13] Fokin A, Troshina K, Chernov A, et al. Reconstruction of class hierarchies for decompilation of C++ programs[C]. *2010 14th European Conference on Software Maintenance and Reengineering*, 2011: 240-243.

[14] Jin W, Cohen C, Gennari J, et al. Recovering C++ Objects from Binaries Using Inter-Procedural Data-Flow Analysis[C]. *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, 2014: 1-11.

[15] Katz O, El-Yaniv R, Yahav E. Estimating Types in Binaries Using Predictive Modeling[C]. *The 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016: 313-326.

[16] Pawlowski A, Contag M, van der Veen V, et al. MARX: uncovering class hierarchies in C++ programs[C]. *Proceedings 2017 Network and Distributed System Security Symposium*, 2017: 1-15.

[17] Schwartz E J, Cohen C F, Duggan M, et al. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 426-441.

[18] Reversing microsoft visual c++ part 2: Classes, methods and rtti, Skochinsky, I., http://www.openrce. org/articles/full_view/23, 2018.

[19] Srinivasan V, Reps T. Recovery of Class Hierarchies and Composition Relationships from Machine Code[M]. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014: 61-84.

[20] Yoo K, Barua R, Processing C A. Recovery of object oriented features from C++ binaries[C]. *2014 21st Asia-Pacific Software Engineering Conference*, 2015: 231-238.

[21] Itanium ABI, Itanium, https://github.com/itanium-cxx-abi/cxx-abi, 2018.

[22] C++: under the hood, Gray, J., http://www.openrce.org/articles/fles/jangrayhood.pdf, 2019.

[23] Feng Z N, Wang Z Y, Dong W Y, et al. Bintaint: A static taint analysis method for binary vulnerability mining[C]. *2018 International Conference on Cloud Computing*, *Big Data and Blockchain*, 2019: 1-8.

[24] IDA Pro, IDA Pro, https://www.hex-rays.com/, 2019.

[25] IDA Python, IDA Python, https://github.com/idapython/src, 2019.

[26] VEX IR, VEX IR, https://docs.angr.io/advanced-topics/ir, 2018.

[27] Shoshitaishvili Y, Wang R Y, Salls C, et al. SOK: (state of) the art of war: Offensive techniques in binary analysis[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 138-157.

[28] RECLASSIFY data set, RECLASSIFY, https://github.com/RECLASSIFY/RECLASSIFY-Data-Set, 2020.

[29] Debug Interface Access Software Development Kit, Microsoft, https://docs.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/debug-interface-access-sdk?view=vs-2015, 2016.

[30] Bounov D, Gökhan Kıcı R, Lerner S. Protecting C++ dynamic dispatch through VTable interleaving[C]. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016: 1-15.

[31] Dewey D, Giffin J T. Static detection of C++ vtable escape vulnerabilities in binary code[C]. *Proceedings 2012 Network and Distributed System Security Symposium*, 2012: 1-21.

[32] Dewey D, Reaves B, Traynor P, et al. Uncovering use-after-free conditions in compiled code[C]. *2015 10th International Conference on Availability*, *Reliability and Security*, 2015: 90-99.

[33] Gawlik R, Holz T. Towards automated integrity protection of C++ virtual function tables in binary programs[C]. *The 30th Annual Computer Security Applications Conference*, 2014: 396-405.

[34] Prakash A, Hu X C, Yin H. vfGuard: strict protection for virtual function calls in COTS C++ binaries[C]. *Proceedings 2015 Network and Distributed System Security Symposium*, 2015: 1-15.

[35] Sabanal, P.V., Yason, M.V., Reversing c++, Black Hat DC, 2007.

[36] van der Veen V, Göktas E, Contag M, et al. A tough call: Mitigat-

ing advanced code-reuse attacks at the binary level[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 934-953.

[37] Zhang C, Song C Y, Chen K Z, et al. VTint: protecting virtual function tables' integrity[C]. *Proceedings 2015 Network and Distributed System Security Symposium*, 2015: 1-15.

[38] I. Skochinsky, Practical c++ decompilation, Recon 2011, 2011.

杨晋 于 2017 年在河北大学网络工程专业获得学士学位。现在中国科学院信息工程研究所攻读硕士学位。研究领域为网络与软件安全。研究兴趣包括: 程序分析、漏洞挖掘与利用。Email: yangjin@iie.ac.cn

龚晓锐 于 2014 年 在北京大学获得硕士学位。现任中国科学院信息工程研究所正高级工程师。研究领域为攻防对抗。研究兴趣包括: 移动互联网安全、网络安全攻防对抗。Email: gongxiaorui@iie.ac.cn

吴炜 于 2014 年在中国科学技术大学信息安全专业获得学士学位。现在中国科学院信息工程研究所攻读博士学位。研究领域为网络与软件安全。研究兴趣包括: 漏洞挖掘与利用、程序分析及网络对抗。Email: wuwei@iie.ac.cn

张伯伦 于 2018 年在中国科学技术大学少年班学院获得信息安全专业学士学位。现在中国科学院信息工程研究所攻读博士学位。研究领域为网络与软件安全。研究兴趣包括: 漏洞挖掘与利用、程序分析、机器学习在安全领域的应用等。Email: zhangbolun@iie.ac.cn