分类号 TP313 学校代码 10590
U D C 004 密 级 公开

深圳大学硕士学位论文

二进制代码的类型恢复及其应用

学位申请人姓名 文 成
专 业 名 称 软件工程
学院（系、所） 计算机与软件学院
指 导 教 师 秦胜潮、许智武

深圳大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文二进制代码的类型恢复及其应用是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律后果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

本学位论文作者完全了解深圳大学关于收集、保存、使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属深圳大学。学校有权保留学位论文并向国家主管部门或其他机构送交论文的电子版和纸质版，允许论文被查阅和借阅。本人授权深圳大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（涉密学位论文在解密后适用本授权书）

论文作者签名：　　导师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘　要

二进制代码的类型恢复是一个非常具有挑战性的问题，这主要是由于高级语言在经过编译之后丢失了大量与类型相关的信息。然而，二进制代码的类型恢复又对理解和分析二进制代码起着重要的作用，并且它在反编译、逆向工程、漏洞分析及恶意代码检测等领域都有着重大的需求。因此，研究二进制代码的类型恢复问题就具有非常重要的现实意义。

目前，现有的二进制代码类型恢复的研究工作大多倾向于使用程序分析技术，但这种方法恢复出来类型信息往往比较保守，而过于保守的类型信息对理解和分析二进制代码的帮助有限，且这种方法在实际使用中的执行效率不高。为此，本文提出了一种新的方法来恢复二进制代码中变量的类型，该方法效率较高，能够在一定程度上恢复变量的数据类型信息。

首先，本文详细介绍了二进制代码的类型恢复方法，其主要原理来自“鸭子类型”系统：一个变量的类型是由它的行为和属性决定的。本文的方法融合了程序分析与机器学习方法。具体来说，本文先从二进制代码的指令流和数据流中提取出关键信息，这些信息反映了变量的行为和属性。然后根据提取出来关键信息，本文使用机器学习方法训练以基本类型为标签的分类器，训练完成后的分类器将被用于预测其它二进制文件中的数据类型信息。而对于指针和结构这样的复合类型，本文利用一个简单的指向分析，确定所有关联变量，利用分类器恢复所有关联变量的基本数据类型，并在此基础上再进行复合类型变量的类型恢复。

其次，本文还将二进制代码的类型恢复技术应用于恶意软件检测。本文的恶意软件检测方法同样基于分类器，但与目前许多现有的工作不同，本文不仅考虑了二进制代码的行为特征，还考虑到了二进制代码的数据类型特征。

最后，本文实现了一款针对二进制代码类型恢复的原型工具：BITY，并做了一系列实验来评估本文的方法。实验结果表明：（1）本文提出的二进制代码类型恢复方法能在一定程度上较准确地恢复出二进制程序中变量的数据类型；（2）无论是在精确类型还是在可兼容的类型上，BITY 都比商业工具 Hex-Ray 和开源工具 Snowman 要准确；（3）BITY 工具执行效率高，具有较强的可伸缩性，比较适合用于实际使用；（4）二进制代

码的数据类型信息对检测恶意软件有一定的帮助。

关键词：二进制代码分析；类型恢复；数据类型；机器学习；恶意软件检测

Abstract

Recovering type information in binary code is a great challenging problem due partly to the fact that much type-related information has been lost during the compilation from high-level source code. However, recovering type information in binary code helps a lot in comprehension and analysis of binary code. And it is required, or significantly benefits, many applications, such as decompilation, reverse engineering, vulnerabilities analysis and malware detection. Therefore, the research of the binary code type recovery has great practical significance.

Currently, most of the existing research on binary code type recovery tend to resort to program analysis techniques, which can be too conservative to infer types with high accuracy or too heavyweight to be viable in practice. In this paper, we propose a new approach to recovering type information for recovered variables in binary code, which is more precise and more efficient.

First, we present our approach to recovering type information in binary code. The idea is motivated by “duck typing”, where the type of a variable is determined by its features and properties. Our approach uses a combination of machine learning and program analysis. In detail, we first extract critical information form instruction-flow and data-flow, namely behaviors and features of variables in binary code. According to these behaviors and features, we learn a classifier with basic types as levels, using various machine learning methods, and then use this classifier to predict types for new, unseen binaries. For composite types, such as *pointer* and *struct*, we perform a point-to analysis to recover the target variables and use the classifier to recover the base type for these target variables, base on which, composite types are recovered.

Second, we also apply the type recovery technology of binary code to malware detection. Our malware detecting approach is based on classifier. Different from most existing work, we take into account not only the behavior information but also the data information. As far as we know, our approach is the first one to consider data types as features for malware detection.

At last, we have implemented our approach in a tool called *BITY* and used it to conduct a series of experiments to evaluate our approach. The results show that (1) our approach can precisely recover the type information in binary code; (2) our tool is more precise than the commercial tool Hey-Rays and the open source tool Snowman, both in terms of correct types and compatible types; (3) our prototype BITY is efficient and scalable, which is suitable in practice; (4) the type information we recover is capable of detecting malware.

Keywords：Binary Analysis；Type recovery；Data Type；Machine Learning；Malware Detection

目　　录

第 1 章 绪论

1.1 课题研究背景与意义

随着信息时代的发展，无论是在 PC 机上，还是在智能手机和嵌入式设备，应用程序的数量都在日益增长。虽然这些应用程序大部分是用高级语言编写的，但大多数软件都以二进制代码的形式发布，换句话说，绝大多数的应用程序只能获取到可执行文件，即其二进制代码形式，而获取不到高级语言层次的源代码。目前，很多工作都依赖二进制代码的分析，例如，在恶意软件的检测和分析过程中，二进制代码的逆向分析几乎是唯一的手段；在商业软件的安全审查和分析中，由于软件发布者不公开源代码，也只能审查其发布的二进制代码。

分析源代码和分析二进制代码不同，它们之间存在着巨大的差别，源代码中一行简单的代码在二进制代码中可能涉及多条指令以及大量的寄存器操作。最关键的是，编译器在编译的过程中，丢失了源代码中大量的高级语言信息。因此，相对于源代码的分析，二进制代码的分析更加复杂，具有很大的挑战。显然，在编译过程中丢失的高级语言信息非常有助于增强代码的可理解性，而其中一个非常重要的信息就是程序中变量的数据类型信息。程序中的变量是存储数据的所在处，它们有名字、值和数据类型。变量的数据类型决定了这些数据如何存储到计算机的内存中，这些数据有哪些操作方式，以及这些数据该怎么解释。因此，恢复出这些类型信息对于分析和理解二进制代码来说有着非常重要的意义。

从 20 世纪 90 年代末期，就开始出现二进制代码类型恢复的工具和应用，这些工具和应用被广泛用于各种领域中，如漏洞检测[1][2][3]、逆向工程[4][5]、反编译[6][7]、恶意代码检测[8][9]、二进制代码重用[10][11]、二进制代码重写[12][13]等等。在这些领域的应用中，二进制代码的类型恢复几乎已经成为其中一个不可或缺的步骤，例如，反编译技术从二进制代码恢复其高级语言信息，这包括表达式、类型和过程调用等，其中恢复变量的类型是反编译过程中一个非常重要的步骤；对于维护了很长时间的遗产软件，可能以前的开发文档不全，那么恢复其中函数的参数类型、返回值类型，能有助于遗产软件的再利用；在逆向工程中，恢复变量的数据类型和高层次的数据抽象有助于逆向工程师理解和分析二进制代码。如今，恶意软件和不受信任的代码与日俱增，复杂程度也愈来愈高，许多病毒、蠕虫、和木马已经对网络安全构成了巨大威胁，软件安全分析师对二进制代码分

析工具的需求已经越来越大。这进一步说明二进制代码的类型恢复有着十分重要的意义和迫切的需求。

1.2 国内外研究现状

关于二进制代码的类型恢复工作，其相关工作有着很长的历史积累，最早的文献可以追溯到 90 年代末。目前相关的研究人员不多，但也有不少卓有成效的研究，有许多成熟的技术已经应用到了实际使用中。在这些研究工作中，它们研究的侧重点，涉及到的领域，应用的方向大多各不相同。此处汇总了一些涉及到二进制代码的类型恢复的研究工作。

Balakrishnan 等人[14]根据 x86 可执行程序对内存的访问方式，将内存区域抽象地看成多块区域（区分为全局数据区，栈区，堆区），并且将程序中的每个函数所访问区域抽象地看作一块独立的区域。它们提出，可以使用一种值集分析（value-set analysis, VSA）算法，发现这些内存区域中的变量位置。

Lin 等人[15]的 Rewards 工具和 Slowinska 等人[2]的 Howard 工具都使用了动态分析的方法，执行给定的二进制文件，通过监视、捕获和分析程序执行时的动态信息，提取程序中数据结构的语法和语义，产生类型约束。该方法能够恢复程序中的数据结构。

JongHyuo 等人[16]使用了一种基于推理的类型推断方法，通过分析二进制代码的中间语言，为变量产生类型约束，最后再用一个约束求解器统一求解。他们还实现了一种新的基于类型重构理论的类型推理系统 TIE，可处理控制流，可应用于静态和动态的环境中。TIE 使用 BAP 转化二进制代码到二进制分析语言 BIL，使用 DVSA 算法分析内存的访问模式来确认变量类型，然后把 DVSA 恢复的变量传递到类型重构算法，最终生成变量类型。但是 TIE 给出的输出结果是一个上界和下界，并不是一个确切的类型，这种不准确的信息有时候对二进制工程师的帮助不大，并且 TIE 的算法成本过高，不一定有实效。

Balakrishnan 等人[14][17]还提出了一种将值集分析（value-set analysis, VSA）与聚合结构体识别（aggregate structure identification，ADI）相结合的算法 ASI，比 IDA 识别的 a-locs（abstract location）更为严格。ASI 是基于程序访问的内存模式来确定结构，需要假设数据访问模式在程序中是比较清晰的，但这在 x86 下并不成立。该算法以 VSA 恢复的信息作为输入，可恢复除显式的地址和偏移以外的间接引用内存的 a-locs，可识别结构体、数组以及数组与结构体的嵌套。但 VSA 方法开销过大，且 ASI 作为一个启发式

方法只在特定环境下适用，通用性不强。

Robbins 等人[18]将二进制代码的类型推断问题转为对一个有理树（rational-trees）的约束求解问题，最后再使用 SMT 求解器求解。

Yan 等人[19]研究的是二进制代码类型恢复中的一个子任务：确定一个整形变量是有符号的（signed）还是无符号的（unsigned）。他们提出了一种基于图的算法来解决该问题，即构建一个图，图中的每一个节点都对应程序中的一个整形变量，图中的连线代表两个整形变量间具有相同的类型。通过计算图的“最小割”，可以计算出一个合理的类型推断结果。

Raychev 等人[20][21]提出了一种新的方法，通过从大量的代码库程序（big code）中学习概率模型，能使用这个模型来预测程序的一些属性，其中就包括了变量的类型信息。该方法需要根据程序的结构，构建一张图，图中的“节点”代表程序中的变量，图中的“边”体现出变量之间的依赖性和约束。不过，该工作分析的是高级语言层次的源代码，尽管该方法分析源代码时非常有效，但由于在二进制代码中丢失的信息很多，又需要考虑大量的寄存器变量，该方法对于分析二进制代码的适用性不强。

Katz 等人[22]提出，使用对象踪迹（object tracelets）可以描述一个 C++中“类对象”的特征，同样，对象踪迹可以描述一种类型所具有的普遍性特征。那么，只要求解变量的踪迹与哪个类型的踪迹最相似，就可以预测出变量最可能的几种类型。该方法需要程序中部分变量的类型是已知的，用于训练各个类型的踪迹，因此该方法对于分析无调试信息的二进制程序来说适用性不强，一般多用于遗产软件的设计或用于理解软件系统的结构

Zheng 等人[23]使用神经网络来恢复二进制代码中函数的参数和返回值，该工作将二进制代码作为训练数据，训练神经网络用于恢复函数的参数和返回值。该工作将恢复函数的参数个数和参数的数据类型看成两个独立的子问题，并为两个子问题分别训练神经网络。实验表明，该研究工作训练的神经网络对恢复函数参数个数的准确率能达到 84%，对恢复参数类型的准确率能达到 81%。此外，该工作还考虑到的编译器的各种优化（O0-O3），支持 x86 和 x64 两种架构。

MemPick[24]是一个用于恢复程序中高级数据结构的工具，例如单链表、双链表以及各种类型的树（AVL 树、红黑树、B 树）；Retypd[25]工具支持更多复杂类型的恢复，例如递归类型；ARTISTE[25]工具结合了循环不变式和指向分析方法，能动态地检测恢复二进制代码中的数据类型；PointerScope[27]通过观测指针是否被误用，来检测恶意代码；

SecondWrite[28]结合了指向分析和 VSA 方法，能为 x86 架构的可执行文件恢复其中的函数原型和数据类型，还能产生一种中间语言（IR）用于分析或重写二进制代码；SmartDec[29]是一款反编译工具，可以恢复堆区中的结构体；IDA Pro 是一个世界顶级的交互式反汇编工具，其中的 Hex-ray 插件，用于将汇编层次的代码反编译为可读性更强的 C 语言，但由于它是商业软件，其具体方法并未公开。

另外，Caballero 等人也整理和对比了从 1999 年至 2016 年的相关工作，感兴趣的读者可以详细阅读[30]。

二进制代码的类型恢复还面临着很多挑战，目前存在着许多值得研究的问题。主要有以下几点：一、根据 Caballero 等人的调研报告，目前支持二进制代码类型恢复的相关工具不多，在目前现有的一些研究工作中还没有一项研究工作支持恢复所有的数据类型，多数研究工作恢复的数据类型侧重点各不相同。例如一些漏洞分析的工作侧重于指针类型的恢复；二进制代码重用的工作只关心恢复函数的参数类型和返回值类型；目前很多反编译工具准确率不高，反编译出来的高级语言无法通过类型检查；还有一些工作只关注更高层次的数据结构的恢复。二、目前许多工具推测的类型过于保守。大多数时候，精确的类型信息对于理解和分析二进制代码的帮助更大，而过于保守的类型信息意义不大。三、目前许多工具使用的方法成本较高、效率较低。许多研究工作倾向于使用约束求解、符号执行这类较复杂的程序分析方法，并不适用于分析规模较大的程序。例如 DIVINE[17]分析一个 55000 行汇编代码的程序用了两个小时。其中一个原因是因为低级语言代码中的指令比高级语言代码的语句要多很多。另一个原因是对于同一个指令有很多不同的使用方式，需要尽可能考虑所有情况则需要产生很多约束。

总体来说，目前关于二进制代码的类型恢复涉及到的领域多，应用方向广泛。尽管目前已经有一些非常有价值的研究成果，但也还存在一些问题，有很多问题值得去研究和探索。

1.3 本文的研究动机

第 1.2 节介绍了目前二进制代码的类型恢复技术所面临的一些挑战，这些挑战都是在当前的研究工作中值得去克服的。本节以两个简单的实例进一步说明现有的一些工具仍存在的问题，并解释本文的研究动机。

实例一来自 GNU 软件包 *coreutils (v8.4)*中的 *base64* 程序，图 1-1 左侧的代码片段是通过对 *base64* 的二进制程序进行反汇编得到的，为了方便理解和解释说明，图的右侧附

上了对应的 C 语言源代码片段。在右图的 C 语言程序中，decode 变量是一个布尔型变量，其功能是用于记录用户的选项，然而，编译后 decode 变量体现为栈中的一个字节，且其偏移量为-1，该变量在左图中表示为 [ebp − 1] ，其布尔类型的信息已经丢失。大多数支持类型恢复的工具都采用一种比较保守程序分析方法，Hex-Rays 对该变量的类型恢复结果为 char 类型，与源代码中显式定义的 bool 类型不一致。bool 类型的变量长度为 8 位，仅有 true 和 false 两种值，只能进行异或操作，而 char 类型的变量长度为 8 位，能进行+ - * / %和字符操作。尽管此处将该变量推断为 char 类型也不影响该程序的语义，但类型恢复旨在恢复出源代码中开发人员显式定义的类型，故 Hex-Rays 的结果错误。而另一个开源工具 SmartDec 将该变量的类型推断为 byte_t (一个字节的变量)，尽管该推断是正确，但并未具体指出该类型是 byte_t 的哪一个子类型，本文将这种推断称为过于保守的类型恢复。

```
        mov     byte ptr [ebp-1], 0
        cmp     dword ptr [ebp-8], 64h
        jz      short loc_40101B
        jmp     short loc_40101F
loc_40101B:
        mov     byte ptr [ebp-1], 1
loc_40101F:
        movzx   eax, byte ptr [ebp-1]
        test    eax, eax
        jz short loc_401035
        call    do_decode
loc_401035:
        retn
```

```
int main( )
{
    bool decode =   false;
    int opt = getopt;
    switch (opt){
        case ‘d’:
            decode = true;
            break;
        default:
            break
    }
    if (decode)
        do_decode;
}
```

图 1-1 *base64* 程序的代码片段

实例二是对三个不同类型的指针变量进行赋值，如图 1-2 所示，图的左侧是程序的汇编代码片段，图的右侧是对应的 C 语言源代码。在指令非常少的情况下，推断出正确的类型显得更困难。SmartDec 推断这 3 个变量的类型都是整数，这显然是错误的。而 Hex-Rays 推断变量 i 和 f 的类型是 Dword*，d 的类型是 Qword*。虽然 Hex-Rays 能推断出该变量的类型为指针，但是该结果相对于 int*, float*, double* 来说就显得不够精确，其推断结果也是存在过于保守的问题。

可以注意到，对于不同类型的变量，在编译之后对应的指令也有所不同。例如实例二中的 int，float 和 double 类型的变量对应的赋值指令分别是 mov，movss 和 movsd。因此，一个简单的解决方案就是针对这三个不同的指令，设定三个规则，分别用来推断这三种的类型。然而，就 mov 指令来说，mov 指令不仅仅会被 int 类型的变量使用，还可能被指针类型的变量使用，这就对同一个指令，需要产生不同含义的约束。尽管这个方法有效，但是指令集太过庞大，对于每种指令都要产生许多不同的约束也十分困难。例如前缀为 mov 的操作码就超过 30 种，并且不同的源操作数和目标操作数也会让指令有不同的含义。

	int* i;
mov [i], 0Ah	*i = 10;
movss xmm0, ds:XX	float* f;
movss [f], xmm0	*f = 10.0;
movsd xmm0, ds:XX	double* d;
movsd [d], xmm0	*d = 10.0;

图 1-2 例：给不同类型的变量赋值

一般来说，指令通常可以分为操作数和操作码这两个部分。操作数指出了参与运算的数据来源和运算结果所存放的位置等；而操作码能够说明该指令所要完成的动作，反映的是指令的功能。程序中的变量往往以操作数的形式出现，与之相关的操作码正是体现了变量的行为特征。因此与变量相关的指令中，操作码与操作数的组成形式可以反映变量是如何被存储、操作和解释的。正如实例二中表现的那样，mov ...、movss ...、movsd ...这条指定是针对不同变量的。受启发于“鸭子类型”系统，推断变量的类型，不是根据其显示定义的名称，而是根据它的行为和属性。本文将与变量相关的指令表示形式作为特征，并从中学习知识来预测其它二进制文件中的数据类型信息。

1.4 本文的主要研究内容

本文的主要研究内容是，对于一个给定的二进制可执行文件，恢复该程序的数据类型信息，恢复的类型要尽可能接近源代码中开发人员显式定义的类型。为此，本文提出了一种新的方法来恢复出二进制代码中变量的数据类型。与现有的一些工作不同，本文没有采用像约束求解那样的分析技术，而是融合了程序分析与机器学习方法，先从二进

制代码的指令流和数据流中提取出关键信息，再利用这些关键信息来训练分类器，最后根据分类器的预测结果整合得出变量的类型。

本文二进制代码类型恢复方法的主要原理来自于“鸭子类型”系统：一个对象的类型是由它的行为和属性决定的。在“鸭子类型”系统中，如果一只鸟走起来像鸭子、游起泳来像鸭子、叫起来也像鸭子，那么它就可以被当作鸭子。也就是说，“鸭子类型”系统并不关注对象本来叫什么名字，而是关注对象具有什么样的行为和属性。在二进制代码中，代码段可以表示为机器指令，而机器指令最能体现二进制代码的行为特征，它表明了二进制代码是用来做什么和怎么做的。机器指令是用二进制代码表示的，能被计算机直接执行，它通常由操作码和操作数两部分组成。操作码表示该指令应该进行何种操作，即指令的功能。而操作数是操作码作用的对象，它指出了操作所需要的数据来源，以及运算结果所存放的位置等。程序中的变量往往以操作数的形式出现，与之相关的操作码正是体现了变量的行为特征。因此，若一个变量的行为与指针类似，那么就可以推断这个变量的类型为指针。

本文的方法融合了程序分析方法和机器学习方法，首先利用一些程序分析技术，从二进制代码中提取出一些变量特有的行为和特征，然后根据些行为和特征，利用机器学习的方法训练以基本类型为标签（label）的分类器，训练完成后的分类器将被用于预测其它二进制文件中的类型信息。具体来说，本文首先收集了一些二进制文件的样本和相应的调试信息作为数据集，其中二进制文件的样本应涉及到各种数据类型的使用，而调试信息包含了各个变量的类型信息。其次，为数据集中每个样本进行二进制代码静态分析。在经过反汇编过程之后，为其恢复目标变量。根据 DU 链和 UD 链提取出与目标变量相关的指令以及目标变量的一些基本信息（例如，变量占内存空间的大小）。再者，基于这些恢复的目标变量及其相关信息，本文训练了一个以基本类型的标签的分类器。最后，本文使用这个训练好的分类器，对未知类型变量预测出其最有可能的基本类型。而对于指针和结构等这一类复合类型，本文利用一个简单的指向分析先确定所有关联变量，和利用分类器恢复所有关联变量的基本数据类型，并在此基础上再进行复合类型变量的类型恢复。

本文将上述方法实现为一个工具，称为 BITY。此工具由 Python 语言编写，其中，IDA pro 作为前端，用于解析二进制文件，将二进制代码转换为可读性更强的汇编指令，分类器是使用 Scikit-learn 实现的。本文设计了一系列实验来测评 BITY 工具。首先，使用不同的机器学习方法训练分类器，就二进制代码类型预测这个问题上，对不同的机器

学习方法的效果进行对比和总结。其次，使用 *coreutils (v8.4)*作为基准，分别以“准确率”和“距离”这两个评价指标，将 BITY 与商业工具 Hex-Rays 和开源工具 Snowman 进行对比实验，实验表明 BITY 在精确类型和可兼容的类型上都比 Hex-Rays 和 Snowman 表现得好。最后，使用不同大小的二进制文件（从 10KB 到 1.3GB）对 BITY 进行性能评测，实验表明此工具的执行效率高，比较容易付诸于实际应用当中。

最后，本文还将二进制代码的类型信息应用于恶意软件检。本文将二进制代码的类型信息作到恶意软件检测的重要特征之一，不仅增加了特征的多样性，还提高了反恶意软件的能力。

本文的主要贡献总结如下：

1. 本文提出了一种新的方法，融合了程序分析方法与机器学习方法，能一定程度上恢复二进制代码中变量最可能属于的类型。本文的方法支持基本类型的恢复，同时也支持部分复合类型的恢复。
2. 本文实现了一个原型工具 BITY，并做了一系列实验来评估本文的方法。实验表明，本文提出的方法在一定程度上能够较准确地恢复出二进制程序中变量的类型。无论是在精确类型上还是在可兼容的类型上，本文实现的原型工具 BITY 都比商业工具 Hex-Rays 和开源工具 Snowman 要准确，并且该原型工具在分析实际应用时的效率较高，具有较强的可扩展性，适合于实际使用。
3. 本文将二进制代码的类型信息应用于恶意软件的检测，并实现了一款能检测恶意软件的原型工具。实验表明，类型信息对检测恶意软件有一定的帮助，能够增加特征的多样性，提高反恶意软件的能力。

1.5 本文的组织结构

本文共分为六章：

第一章为绪论，首先介绍了二进制代码的类型恢复的背景和意义，紧接着说明了目前的研究工作已取得的成果和存在的挑战，然后以两个简单的例子进一步解释本文的研究动机与目前的大多数工具还存在的一些不足，最后描述本文的主要研究内容。

第二章为相关理论知识，介绍论文课题涉及到的理论和技术，并对这些理论和技术作解释说明。

第三章详细介绍了二进制代码类型恢复方法。首先介绍了该方法的整体流程。并分别详细描述了本文方法中的四个主要任务：1. 二进制代码分析；2. 分类器的训练；3. 基

本类型的预测；4. 复合类型的恢复。

第四章介绍了二进制代码的类型恢复在恶意软件检测中的应用，首先介绍了恶意软件检测的背景，然后提出本文的恶意软件检测方法，并详细介绍了特征提取器和分类器这两个组件。

第五章是实验与结果评价，设计了一系列实验来评估本文的原型工具，并将本文的原型工具与商业工具 Hex-Rays 和开源工具 Snowman 进行对比，就准确率和距离这两个指标进行评价。本章的另一个实验是使用不同大小的二进制文件对本文的原型工具进行性能评测。另外，本章还展示了第四章中恶意软件的检测部分的详细实验与评价。

第六章是总结与展望，总结了本文的研究工以及还存在的局限性，并对未来的研究工作进行展望。

第 2 章 相关理论知识

2.1 二进制代码与汇编指令

二进制代码是由两个基本字符‘0’和‘1’组成的代码，二进制代码的特点是计算机可以直接识别，不需要进行任何翻译，但由于其书面形式全是“密”码，可读性非常差，不便于人类理解与分析。二进制代码可以分为二进制指令和二进制数据。

分析二进制代码，大多都将其转换为等价的、可读性更强的 IR（中间代码，intermediate representation）。汇编代码是最为接近二进制代码的 IR，很多工作都会选择汇编代码作为分析二进制代码的 IR。汇编代码同样分为汇编指令和数据。汇编指令与二进制代码指令具有对应关系，一条汇编指令必将唯一对应一条二进制代码指令，这就使得汇编指令与二进制代码指令的相互转化成为可能。将二进制代码转化为汇编代码的过程称作反汇编。反汇编的主要过程是解析可执行文件格式，提取其中代码块，按一定策略将代码区域中的二进制指令映射为汇编指令。它的核心工作就是要能够解析二进制代码，根据 CPU 的指令集规范理解它的含义，并且将其展现为对应的文本汇编形式。目前反汇编技术已经比较成熟，有许多反汇编工具可以完成此任务，常见的反汇编工具有：IDA Pro、Sourcer、Capstone 和 OllyDbg 等。

汇编指令可以看作是一个由操作码（Opcode）和操作数（Operand）组合而成的表达式，就指令与操作数的关系而言，指令无非是无操作数、单操作数、双操作数和三操作数。所谓的操作码就是唯一代表着指令意义的一段二进制代码，它是指令操作功能的记述，用来告诉 CPU 需要做什么。每一条指令都有一个操作码，它表示该指令应该进行何种性质的操作。操作数是操作码作用的对象，它指出了操作所需要的数据来源。通常一条指令均包含操作码和操作数，例如指令 cmp eax, ebx 是一个双操作数指令，cmp 是该指令的操作码，指定程序比较两个操作数的大小，寄存器 eax 和寄存器 ebx 是该指令的两个操作数，指定了 cmp 操作的数据来源。就指令的操作数而言，可以是立即数、寄存器或者内存地址。

对于变量恢复与类型恢复问题，变量在操作数中通常以内存地址的形式出现，或者是值存放于寄存器中，而与之有关的操作码则说明了对数据进行何种操作。无论是与变量相关的操作码，还是操作数的个数与存放数据的寄存器，都对变量有一定的约束，能反映出变量所属的数据类型。

2.2 类型系统

类型系统是在计算机科学中，用于定义如何将编程语言中的数值和表达式归类为许多不同的类型，如何操作这些类型，这些类型如何相互作用。类型可以确认一个值或者一组值具有特定的意义和目的（虽然某些类型，如抽象类型和函数类型，在程序运行中，可能不表示为值）。类型系统在各种语言之间有非常大的不同，最主要的差异存在与编译时期的语法，以及运行时期的操作实现方式

在程序设计语言中，常常借用类型来排除一大类语义错误。类型之所以能够起到这个作用，是因为它包含了一定的语义信息。程序设计师用程序设计语言表示和解决应用问题的过程，而分类是人类的一种主要思维方式，有了类型，人们就可以在程序设计中方便地运用这种思维方式，因而，程序设计语言的类型系统被作为表达应用领域概念和概念间关系的有效工具。

类型指派（Typing）是赋予一组比特（bit）某个意义。类型通常和存储器中的数值或对象相联系。因为在电脑中，任何数值都是以一组比特简单组成的，硬件无法区分存储器地址、脚本、字符、整数、以及浮点数。类型可以告知程序和程序设计者，应该怎么对待那些比特。

类型系统提供的主要功能有：

1. 安全性：使用类型可允许编译器侦测无意义的，或者是可能无效的代码。例如，可以识出一个无效的表达式“Hello, world”+3，因为不能对（在平常的直觉中）逐字字符串上加上一个整数。强类型提供更多的安全性，但它并不能保证绝对安全。
2. 优化：静态类型检查可提供有用的信息给编译器。例如若一个类型指明某个值必须以 4 的倍数对齐，编译器就有可以使用更具有效率的机器指令。
3. 可读性：在更具有表现力的类型系统中，若其可以阐明程序设计者的意图的话，类型就可以充当为一种文件形式。例如，时间戳记可以是整数的子类型；但如果程序设计者声明一个函数为返回一个时间戳记，而不只是一个整数，这个函数就能表现出一部分文件的阐释性。
4. 抽象化：类型允许程序设计者对程序以较高层次的方式思考，而不是烦人的低层次实现。例如，程序设计者可以将字符串想成一个值，以此取代仅仅是字节的数组。或者类型允许程序设计者表达两个子系统之间的接口。将子系统间交互时的必要定义加以定位，防止子系统间的通信发生冲突。

在每一个编程语言中，都有一个特定的类型系统，保证程序的表现良好，并且排除违规的行为。作用系统对类型系统提供更多细微的控制。

2.3 子类型关系

子类型关系 <: 是一种类型变量间的关系，S <: T 可读做“S 是 T 的一个子类型”（或“T 是 S 的父类型”），这种子类型与从属关系类似，若 A <: B，则 A 类型的数值也可以是 B 类型的。

子类型关系 <: 是一种偏序关系，满足自反性、反对称性和传递性：

$$S <: S \tag{2-1}$$

$$\frac{S <: T \quad T <: S}{S = T} \tag{2-2}$$

$$\frac{S <: U \quad U <: T}{S <: T} \tag{2-3}$$

可以找到一个所有类型的超类型。这里引入一个新的类型常量 Top（T），加上一条规则使 Top 为子类型关系的最大元素。

$$S <: Top \tag{2-4}$$

同样也存在含最小元素的子类型关系。这里引入一个新的类型常量 Bot（⊥），它是所有类型的子类型。

$$Bot <: T \tag{2-5}$$

类型推断与这里的子类型关系之间的桥梁可以通过一个称为“包含”的新类型规则建立起来。式$2-6$中，一个类型化上下文（也称类型环境）Γ 是一个变量和它们类型的序列，t: S 表示自由变量 t 的类型为 S。这个规则表明，如果 S <: T，那么类型为 S 的元素也可以是 T 类型的：

$$\frac{\Gamma \vdash t: S \quad S <: T}{\Gamma \vdash t: T} \tag{2-6}$$

子类型关系在各种语言之间有非常大的不同，以 C 语言中的类型为例，有子类型关系 bool <: short <: int <: long long int，而在其它编程语言中，类型的种类和名称、各类型之间关系则不同。本文关于二进制代码中的子类型关系详见 3.2 节，这种子类型关系还是类型恢复的结果评判中的重要标准，也是后文中定义距离函数 $d(s,t)$ 的依据。

2.4 类型恢复

在大多数高级语言中，变量在定义的时候会声明其数据类型，变量的类型决定了需

要为变量分配多大的存储空间，也决定了变量的使用规则等，例如 int 类型的变量占两个字节，能够进行四则运算等，这样的信息非常有助于整个程序的理解。而问题是，在二进制代码中，并不体现变量的声明语句，也不会显式地给变量标注数据类型。高级语言代码在经过编译后，变量的数据类型声明语句不会产生相应的代码，编译器只是按照这些变量声明语句提供的信息，将存储空间分配给变量，建立符号表，并将目标代码中对变量符号的访问转换为相应存储单元的访问。在编译结束之后，符号表一般不保留在二进制代码中，这样对数据类型的恢复就只有依赖于变量存储单元的分配形式及其使用方式，也需要从目标代码的指令流中提取一些隐式信息，结合变量的行为和属性去推断其类型。另外，在高级语言代码中一般会定义复杂变量类型，如多级指针，结构等，如果要恢复这些变量的类型，那就需要整合变量信息以形成高级语言代码中复杂变量类型，这就是复合类型的恢复问题。

二进制代码的类型恢复的主要工作是，对于一个给定的二进制可执行文件，恢复该程序的类型信息，其类型要尽可能的接近源代码中开发人员显式定义的类型。一个可执行文件包含数据和代码，它们都强烈地影响程序行为，并且可以成为二进制代码类型恢复的目标。由于目前已经有许多工具可以区分数据和代码，二进制程序的类型恢复主要处理的是存储程序数据的类型变量。

二进制代码的类型恢复主要包括两个任务：变量恢复和类型恢复。变量恢复旨在从低级语言中识别出高级语言中对应的变量，例如函数接受几个参数、函数有哪些局部变量。类型恢复旨在为每一个变量提供一个高级语言层次的类型，例如某个函数的接受何种类型的参数、返回何种类型的变量。

二进制代码的类型恢复受编程语言，编译器，操作系统和指令集架构的影响。编程语言定义了内置的类型；编译器选择应用二进制接口（ABI），它包括类型表示、数据结构对齐，函数调用约定和文件格式（PE/ELF）等；指令集架构也会影响数据的表示。但其提出的方法基本是通用的，大多数方法都是使用特定的编程语言和指令集架构来评估的。二进制代码的类型恢复最常见的目标平台是 x86，其次是 x86-64。

2.5 DU 链、UD 链和静态单赋值形式

“定义-使用链”（define-use chain，简记为 DU 链），是指由一个定义，一个变量和对它所有的使用组成的数据结构，并且这个变量不经过中间定义就能到达使用处。与之相对的是“使用-定义链”（use-define chain，简记为 UD 链），它是指由一个使用，一个

变量和它的所有的定义组成的一个数据结构。DU 链和 UD 链通过使用静态代码分析中的数据流分析创建的，是变量分析的一个步骤，它的作用是使所有的变量逻辑表示可以被确定并通过代码进行跟踪。它在编译优化中的运用较多，如常数传播，公共子表达式消除等。

以图 2-1 中的代码为例，在标记 A，B，C 三处，变量 x 被赋予不同的值。在标记 1 处，对于 x 的 UD 链指出此时的值来自标志 B 处，同理，在标记 2 处，对于 x 的 UD 链指出此时的值来自标志 C 处。因为在标志 2 处的值不依赖于标志 1 处或更早的定义，这样的话，不妨将其设置成不同的变量，如图 2-2 的代码，这个过程叫作有效范围分割。主要的目的是将程序转换为静态单赋值形式（Static Single Assignment，简写为 SSA）。SSA 是一种中间表示，如果在某个过程内赋值的每一个变量作为赋值目标只出现一次，就将这个过程叫单赋值形式。它能有效地将程序中的运算值和它们的存储值分开，从而使得若干优化具有更有效的形式。

```
int x = 0;        //A
x = x + y;        //B
// 1.some uses of x
x = 35;           //C
// 2.some more uses of x
```

图 2-1 示例代码

```
int x_0 = 0;          //A
x_1 = x_0 + y;        //B
// 1.some uses of x
x_2 = 35;             //C
// 2.some more uses of x
```

图 2-2 有效分割

静态单赋值形式（SSA）是 IR（中间代码，Intermediate representation）的特性，它能有效地将程序中的运算值和它们的存储位置分开，从而使得若干优化能具有更有效的形式。在原始的 IR 中，已存在的变量可被分割成许多不同的版本，在许多教科书当中通常会在旧的变量名称加上一个下标而成为新的变量名称，用于标明每个变量及其不同版本。在用 SSA 形式表示的过程中，DU 链和 UD 链是非常明确的，变量的使用会用到

一个特定定义产生的值，当且仅当该过程的 SSA 形式中此变量的定义和使用具有完全相同的名字。

SSA 形式可以保证每个被使用的变量都有唯一的定义，即 SSA 能带来精确的使用-定义关系。概括起来，SSA 带来四大益处：

1. 因为 SSA 使得每个变量都有唯一的定义，因此数据流分析和优化算法可以更加简单。
2. 使用和定义关系所消耗的空间从指数增长降低为线性增长。若一个变量有 N 个使用和 M 个定义，若不采用 SSA，则存在 M×N 个使用-定义关系。
3. SSA 中因为使用和定义的关系更加的精确，能简化构建干扰图的算法。
4. 源程序中对同一个变量的不相关的若干次使用，在 SSA 形式中会转变成对不同变量的使用，因此能消除很多不必要的依赖关系。

第 3 章 二进制代码的类型恢复方法

3.1 整体流程

本章详细介绍了二进制代码的类型恢复方法。简单地说，该方法以二进制代码作为输入，通过反汇编过程将二进制代码转化为汇编代码，基于得到的汇编代码进行静态分析，恢复程序中的目标变量，并最大限度地从指令流和数据流中提取隐式的信息。预先训练好的分类器将根据提取出来的信息预测其它目标变量的基本类型。最后整合分类器给出的结果，进行复合类型的恢复，并输出二进制代码中的类型信息。

二进制代码的类型恢复方法的整体流程如图 3-1 所示，具体来说，本文首先收集了一些二进制文件的样本和相应的调试信息作为数据集，其中二进制文件的样本应涉及到各种数据类型的使用，而调试信息包含了各个变量的类型信息。其次，为数据集中每个样本进行二进制代码分析。在经过反汇编过程之后，为其恢复目标变量。根据 DU 链和 UD 链提取出与目标变量相关的指令以及目标变量的一些基本信息（例如，变量占内存空间的大小）。再者，基于这些恢复的目标变量及其相关信息，本文训练了一个以基本类型为标签的分类器。最后，本文使用这个训练好的分类器，对未知类型变量预测出其最有可能的基本类型。而对于指针和结构等这一类复合类型，本文利用一个简单的指向分析先确定所有关联变量，和利用分类器恢复所有关联变量的基本数据类型的信息，并在此基础上再进行复合类型变量的类型恢复。

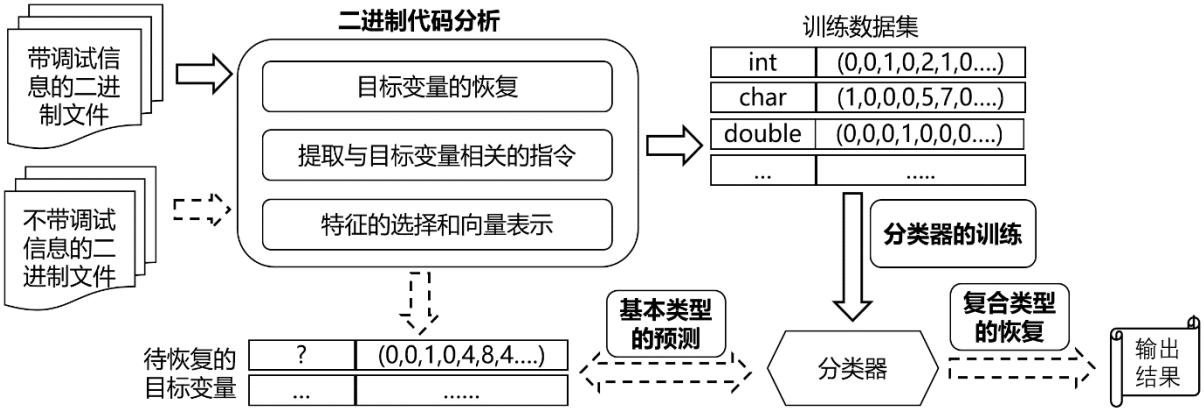

图 3-1 二进制代码的类型恢复方法

具体的来说，二进制代码的类型恢复方法主要有四个任务：1. 二进制代码分析；2. 分类器的训练；3. 基本类型的预测；4. 复合类型的恢复。第 3.2 节将介绍类型的格，从

第 3.3 节开始会分别详细说明本文方法的四个主要任务。

3.2 类型格

本文训练分类器时使用的标签（label）都是一些基本类型，数据集中变量的基本类型都可以从调试信息中提取，用 L 表示这些基本类型的集合:

$$L = \{bool, char, short, float, int, pointer, long\ long\ int, double, long\ doube\} \quad (3-1)$$

选用这些类型作为训练标签，其主要原因是这些类型已经足够表示一个变量的数据类型，而更复杂的类型（结构，指针，多级指针）可以由这些基本类型整合而成。

图 3-2 为本文使用的类型格（Lattice），⊤是子类型关系的最大元素，表示变量可以是任何类型，⊥是所有类型的子类型，表示变量不能是格中的任何类型（矛盾）。特别的，在指针类型之下还指向这个格本身，即这是一个多层次的格，这就使得这个格能够覆盖多级指针的类型（详见第 3.6.1 节，指针类型的恢复）。这个类型格描述了类型的层次结构和不同类型之间的距离，因而能够用于测评方法的精度（详见第 5.2 节）。类似 TIE[16]，本文的方法考虑到不同类型的变量占存储空间的大小，占存储空间大小不同的类型之间是没有子类型关系的。例如，short 不是 int 的子类型，int 不是 double 的子类型（对于 32 位 C 类型，short 占 2 个字节，int 占 4 个字节，double 占 8 个字节）。

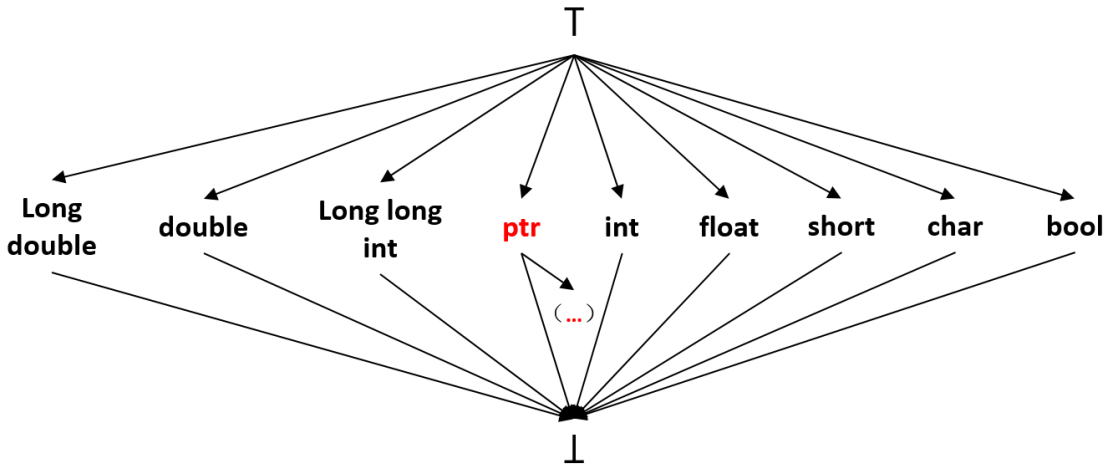

图 3-2 类型的格

3.3 二进制代码分析

本文使用汇编指令作为二进制代码的中间表示形式，汇编指令可以通过任何反汇编器获得，在本文中使用的反汇编器是 IDA Pro。本文只选取了英特尔平台上的 x86 架构的二进制程序作为研究对象，本文的方法对于其它架构的二进制程序同样适用。

高级语言的源代码在经过编译和链接后，源代码中的变量名及数据类型都不会出现在二进制代码中。因此，第一个步骤就是找出二进制代码中的目标变量。根据“鸭子类型”系统的原理，变量的类型是由其行为特征和属性决定的，所以第二个步骤就是提取目标变量的行为特征，即提取与目标变量相关的指令，这些指令经过抽象和筛选后，将代表变量的特征；最后根据这些特征，将变量抽象为特征空间中的一个向量。因此，二进制代码分析应包括三个步骤：（1）目标变量的恢复；（2）提取与目标变量相关的指令；（3）特征的选择和向量表示。本节的余下内容将详细阐述这三个步骤。

本文以一个 C 语言运行库中的程序 *memchr* 作为例子来解释本文的方法，该程序对应的汇编代码和 C 语言源代码如图 3-3 和图 3-4 所示。

```
%% ASM Code Snippet
        ……
        loc_401009 :
07      cmp   dword ptr [ ebp +10h ] , 0
08      jz      short loc_40103A
09      mov   eax , [ ebp +8]                  % eax^0
10      movsx    ecx , byte ptr [ eax ]        % ecx^0, eax^0
11      mov   [ ebp−44h ] , ecx                % ecx^0
12      mov   edx , [ ebp+0Ch ]                % edx^0
13      mov   [ ebp−48h ] , edx                % edx^0
14      mov   eax , [ ebp +8]                  % eax^1
15      add    eax , 1                         % eax^2
16      mov   [ ebp +8 ] , eax                 % eax^2
17      mov   ecx , [ ebp−44h ]                % ecx^1
18      cmp   ecx , [ ebp−48h ]                % ecx^1
19      jz      short loc_40103A
20      mov   eax , [ ebp +10h ]               % eax^3
21      sub    eax , 1                         % eax^4
22      mov   [ ebp +10h ] , eax               % eax^4
23      jmp   short loc 401009
        loc_40103A :
24      cmp   dword ptr [ ebp +10h ] , 0
25      jz    short loc_401051
26      mov   eax , [ ebp +8]                  % eax^5
```

```
27    sub   eax , 1                    % eax^6
28    mov  [ ebp +8 ] , eax            % eax^6
29    mov  ecx , [ ebp +8]             % ecx^2
30    mov  [ ebp−44h ] , ecx           % eax^2
31    jmp   short loc_401058
      loc_401051 :
32    mov  dword ptr [ ebp−44h ] , 0
      loc_401058 :
33    mov  eax , [ ebp−44h ]           % eax^7
      ……
```

图 3-3 *memchr* 对应的汇编代码

```
%% C Code
char *memchr (char *buf, int chr, int cnt)
{
      While (cnt && *buf++ != chr)
            cnt--;
      return (cnt ? --buf : NULL)
}
```

图 3-4 *memchr* 对应的 C 语言源代码

3.3.1 目标变量的恢复

在二进制代码中，需要恢复的目标变量包括全局变量，栈区中的变量，以及堆区中的变量。变量是内存块的抽象，访问这些变量主要通过“直接寻址”和“间接寻址”的方式。直接寻址的特点为在指令中直接给出操作数所在内存的地址，而间接寻址是相对于直接寻址而言的，寻址得到的数据是一个地址，需要通过访问这个地址来找到最终的数据。在二进制代码中，全局变量一般是用直接寻址的方式访问的，因此直接以内存地址表示的操作数都可以看作全局变量。而栈中的变量和堆区中的变量都是使用间接寻址的方式访问的，被访问变量有一个固定的表达形式，一般表示为“[base + index × scale + offset]”。其中，base 和 index 是寄存器，scale 和 offset 是一个整型常量。因此，恢复栈区和堆区中的变量就是识别这种表达形式的内存块。G. Balakrishnan 等人[14]最早提出可以使用集值分析（VSA）的方法来恢复二进制代码中的目标变量。本文的方法类似 VSA，将内存区域抽象地看成多块区域（区分为全局数据区，栈区，堆区），并且将程序中的每个函数所访问区域抽象地看作一块独立的区域（每个函数都可以看作有各自独立的栈区

域，局部变量按一定的顺序分布在栈区域中）。从全局数据区中恢复全局变量，从各个函数独自的栈区域中恢复局部变量，若全局数据区或栈区中有指向堆区的指针，则通过指针的值及其偏移量来恢复堆区中的变量。由于全局变量使用直接全局数据区的地址访问，较容易恢复，而函数中局部变量使用间接寻址的方法访问，其中变量的恢复比较复杂，本文将重点描述函数中局部变量的恢复。

每个函数被调用的时候，其所访问的区域都可以抽象地看作一块独立的区域，该区域的活动由栈帧记录，栈帧是记录在栈上面的，记录了该函数的调用信息，包括函数的参数、函数的局部变量、函数执行完后返回的位置、以及上一个栈帧的栈底指针等信息等等，栈帧结构如图 3-5 所示。函数被调用时，调用者依次把参数压人栈帧，然后调用函数，函数被调用以后，在栈帧中取得数据，并进行计算。函数计算结束以后，堆栈恢复原状。需要注意的是，对于不同的函数调用方式，参数的传递方式是不同的，例如在 __cdecl 和 __stdcall 调用约定中，参数是从右到左依次压栈的，而在 __fastcall 调用约定中，左边开始的两个不大于 4 字节的参数分别放在 ecx 和 edx 寄存器中，其余的参数依旧自右往左压栈传送。但不管是哪一种函数调用方式，函数的返回值一般都存放在 eax 寄存器中。

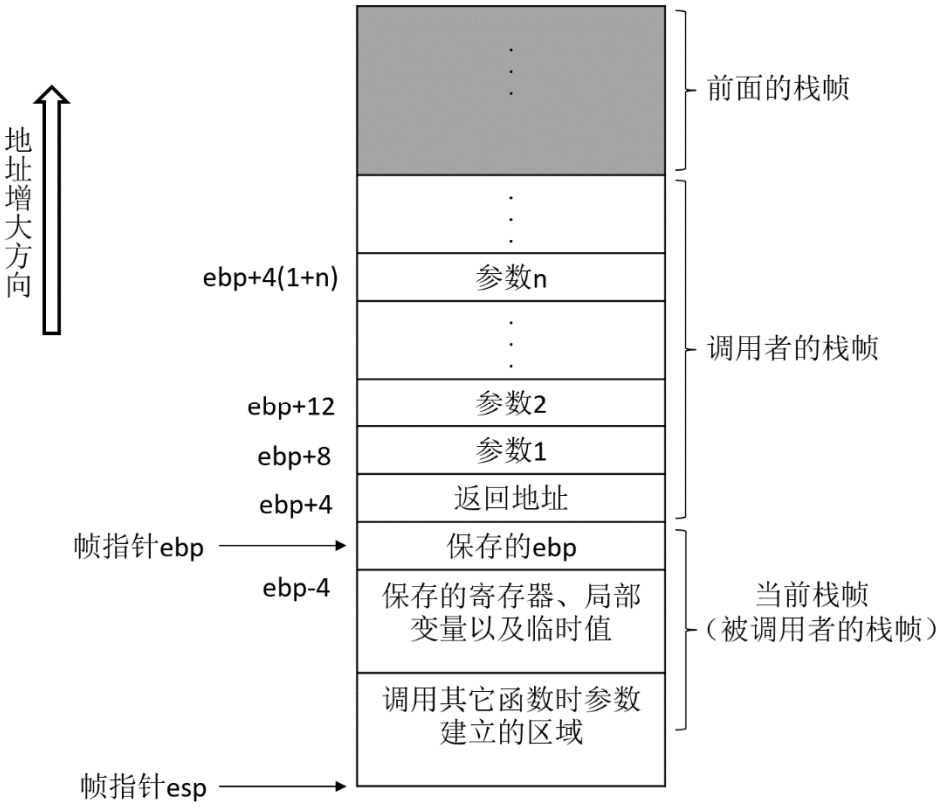

图 3-5 栈帧中的布局

访问栈帧中的变量需要通过两个指针寄存器，以 x86 程序为例（x64 则用 rbp 和 rsp 寄存器），寄存器 ebp 中存放帧指针，指向栈帧的最底部，而寄存器 esp 中存放栈指针，指向栈帧的最顶部。栈帧中大多数信息的访问都是通过帧指针的，一般来说，栈从高地址向低地址存储，越是低的地址则是越靠后入栈的，函数的参数的访问形式为 [ebp + offset]，而局部变量的访问形式一般为 [ebp − offset]。

本文的方法是对每一个函数，通过分析变量在栈帧中的布局，以及数据在寄存机中的传递过程，恢复所有可能的目标变量。此外，由于寄存器作为数据传输的中间存储媒介，同一个寄存器在不同时间段中的值是不同的，因此本文将不同时间段的寄存器看作不同的变量，并对取值不同的寄存器赋予不同的标记。

Algorithm 1 Variable Recovery Algorithm *varrec(f)*

Input: a target function f

Output: the possible variable set V

1: $V = \emptyset$
2: $D = regmark(f)$
3: **for** each $instruction\ i \in f.b$ **do**
4: **if** $[ebp \pm offset] \in i$ **then**
5: $V = V \cup \{[ebp \pm offset]^f\}$
6: **end if**
7: **if** $reg_n \in D(i).u \cup D(i).d$ ***then***
8: $V = V \cup \{reg_n^f\}$
9: **end if**
10: **if** $addr \in i$ **then**
11: $V = V \cup \{addr\}$
12: **end if**
13: **end for**
14: **return** V

图 3-6 算法 1：变量恢复

算法 1 描述了目标变量的恢复过程，如图 3-6 所示。该算法以一个函数的汇编代码段作为输入，返回该函数中的目标变量的集合。在该算法的描述中，上标 f 代表着这个变量属于函数 f，V 是一个用于收集变量的集合，一旦算法找到一个新的变量就将这个新的变量添加到集合 V 中。算法刚开始执行的时候，集合 V 初始化为空（第 1 行）。接

着对函数中出现的寄存器作标记（第 2 行， 见算法 2），同一个寄存器（即 eax, ebx, ecx, edx）在取值改变的情况下应该被赋予不同的标记，即将函数转化为静态单赋值形式。最后，算法扫描每一条指令来收集变量（第 3-13 行）：第 4 行-第 6 行从栈帧中收集变量，第 7 行至第 9 行收集该函数中的寄存器变量，第 10 行至第 12 行收集全局变量。由于堆区中的变量一般为复合变量，主要是通过指针访问的，堆区变量的恢复在 3.6 节详细阐述。另外还有一些特殊情况，比如考虑到编译器的优化，多个变量可能会共享一个堆栈地址，此处不作考虑。

算法 2 是标记寄存器的过程，如图 3-7 所示，将函数体转化为静态单赋值形式（SSA）。该算法以一个函数的汇编代码段作为输入，并返回一个用于给不同指令中的寄存器做标记的映射 D。算法 2 的一个重要依据就是“定义-使用链”，因此此处先简要说明一下“定义-使用”关系。如果一个指令 i 从寄存器 r 中读取数据，那么指令 i 就是寄存器 r 的一个“使用”。而如果指令 i 是将数据写入寄存器 r 中，那么寄存器 r 则是被指令 i “定义”。例如，指令“add eax, 1”先从寄存器 eax 中读取数据，然后对数据做加 1 操作，再将结果存至 eax 寄存器中。所以该指令既是寄存器 eax 的一个“使用”，又是寄存器 eax 的“定义”。另外，一个函数调用指令是对寄存器 eax 的定义，例如指令“ call _function_ ”，因为当函数返回时，其返回值存放在寄存器 eax 中。如果其它寄存器在函数调用时被改变，那么函数调用也是这些寄存器的“定义”。算法 2 的这个标记过程本质上是一典型的“定义-使用”和“使用-定义”分析，在这里本文只考虑数据寄存器。

表 3-1 程序 *memchr* 中的非寄存器变量

变量	偏移量
Parameter1	[ebp + 8]
Parameter2	[ebp + 0Ch]
Parameter3	[ebp + 10h]
LocalVar1	[ebp – 48h]
LocalVar2	[ebp – 44h]

现在以程序 *memchr* 为例，找出该程序中的目标变量。执行算法 2，标记函数中的寄存器，不同下标的寄存器可以看作不同的变量。在这个例子中，对于寄存器 eax 就有 8 个不同的“定义”（即 $\mathrm{eax}_0 - \mathrm{eax}_7$），对于寄存器 ecx 有 3 个不同的“定义”（即 $\mathrm{ecx}_0 - \mathrm{ecx}_2$），对于寄存器 edx 有 1 个不同的“定义”（即 edx_0）。这看起来数据寄存器的变量个数有很多，但这里还可以通过“定义-使用链”来减少需要考虑的变量（参见下一节）。除了寄存器变量之外，算法 1 还会从栈帧中恢复非寄存器变量，表 3-1 列出了该例子中

的非寄存器变量。其中，有三个变量是函数的参数，另外两个变量是函数的局部变量。三个参数在汇编代码中的表示为 [ebp + 8]、[ebp + 0Ch]、[ebp + 10h]，分别对应源代码中的 buf、chr、cnt 变量。而另外两个局部变量在源代码中并没有显式定义，它们只是用于在低级指令中暂时存放*buf 和 chr 的值。

Algorithm 2 Register Marking Algorithm $remark(f)$

Input: a target function f

Output: the marked mapping D

```
let cfg be the control − flow graph of f
for each node n ∈ cfg do
    D(n).u = ∅ and D(n).d = ∅
end for
num = [ eax → 0, ebx → 0, ecx → 0, edx → 0 ]
enqueue the entry node e and num into queue q
while q ≠ ∅ do
    (n, cur) = dequeue q and (ou, od) = D(n)
    for each r ∈ {eax, ebx, ecx, edx} do
        if n is a use of r then
            D(n).u = D(n).u ∪ {r_cur[r]}
        end if
        if n is a definition of r and D(n).d = ∅ then
            num [r] + + and D(n).d = {r_num[r]}
            cur[r] = num[r]
        end if
    end for
    if (ou, od) ≠ D(n) then
        for each successor n′ of n do
            enqueue (n′, cur) into q
        end for
    end if
end while
return D
```

图 3-7 算法 2：标记寄存器

当然，本文给出的图 3-3 中的汇编代码并不是唯一的，使用不同的编译器或不同的编译条件可能会产生不同的汇编代码，从而或产生不同数量的中间变量，但处理的方法是一致的。

3.3.2 提取与目标变量相关的指令

二进制代码分析第二个步骤就是分别为每一个目标变量提取相关的指令，这些指令能反映出变量是如何存储、解释和操作的，在经过抽象和筛选后，这些指令将作为变量的特征用于训练分类器。

要提取与目标变量相关的指令，最简单的一个解决方案就是提取那些直接使用了目标变量的指令。以 *memchr* 例子中的 [ebp + 8] 变量为例，图 3-8 列出了直接使用了变量 [ebp + 8] 的指令，从中可以看到所有指令的操作码都是一个 mov 操作，然而 mov 操作可以用于很多类型的变量，仅仅依靠这些指令的话信息量明显太少。实际上，这个方案忽略了一些没有直接使用目标变量的指令。由于高级语言的代码在编译成低级别的指令后，一行代码可能被编译成多条指令，其中有些指令并没有直接使用目标变量，因此这个简单的方案不可行。例如，如图 1-1 所示，C 语言代码中的“if ($decode$)”这一行代码经过编译后，其对应的汇编指令有两行。其中的一条指令直接使用了变量（movzx eax, byte ptr [ebp − 1]），而另一条指令没有直接使用变量（ test eax, eax）。尽管第二条指令没有直接使用变量，但寄存器中存放的值就是变量 [ebp − 1] 的值，此时寄存器 eax就可以当作变量 [ebp − 1]。第一条指令将一个 8 字节的变量 [ebp − 1] 的值存放到 eax 寄存其中，第二条指令则是判断 eax 寄存器中的值是否为 0，综合这两条指令透露出来的信息，这个变量最有可能是一个布尔型变量，而且第二条指令更能体现布尔类型的行为。由此可见，提取变量相关的指令不仅需要提取直接使用了变量的指令，还需要提取间接使用到该变量的指令，即当变量被存放到寄存器中之后，还需要提取与该寄存器相关的指令。

另一方面，提取与寄存器相关的指令也不是一件简单的事。汇编指令中有许多不同的数据寄存器，例如 eax、ebx、ecx、edx，它们通常被用作临时存储数据的中介，并且在不同的时间点，它们可以存储不同类型的数据。不同时间段的同一个寄存器里可能会存放着不同变量的值，当一个寄存器被定义，若要提取该寄存器相关的指令，则只需要提取该寄存器被再次定义之前的相关指令。另外，根据经典类型系统[31]，当将一个变量赋值给另一个变量时，前者的类型是后者的一个子类型。这表明被赋值变量的行为也属

于赋值变量。特别要说明的是，根据引理 3.1，在本文的类型格中，赋值变量与被赋值变量应具有相同的类型。因此，本文在提取指令时将赋值变量与被赋值变量的相关指令做合并，合并之后的指令集合同时作为赋值变量与被赋值变量的相关指令。

```
09        mov   eax , [ebp +8]
14        mov   eax , [ebp +8]
16        mov   [ebp +8] , eax
26        mov   eax , [ebp +8]
28        mov   [ebp +8] , eax
29        mov   ecx , [ebp +8]
```

图 3-8 程序 *memchr* 中直接使用了[ebp+8]的指令

引理 3.1. 若 t 和 s 分别表示格 L 中的两种类型，则有 $t <: s \Leftrightarrow t = s$.

证明：在图 3-2 给出的类型格 L 中，若 t 和 s 为 L 中的不同的两种类型，不存在 $t <: s$。对于任意一种类型 t，只有 $t <: t$。

“定义-使用链”和“使用-定义链”是本文提取与目标变量相关的指令的基础，在提取与某一变量相关的指令时，正是利用数据寄存器上的“定义-使用链”来提取那些并没有直接使用到目标变量的指令，若某一寄存器被目标变量的定义，则当该寄存器被使用时，也被认为是该变量的使用。还是以图 1-1 中的“$\text{if}\,(decode)$”这一行代码为例，由于在“$\text{movzx eax, byte ptr [ebp} - 1]$”中，寄存器 eax 被变量 $[\text{ebp} - 1]$ 定义，而在“test eax, eax”中，寄存器 eax 被使用。此时寄存器 eax 的使用就被认为是变量$[\text{ebp} - 1]$的使用。因此，“test eax, eax”也是与变量 $[\text{ebp} - 1]$ 相关的指令。

算法 3 描述了为目标变量提取相关指令的过程，如图 3-9 所示。该算法以一个目标变量 v 作为输入，返回与该变量相关的指令的集合 I。算法开始时，集合 I 被初始化为空集。如果某一变量 v 只在函数 f 的栈帧中出现，即该变量为函数 f 的局部变量，那么算法 3 只对函数 f 进行分析即可，否则对整个程序进行分析（第 2-6 行）。然后，调用算法 2 对每一个函数标记“定义-使用链”，使用集合 D 来记录标记结果（第 8 行）。对于每一条指令，如果该指令包含目标变量 v，则将该指令添加到集合 I 中（第 10-11 行）。并且，如果该指令还是对某个数据寄存器的定义，则与该数据寄存器相关的指令也需要添加到集合 I 中（第 12-13 行）。此外，本文还考虑到了过程间的分析：1. 如果当前指令调用了函数 f'，并且变量 v 是函数 f' 中的参数，那么需要在函数 f' 中提取参数 $v^{f'}$ 的相关指

令并添加到集合 I 中（第 16-18 行）；2. 如果变量 v 还是函数 f 的返回值，那么对于每一个调用了函数 f 的函数 f'，为函数 f 的返回值（返回值一般保存在寄存器 eax 中）提取相关指令（第 20-24 行）。

Algorithm 3 Instruction Extraction Algorithm $insext(v)$

Input: a target variable v

Output: the related instrcution set I

```
1:  I = ∅
2:  if v ∈ f then
3:      F = {f}
4:  else
5:      F = the set of all functions
6:  end if
7:  for each function f ∈ F do
8:      D = regmark(f)
9:      for each instruction i ∈ f do
10:         if v ∈ i or v ∈ D(i).u then
11:             I = I ∪ {i}
12:             for reg_n ∈ D(i).u do
13:                 I = I ∪ insext(reg_n)
14:             end for
15:         end if
16:         if i calls function f' and v is an argument then
17:             I = I ∪ insext(v^f')
18:         end if
19:     end for
20:     if v is (stored in) the return variable of f then
21:         for f' calling f do
22:             I = I ∪ insext(eax_f^f')
23:         end for
24:     end if
25: end for
26: return I
```

图 3-9 算法 3：相关指令的提取

还是以程序 *memchr* 中的变量 [ebp + 8] 为例，根据算法 3 提取出来的相关指令，如图 3-10 所示，注释中标明了寄存器的“定义”和“使用”情况，寄存器的下标 n 代表该寄存器是第 n 次被定义。与图 3-8 相比，图 3-10 多了四条更有意义的指令，这些指令是根据数据寄存器的“定义-使用链”而收集的。

```
09    mov     eax , [ebp +8]          %% def of eax1 by [ebp +8]
10    movsx   ecx , byte ptr [eax]    %% use of eax1
14    mov     eax , [ebp +8]          %% def of eax2 by [ebp +8]
15    add     eax , 1                 %% use of eax2 , def of eax3 by eax2
16    mov     [ebp +8] , eax          %% use of eax3
26    mov     eax , [ebp +8]          %% def of eax6 by [ebp +8]
27    sub     eax , 1                 %% use of eax6 , def of eax7 by eax6
28    mov     [ebp +8] , eax          %% use of eax7
29    mov     ecx , [ebp +8]          %% def of ecx3 by [ebp +8]
30    mov     [ebp -44h] , ecx        %% use of ecx3
```

图 3-10 程序 *memchr* 中变量[ebp+8]的相关指令

3.3.3 特征的选择和向量表示

根据 x86 指令集的官方文档[32]，存在超过 600 种操作码，而每一个操作码都有很多不同的用法，可以组合不同类型的操作数形成指令，并且变量在指令中既可能作为左操作数也可能作为右操作数。操作码与操作数的不同组合体现了不同的含义，故本文对收集到的相关指令进行抽象处理。首先，应注意到并非所有指令对类型恢复都有帮助，有些指令只是完成一些功能上的操作，对恢复类型帮助不大。例如“push ...”和“pop ...”只是用于将变量压入栈或弹出栈，并没有反映出对恢复类型有帮助的信息。在预处理的时候，这些明显对恢复类型没有什么帮助的指令都可以不考虑。第二，注意到对于拥有两个操作数的指令，目标变量位于左操作数和右操作数具有完全不同的含义。以二元操作码 mov 所构成的指令为例，mov 操作码接受两个操作数，左操作数和右操作数分别代表数据传输的终点和起点。因此，根据指令中变量在左操作数还是右操作数，以及相邻操作数的不同，本文对同一操作码但不同操作数的指令进行区分。第三，即使操作码相同，操作数的不同组合也可能会给指令带来不同的含义。因为操作码和操作数的组合实在是太多了，本文对操作数进行抽象。根据数据寄存器的大小，用 Reg^n 表示寄存器，其中上标 n 代表寄存器的大小。立即数的抽象表示分别为 0、1、imm，其中 0 和 1 是比较

特殊的立即数，布尔类型的变量只有这两个取值，imm 代表 0 和 1 以外的立即数。而显式的内存地址则用 addr 表示。经过抽象后，指令转化为更简洁的表达形式，本文将其称作“指令的使用模式”。如表 3-2 所示，这些是 mov 操作码的使用模式，其中下划线代表着目标变量。

表 3-2 *mov* 指令的使用模式

mov _ , reg^{32}	mov reg^{32} , _	mov _ , reg^{16}	mov _ , reg^{16}
mov _ , reg^{8}	mov _ , reg^{8}	mov _ , addr	mov addr , _
mov _ , 0	mov _ , 1	mov _ , imm	

由于操作码的数量多达六百多种，每种操作码可能还有许多种使用模式，如表 3-2 所示，mov 操作码就有 11 种使用模式，这使得需要考虑的指令种类非常多。然而，并非所有指令使用模式都具有很强的类别区分能力，并非所有指令对恢复类型都能贡献足够的信息。因此为了简化学习模型，增强模型的泛化能力，需要从所有指令的使用模式中筛选出最具有代表性、最具有类别区分能力的指令使用模式。此处本文使用了一个经典的统计方法，即词频-逆向文件频率（TF-IDF）算法[33]。该算法常用于自然语言处理中，可以用来评估一个字段相对于一个语料库中的其中一份文档的重要程度。而在此处，本文用 TF-IDF 算法来评估一个指令使用模式对于某一类型的类别区分能力。类别区分能力随着指令使用模式在该类型的相关指令中出现的次数成正比增加，但同时会随着它在所有类型的相关指令中出现的频率成反比下降。

给定一种类型的所有变量的相关指令使用模式，TF 表示某一个指令使用模式出现的频率，对于某一指令的使用模式 t_i 来说，它的 TF 值可表示为：

$$\mathrm{tf}_{i,j} = \frac{T_{i,j}}{\sum_k T_{k,j}} \qquad (3-2)$$

以上式子中 $T_{i,j}$ 是指令使用模式 t_i 对于 j 类型的变量中所有的相关指令中出现的次数，而分母则是j 类型的变量中所有的相关指令的总数。IDF 是评判一个指令使用模式普遍重要性的度量。某一指令使用模式的 IDF，可以由所有类型的变量数目除以使用到该指令使用模式的变量数目，再将其商取对数得到：

$$\mathrm{idf}_i = \frac{|\mathrm{L}|}{|\{k: \mathrm{t}_i \in l_k\}|} \qquad (3-3)$$

其中 |L| 是所有类型的变量数目，$|\{k: \mathrm{t}_i \in l_k\}|$ 是用到了指令使用模式 t_i 的变量的数目。若一个指令的使用模式在某一类型的变量中被使用的频率高，并且该指令使用模式在其它类型的变量中被使用的少，就可以产生高权重的 TF-IDF 值。

$$\mathrm{TFIDF}_{i,j} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i \quad (3-4)$$

TF-IDF 的值越高则认为该指令的使用模式对该类型具有很好的类别区分能力，适合用来分类。因此，TF-IDF 算法倾向于过滤掉类别区分能力低的指令使用模式，保留重要的指令使用模式。

根据 TF-IDF 算法的结果，本文选择 n 个类别区分能力最强、最具有代表性的指令使用模式作为机器学习中的特征指示。本文在实践中发现，选择 100 条指令使用模式就已经足够。另外，其它对恢复类型有帮助的信息也有必要作为特征指示。例如，变量占用的内存大小对于恢复类型的帮助很大，如果能够获取到变量占用的空间，那么其占用多大的空间就作为该变量的一个特征。另外，对于能够准确识别名称的函数，比如系统调用库中的函数，函数参数的类型是已知的，那么如果一个变量是该函数的参数，该变量的类型就应该是对应的参数类型的子类型。为此，算法 3 中的第 16-18 行为将父类型的相关指令添加到子类型的相关指令中。具体来说，如果某个函数 f' 的原型已知，变量 v 是 f' 中的其中一个参数，其在函数 f' 对应的参数名为 $v^{f'}$，变量 v 是 $v^{f'}$ 的子类型，$v^{f'}$ 作为父类型，其具有的特征可以作为子类型变量 v 的特征。若变量 v 作为函数 f 的返回值（第 20-24 行）同理。

最后，在将上述特征数据用于训练分类器之前，还需要建立数学模型，本文用空间向量模型来建立数学模型。详细地说，就是把对变量的信息处理简化为向量空间中的向量运算，当目标变量都被表示向量空间中的向量，就可以通过计算向量之间的相似性来度量变量直接的相似性。前文已经介绍了特征的选取，根据这些特征将变量表示为向量，比如选取的指令使用模式及其它信息的数目为 100，那么就用维度为 100 的向量来表示变量，具体各个维度的值就是这个特征的特征值。关于权重的计算方法有很多种，对于表示指令使用模式的特征，本文使用该指令使用模式出现的次数来代表，而对于表示其它信息的特征就分别使用 1 和 0 代表是否具有这个特征。用更形式化的表述，变量表示为如下向量：

$$v = [t_1 : x_1, t_2 : x_2, \ldots, t_n : x_n] \quad (3-5)$$

上式中 n 为选取的特征的数目，t_i 是一个特征项，x_i 则是特征项 t_i 的权重。例如，*memchr* 程序中变量 [ebp + 8] 的特征项如表 3-3 所示，表中只列出了权重非 0 的特征。特别注意一下“mov eax , _ ”和“mov ecx , _ ”，由于 eax 和 ecx 都是 32 位的数据寄存器，这两条指令实质上相同的指令使用模式，在经过处理之后这两条指令被合并，特征项“mov reg^{32} , _ ”的权重为 4。

表 3-3 变量[ebp+8]的特征

处理前		处理后	
特征	特征值	特征	特征值
size	32	$size^{32}$	1
mov eax , _	3	mov reg^{32} , _	4
movsx ecx , [_]	1	movsx reg^{32} , [_]	1
add _ , 1	1	add _ , 1	1
mov _ , eax	2	Mov _ , reg^{32}	2
sub _ , 1	1	Sub _ , 1	1
mov ecx , _	1	Merged to mov reg^{32} , _	
mov [ebp − 44h] , _	1	Mov addr , _	1

3.4 分类器的训练

本文使用有监督学习的方法来训练分类器，为此，还需要为训练样本提供标签。本文预先收集的二进制程序都提供源代码，使用生成调试信息的编译选项重新编译源代码，这样在新生成的二进制文件中就能通过调试信息找到变量名称、变量类型这些关键信息。这些从二进制代码的调试信息中提取的类型信息将作为标签指导分类器的训练。

准备好训练集后，就可以开始训练分类器。用 V 表示特征空间中所有可能的向量，训练分类器的问题可以定义为：给定一个带标签的集合 $D_0 = \{(v_1, l_1), (v_2, l_2), \dots, (v_m, l_m)\}$，其中 m 为变量的个数，v_i 是表示变量的向量，$l_i \in L$ 是变量 v_i 对应的类型。其目标是找到这样一个分类器 C: $V \to L$，使得分类器预测的结果与原给定标签的距离之和最小，即下面公式的值最小，

$$argmin_C \sum_{(v,l)\in D_0} d(\mathrm{C}(v), l) \qquad (3-6)$$

其中，函数 d 的结果是两个类型的距离。与 TIE[16]类似，本文定义了距离函数 $d(s,t)$ 来度量类型 s 与类型 t 的接近程度：若 t 与 s 存在子类型关系，$\mathrm{d}(t,s)$ 的值为类型格中 t 与 s 之间的线段数；否则 $\mathrm{d}(t,s)$ 的值为类型格的最大高度。$d(s,t)$ 的值越小，则说明类型 s 与类型 t 在格中的距离就越近，反之 $d(s,t)$ 的值越大，则说明类型 s 与类型 t 在格中的距离就越远。

分类器 C 旨在找出变量 v 最有可能的类型，本文给分类器 C 的定义是：

$$\mathrm{C}(v) = argmax_{l\in L}\, \mathrm{P}(v, l) \qquad (3-7)$$

其中 $\mathrm{P}(v,l)$ 是指变量 v 是类型 l 的概率。不失一般性，$\mathrm{P}(v,l)$ 的定义如下：

$$\mathrm{P}(v,l) = exp^{Score(v,l)} / \sum_{l \in L} exp^{Score(v,l)} \qquad (3-8)$$

其中 $Score(v,\ l)$ 是一个打分函数，将变量 v 的类型看作 l 能得到的分数越高，则变量v是类型 l 的可能性就越大。

由于要根据变量的行为特征来学习其最有可能的类型，打分函数$Score$的值等于 n 个特征函数 F_i 与相应的权重 w_i 的乘积之和：

$$Score(v,l) = \sum_{i=1}^{n} w_i \times F_i(x_i,l) = \vec{w} \times \vec{F}(\vec{v},l) \qquad (3-9)$$

其中 $\vec{v}$ 是变量 v 的特征向量，$\vec{v} = [t_1{:}x_1, t_2{:}x_2, \dots, t_n{:}x_n]$，$\vec{w}$是由权重 w_i 组成的向量，$\vec{w}$是由特征函数 F_i 组成的向量。

对于特征函数的计算，有多种解决方法，例如：信息熵、相对熵（KL 距离）等。此处本文使用 TF-IDF 的值作为特征函数的值，根据第 3.3.3 节对 TF-IDF 值的数目说明，特征函数的定义为：

$$F_i(x_i,l) = x_i \times tfidf\ {}^{l}_{t_i} \qquad (3-10)$$

其中，$tfidf\ {}^{l}_{t_i}$ 是特征项 t_i 与类型 l 的 TF-IDF 值。

剩下的问题就是为每个特征函数求出合适的权重 w_i，使其满足公式（$3-6$）的值最小。这个问题可以用不同的算法解决，例如决策树、K 邻近算法、朴素贝叶斯和支持向量机等。本文已经尝试使用了一些简单的算法来训练分类器，并对这些算法进行了实验和对比，详情见第五章实验部分。

3.5 基本类型的预测

训练好分类器之后，分类器就可以用于预测新的二进制文件中的数据类型。直观地说，分类器会为新给定的目标变量求出其最大可能性属于的类型 l，使得 l 满足：

$$argmax_{l \in L}\ P(v,l) \qquad (3-11)$$

仍然以 *memchr* 程序中的变量 [ebp + 8]为例，其相关指令包含“mov reg^{32}, __；movsx reg^{32}, [__]”（将变量的值作为地址，再到该地址所表示的单元中取值存放到寄存器中）、“mov __, reg^{32}；add __, 1”（取地址的下一个单元）、“mov __, reg^{32}；Sub __, 1”（取地址的前一个单元），这些是典型的指针变量的用法。根据分类器的评判，$P([\mathrm{ebp}+8], pointer)$ 的值最大，即变量 [ebp + 8] 最有可能的类型就是指针类型。同理，图 1-1 中 *base64* 程序中的 *decode* 变量的相关指令主要包含“mov __, 0；mov __, 1；movzx reg^{32}, __；test __, __”，这些是典型的布尔变量所用到的指令。比起其它变量，$P(decode, bool)$ 的

值更大，因此 *decode* 变量最可能是布尔类型的变量。

3.6 复合类型的恢复

本节说明如何恢复复合类型，主要包括指针和结构体，复合类型的恢复是在恢复出基本类型的基础上进行的，同样也需要结合程序分析和机器学习方法。

3.6.1 指针

指针类型的变量，其值是另一个内存单元的地址，即指针通过地址找到所需的变量单元。这里面涉及到两个变量，指针变量自身和被指针指向的变量。指针变量自身的类型为指针类型，被指针变量指向的变量也有所属的类型，一个完整的指针类型表达应该涵盖这两者的类型，例如 int*类型的变量代表该变量是指针，并且该指针指向的变量类型是整型。指针也可以是多级的，当指针变量所指向的变量也是一种指针时，则称指针变量为多级指针。以常用的二级指针为例，二级指针类型实际上是一级指针变量的地址。指针所指向的变量也可以作为一个目标变量，利用其相关指令来预测其所属的类型。根据这种思路，就可以一级一级地预测出指针指向的变量类型，从图 3-2 给出的类型格开始，若预测出变量的类型为指针，则进入类型格的下一级，再预测指针所指向的变量的类型，最后，整合信息得出完整的指针类型。根据这种思路，多级指针的类型恢复问题也能得以解决。恢复指针类型的详细步骤如下：

（1）一旦分类器将变量 v 预测为指针类型，则根据“指向分析”来寻找变量 v 所指向的变量。算法 4 给出了文本的指向分析算法，该算法是在 Brumley 等人[34]工作的基础上提出的。

（2）如果变量 v 所指向的变量存在，则提取与这些变量相关的指令，利用分类器预测被指向变量的类型 t 。根据“定义-使用链”和引理 3.1，一个指针指向的所有变量应具有相同的类型，根据被指向变量的特征可以预测出被指向变量最有可能属于的类型 t。

（3）当变量 v 所指向的变量能找到至少一个，则该指针变量的类型为 $*t$,否则该指针变量的类型为 $*$ 。

算法 4 是本文给出的指向分析算法，如图 3-11 所示，给定一个目标变量 v ，返回该变量所指向的变量的集合 V。集合 V 从空集开始，一旦找到一个变量 v 所指向的变量 v'，或者指针变量 v 以 $*v$ 的形式被使用，则将 v' 和 $*v$ 添加到集合 V 当中（第 9-10 行）。考虑到变量的传递性，若存在另一个指针变量的值与 v 相同，则另一个指针变量指向的变量也要添加到集合 V 中（第 12-14 行）。以上步骤的思想和 IDB 谓词[34]的思想十分相

似，不需要对表达式和运算符制定规则，因为此处不需要根据指针变量的值来分析其所指向的变量，而是根据指针的使用情况来将指针与其它变量建立联系。另外，算法还考虑到了过程间分析的情况（第 15-23 行）

Algorithm 4 Point-to Algorithm $point\_to(v)$

Input: a target variable v

Output: the possible point − to variable set V

```
1:  V = ∅
2:  if v ∈ f then
3:      F = {f}
4:  else
5:      F = the set of all functions
6:  end if
7:  for each function f ∈ F do
8:      for each instruct i ∈ f do
9:          if i is v' = *v or *v = imm then
10:             V = V ∪ {*v, v'}
11:         end if
12:         if i is v' = v or v = v' then
13:             V = V ∪ point_to(v')
14:         end if
15:         if i calls function f'and *v is an argument then
16:             V = V ∪ {p_{*v}^{f'}}
17:         end if
18:     end for
19:     if *v is (stored in) the return variable of f then
20:         for f' calling f do
21:             V = V ∪ {eax_f^{f'}}
22:         end for
23:     end if
24: end for
25: return V
```

图 3-11 算法 4：指向分析

以程序 *memchr* 中的变量 [ebp + 8] 为例，接着第 3.5 节的内容，由于P([ebp +

8], $pointer$) 的值最大，变量 [ebp + 8] 最可能属于的类型是 *pointer*，那么根据指向分析寻找该变量所指向的变量，记做 ∗ [ebp + 8]，以及被指向变量的相关指令“{movsx byte ptr [eax], ecx}”。其相关指令主要表达对一个 8 位大小的数据做带符号的 mov 操作。经处理后，该被指向变量的特征向量表示如下：

$$[\text{movsx}\, reg^{32}, \_ : 1 ; \text{Size}^{8} : 1] \qquad (3-12)$$

根据分类器的计算：

$$P(*\,[\text{ebp} + 8], char) > P(*\,[\text{ebp} + 8], bool) > \cdots \qquad (3-13)$$

被指向的变量∗ [ebp + 8]最有可能属于的类型是 $char$ 类型，因此可以判定变量 [ebp + 8] 的类型为 $char*$ 类型，这与该变量在源代码中的显式定义是一致的，预测正确。

3.6.2 结构

结构是另一种常见的复合类型，它是由一系列具有相同类型或者不同类型的数据构成的数据集合（数组或结构体）。本文主要考虑通过指针访问的结构，即只考虑通过指针变量 struct ∗ 访问的结构。其原因在于在栈中直接定义和使用的结构，在经过编译器编译之后，结构这个包装好的整体会被拆分成一个个基本类型的变量。例如 struct point {int x；int y}，该结构的变量在编译之后只能找到两个整形变量，而重新将这两个整形变量组合成结构具有十分大的挑战。尽管本文的方法目前还不支持栈中直接定义和使用的结构，但大部分实际应用中，通过 new 创建的结构和在堆区中动态分配的结构都是通过指针的形式访问的，本文的方法能够恢复出通过指针访问的结构。

恢复结构体的第一个步骤就是从所有的指针变量中，将 struct ∗ 和其它类型的指针区别开来。由于指针变量中存放的是被指向变量的起始地址，若被指向变量是结构，起始地址即为结构中第一个变量的地址，起始地址加上偏移量可以访问结构中的变量。因此，在指向分析的基础上寻找结构的指针变量在汇编指令中常用的访问方式：[base + offset]，将通过相同的 base 值访问的变量整合，组合成结构。

若一个指向结构 s 的指针变量 v 的值为 base，则结构 s 的组成如图 3-12 所示，结构 s 中的变量依次表示为 [base + offset_i]。那么根据本章描述的方法，将 [base + offset_i] 作为目标变量，提取 [base + offset_i] 的相关指令，并将其表示成向量，就可以使用分类器预测 [base + offset_i] 的类型 $type_i$。最后，整合 $type_i$ 的信息恢复出结构 s 的类型，并且指针变量 v 是指向该结构的指针。

虽然本文的方法还有很多不足与局限性，不支持栈中直接使用的结构，也没有考虑

嵌套类型，但可以在一定程度上恢复结构类型。根据第 5.2 节中的实验数据，本文从指针类型中恢复了 37.1%的结构类型。

Struct s {
    $type_0$ [base]
    $type_1$ [base + offset_1]
    $type_2$ [base + offset_2]
    $type_3$ [base + offset_3]
    ...
}

图 3-12 结构 s 的组成

3.7 本章小结

本章详细介绍了本文的二进制代码类型恢复方法。首先，本章介绍了该方法的整体框架。接下来，用一个类型格展示了本文恢复的类型的种类以及它们之间的关系。最后，分别详细描述了本文方法中的四个主要任务：1. 二进制代码分析；2. 分类器的训练；3. 基本类型的预测；4. 复合类型的恢复。

第 4 章 恶意软件检测

4.1 恶意软件检测介绍

所谓恶意软件，是指在未经过用户许可的情况下，在终端上安装或运行攻击者特意安排的恶意操作，侵害用户的合法权益的程序代码或指令集合。恶意软件主要包括蠕虫、病毒、特洛伊木马和间谍软件等。随着互联网的蓬勃发展，恶意软件的数量持续上升，其复杂程度也愈来愈高，对网络安全构成了巨大的威胁，给企业和用户造成了巨大的损失。由于恶意软件的作者可以通过修改源代码来发布新版本的恶意软件或者使用混淆等隐藏技术来绕过恶意软件的检测，因此需要采取多种方法和手段来进行恶意软件的检测和预防。

当前恶意软件的检测方法主要有三大类：基于签名的检测方法、启发式方法和机器学习方法。（1）基于签名的检测方法通过将恶意软件的签名与它数据库中已知的恶意软件签名进行匹配。如果与库中某一已知恶意软件签名相匹配，则判定该软件是恶意软件。这个检测方法的优点是速度较快，但是该方法只能检测出数据库中已知的恶意软件样本，跟不上每天产生数百万个新的恶意软件变种的节奏。（2）启发式方法是通过仿真运行程序，根据恶意软件运行时的行为来检测恶意软件。该方法针对不同类型的恶意软件需要用完全不同的规则来构建启发式分析器的判断逻辑。该检测方法的主要限制是开销较大，并且常在恶意行动已经开始时才发现恶意软件，预防结果推迟。（3）机器学习方法是从已有的恶意软件库中，学习恶意软件的行为特征，使用训练后的模型来检测恶意软件。该方法效率较高，但根据所用特征的不同，其绕过检测的难度也不同，而且训练模型也需要频繁更新。

基于机器学习的检测方法是一种兼顾了效率和有效性的方法，能够检测新型的恶意软件或者是恶意软件的变种。但其中存在的问题就是很多人将机器学习当成无所不能的魔术棒，很多人在用机器学习的时候，只是把尽量多的样例交给算法去计算，这样训练出来的模型即使当时准确率很高，也很容易被各种简单的恶意软件规避技术绕开。例如，修改二进制代码中指令的顺序、增加大量的无关指令、增加大量的无关系统调用函数或者是使用其它多态混淆技术。面对这些技术，如果之前训练的模型反恶意软件能力不足，则很容易被恶意软件绕开检测。目前现有大部分的工作[36][37]都以二进制程序的操作码或者是二进制程序用到的系统调用作为特征。实际上，操作码主要体现的是程序的行为特

征，现有的一些工作主要考虑的是程序行为特征而没有考虑到程序数据特征，例如类型信息。

本文的检测方法与大多数现有的工作不同，本文不仅考虑到了程序的行为特征，还考虑到了程序的数据类型特征。该方法是在恢复出二进制程序的类型信息的基础上进行的，使用第三章提出的方法可以恢复二进制代码的类型信息，将类型信息作为重要特征之一，利用机器学习方法训练的模型来检测恶意软件。

4.2 恶意软件检测的整体流程

本文对恶意软件的检测问题的定义如下：给定二进制可执行文件的集合 E 以及每个可执行文件对应的类别，则有数据集 $\mathrm{D}=\{(e_1,c_1),\ldots,(e_m,c_m)\}$，其中，$e_i \in E$ 是一个可执行文件，$c_i \in C=\{benign,malicious\}$是可执行文件 e_i 对应的类别，而 m 是可执行文件的数量，其任务是要找到一个函数 $f:E \to C$，使得对于 $\forall \mathrm{i} \in \{1,\ldots,m\}$，$f(e_i) \approx c_i$。

该问题可以看作是机器学习中的一个二元分类问题，利用已有的数据集训练一个分类器，分类器能判定给的可执行文件是属于恶意软件还是属于非恶意软件。图 4-1 给出了本文的恶意软件检测方法的整体流程。其中包含两个主要的组件：特征提取器和分类器。特征提取器对待检测的样本做预处理（反汇编和恢复类型信息），然后提取有关操作码的信息、系统调用库的信息以及其中的数据类型信息，最后从提取出的信息中选择最具有代表性的特征，用一个向量来描述待检测样本的特征。而训练好的分类器会对特征提取器产生的向量进行分类，将其归类为恶意软件或非恶意软件。第 4.3 节和第 4.4 节分别详细介绍特征提取器和分类器这两个组件。

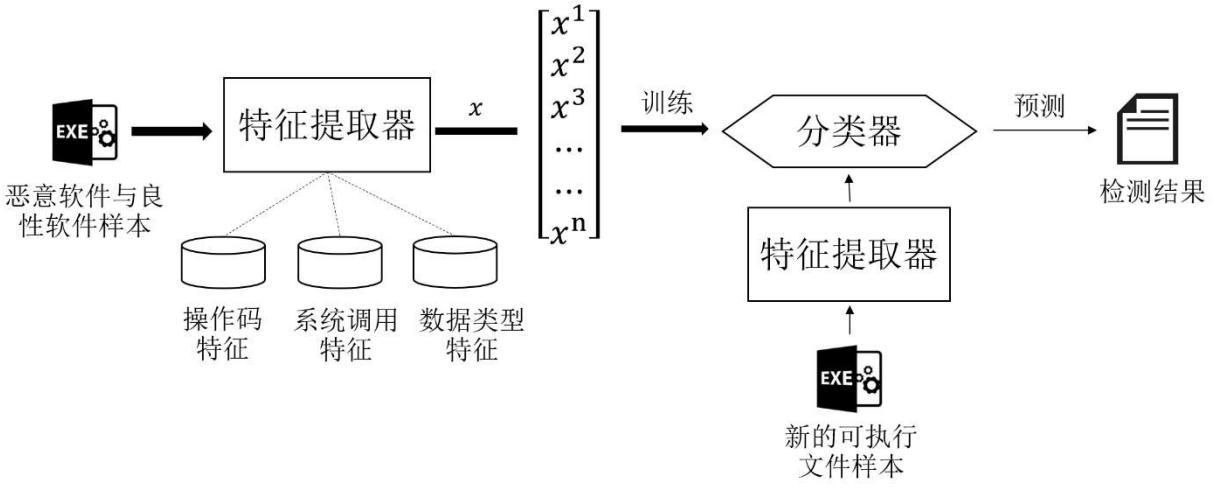

图 4-1 恶意软件检测的整体流程图

4.3 特征提取器

特征提取器是图 4-1 中的一个组件，其主要包括以下步骤：（1）反汇编与类型恢复；（2）信息提取；（3）可执行文件的向量表示。

4.3.1 反汇编与类型恢复

一个可执行二进制软件都有其基本的数据格式，如 windows 的 PE 格式、Linux 的 ELF 格式。可信软件一般会遵循格式要求的约束，恶意软件或被恶意软件感染的可执行文件本身也要遵循格式要求的约束，因此可以通过一些解码技术将其转化为可读性更强的中间表示。以 windows 平台的 x86 应用程序为例，可执行文件主要有$.exe$文件和$.dll$文件，这些文件的二进制形式可读性差，不利于理解与分析，因此本文使用反汇编工具将可执行文件转换为汇编代码。

数据类型信息是能反映程序如何对数据进行存储、解释和操作的。然而在二进制代码和汇编代码中并不会显式地标注变量的数据类型，类型信息无法直接从二进制代码和汇编代码中提取，这就需要利用二进制代码的类型恢复技术来恢复程序中的数据类型信息。本文利用反汇编工具对二进制可执行文件进行反汇编，基于得到的汇编代码，使用第三章提出的二进制代码类型恢复方法恢复其中的数据类型信息，这样就解决了数据类型的信息来源。操作码和系统调用库的信息提取可通过简单的程序分析方法从汇编代码中直接提取。

4.3.2 信息提取

特征提取器的第二步就是从汇编代码中提取信息，这里主要提取三种信息：操作码信息、类型信息和系统调用信息。一般来说，操作码能体现出一个程序的行为和意图，数据类型能够体现出程序所操作的数据的结构，而调用到的系统库则能反映出程序与系统的交互。

Dai 等人[45]从统计分布的角度分析了操作码用于恶意软件检测的可行性，在舍弃操作数的基础上，构造出不包括操作数、仅包括操作码的抽象汇编，并在此基础上使用统计学方法来统计出现频繁的指令，将出现频繁的指令作为特征来构建分类器。故操作码较之机器码更能代表程序的行为特征。系统调用库的使用能代表程序与系统交互的行为，常用的 DLL 文件如图 4-2 所示，这些 DLL 文件分别作用于不同类型的系统资源。恶意软件要产生恶意行为，往往是通过系统提供的动态链接库来使用系统资源的，故通过对系统调用进行分析，可以得到恶意软件的行为特征。因此本文选取的操作码信息、类型

信息和系统调用信息能一定程度上描述一个程序的使命或任务，恶意软件的这三种信息会表现出和正常良性软件的一些差异。

DLL 名称	功能描述
kernel32.dll	用于内存管理和资源处理的底层操作系统函数
user32.dll	用于消息处理、计时器、菜单和通信的窗口管理函数
advapi32.dll	支持众多 API 包括许多安全性和注册表调用的高级 API 服务库
gdi32.dll	包含的函数用来绘制图像和显示文字
wininet.dll	Windows 应用程序网络相关模块
ntdll.dll	控制 NT 系统功能的 NT 内核层动态链接库
ws2_32.dll	网络和互联网应用程序利用包含的 windows 套接字来操作它们的连接
wininet.dll	使程序能够访问标准网络协议，如 FTP 和 HTTP
shell32.dll	用于打开网页和文件，设置默认文件名等功能
psapi.dll	Windows 系统进程状态支持模块
crypt32.dll	Windows 加密 API 应用程序接口模块
wsock32.dll	硬件提取层模块，用于解决硬件的复杂性
ole32.dll	用于编写和整合来自不同应用程序的数据在 Windows 作业系统的骨干部分

图 4-2 恶意软件常用到的 DLL 资源

特征提取器会从反编译得到的.asm 文件中提取关于操作码、数据类型、系统调用库的相关信息，如图 4-3 给出的算法 5 所示。操作码信息是可以直接从汇编代码的指令流中提取的，使用一个列表 l_{opcode} 来收集操作码。刚开始时列表为空，只要汇编代码中出现了列表中不存在的操作码，就将该操作码添加到列表中，否则列表中该操作码的计数加 1（第 4-10 行）。对于系统调用库的提取，与操作码的提取相似使用一个列表 $l_{library}$ 收集出现的系统调用库，由于在.asm 文件中每一个系统调用库只会被导入一次，在汇编代码中的形式为“Imports from xxx. dll”，故将被被导入的 DLL 文件添加到 $l_{library}$ 中即可（第 11-13 行）。对于类型信息的提取，本文第三章的方法的输出结果中包含了二进制程序的变量与数据类型，可从中查找对应变量的类型信息。同样使用一个列表 l_{type} 来收集所有出现过的类型，若在汇编代码中发现目标变量，则查找该变量的数据类型，若 l_{type} 中不存在该类型则将该数据类型添加到 l_{type} 中，否则将 l_{type} 中该类型的计数加 1（第 14-21 行）。最后，建立一个 *Profile*，这个 *Profile* 用于存储.asm 文件中关于操作码、数据类型、系统调用库的相关信息。

Algorithm 5 Executable Information Extraction Algorithm *info*(*asm*)

Input: an assembly code file asm

Output: a *Profile* that collects information about opcode, type and library

1: $l_{opcode} = l_{type} = l_{library} = \{\emptyset\}$
2: $D = regmark(f)$
3: **for** each $line \in asm$ **do**
4: **if** "$opcode, operand$" $\in$ $line$ **then**
5: **if** $opcode \in l_{opcode}$ **then**
6: l_{opcode}["$opcode$"]+= 1
7: **else**
8: $l_{opcode} = l_{opcode} \cup \{$"$opcode$": 1$\}$
9: **end if**
10: **end if**
11: **if** "Imports from dll" $\in$ $line$ ***then***
12: $l_{library} = l_{library} \cup \{$"$dll$": 1$\}$
13: **end if**
14: **if** "[ebp ± offset]" ∈ line **then**
15: recover the $type$ of variable "[ebp ± offset]"
16: **if** $type \in$ l_{type} **then**
17: l_{type}["$type$"]+= 1
18: **else**
19: $l_{type} = l_{type} \cup \{$"$type$": 1$\}$
20: **end if**
21: **end if**
22: **end for**
23: $Profile = [\, l_{opcode}, l_{type}, l_{library} \,]$
24: **return** $Profile$

图 4-3 算法 5：二进制文件的信息提取

4.3.3 程序的向量表示

静态分析获取的信息通常是冗余的，恶意软件的作者为了增加恶意软件检测的难度，会在程序源代码中加入很多垃圾信息，使其中的有效信息提取变得更加困难，所以在特征选择的时候需要去掉冗余的特征，保留比较具有代表性的特征。本文使用 TF-IDF 算

法留下最具有代表性的 l_{opcode} 中的 n 个元素。而数据类型和系统调用库的种类不多，可以全部保留。

根据第二步提取出来的信息，特征提取器会为每一个可执行文件建立一个 *Profile*，这个 *Profile* 可以表示为[l_{opcode}, l_{type}, $l_{library}$]，经过 TF-IDF 算法筛选后，该 *Profile* 中保存着列表 l_i 中的各个元素及其在汇编代码中出现的次数。假设这个字典中项的顺序是固定的，以该字典中的项作为特征，那么字典可以表示为向量 $(x^1, x^2, \ldots, x^n)$,其中 x^i 是第 i 个特征的值，n 是特征的总数。那么一个可执行文件的向量表示就如图 4-4 所示。从这个向量中，可以看出这个可执行文件有以下特点：在该程序中 mov 指令被使用到了 9 次，add 指令被使用了 1 次，lea 指令被使用了 5 次；该程序有 2 个 bool 类型的变量和 3 个 int 类型的变量；

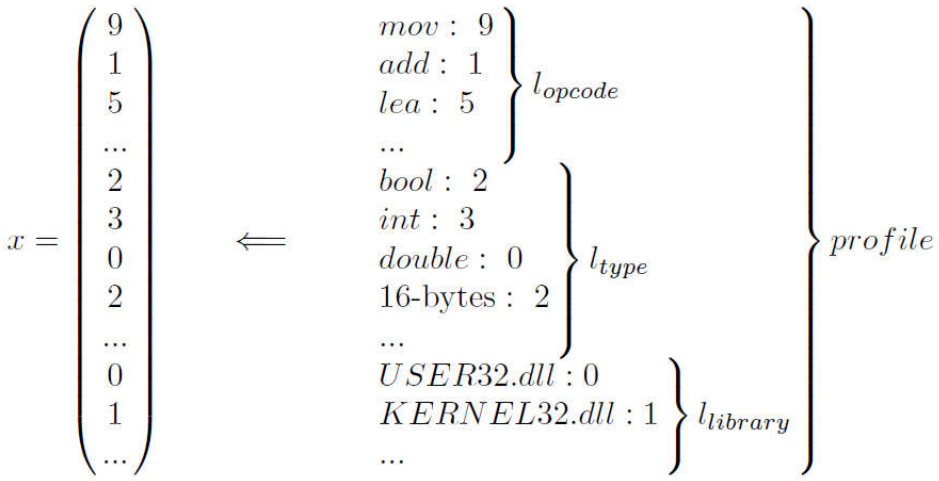

图 4-4 一个可执行文件的向量表示示例

4.4 分类器

分类器是图 4-1 中的第二个组件，其主要包括 2 个阶段：（1）分类器的训练（图 4-1 中的粗体箭头部分）；（2）软件恶意性检测（图 4-1 中的细箭头部分）。先利用已知类别的数据集，通过有监督学习的方法训练分类器。一旦分类器训练完成，就可以使用该分类器来检测新的恶意软件样本。

1. 分类器的训练

特征提取器可以将一个可执行文件 e 表示成向量 x。用 X 表示所有可能的向量，D_0 表示用于训练的数据集，用 0 和 1 分别代表非恶意软件和恶意软件，那么训练的过程就是要找到一个分类器 $C: X \to [0,1]$使得

$$min\sum_{(x,c)\in D_0}|\mathrm{C}(x)-c| \qquad (4-1)$$

其中，用于训练 $\mathrm{C}(x)$ 的方法可根据需求选取，常见的机器学习方法有决策树、随机森林、支持向量机等。

2. 软件恶意性检测

训练完成后，分类器 $\mathrm{C}: X \to [0,1]$ 就可以用于检测新的恶意软件样本。给定一个可执行文件 e，其特征向量为 x，分类器 C 会为样本寻找一个$\mathrm{c} \in C$，使得

$$min|\,\mathrm{C}(x)-c| \qquad (4-2)$$

例如，以 $\mathrm{c}=0$ 表示恶意软件，以 $\mathrm{c}=1$ 表示良性软件，给定的可执行文件 e 的特征向量为 x，若分类器的结果 $\mathrm{C}(x)=0.1$，则该可执行文件 e 将被判断为恶意软件。

4.5 本章小结

文章介绍了二进制代码的类型恢复的一种应用：恶意软件的检测。与其它已有的恶意软件检测工作不同，本文还分析了程序的类型信息，通过将程序的类型信息作为恶意软件检测的重要特征之一，增加了检测特征的多样性，也提高反恶意软件的能力。第 5.4 节为本章的实验部分。

第 5 章 实验结果与评价

作者使用 Python 语言将第三章的方法实现为一个原型工具，名为 BITY。该工具以 IDA Pro[1]作为前端来将二进制代码转化为汇编代码，基于得到的汇编代码进行静态分析，根据筛选的 100 个特征项利用 Scikit-learn[38]提供的机器学习算法训练分类器，训练好的分类器能够为给定的二进制程序预测其中变量的基本数据类型。最后 BITY 还会整合基本类型的信息恢复复合类型（BITY 的恢复的类型仅考虑到图 3-2 中类型格的第三级）。

在训练数据的选择上，考虑到训练集中应该包含各种变量的不同使用方式，本文选取的训练样本包括：一些 C 语言教科书中的配套光盘中的例子、C 语言常用算法程序集[39]、178 个常用算法的源程序、C\C++的库函数以及一些现实生活中的应用程序。选取这些数据作为训练集的原因是：教科书中的例子和常用的算法程序一般都会覆盖所有的类型及其可能的用法，这些例子往往非常适合初学者的学习，而现实生活中的应用程序则能反映出这些类型在实际应用中的普遍用法。

本章给出了对 BITY 工具的详细测评，首先，通过交叉验证实验来评价不同的机器学习方法所训练的分类器的准确度。其次，使用现有的一些评价标准，将 BITY 与商业工具 Hex-Rays[2]和开源工具 Snowman[3]进行对比试验。再者，还针对不同规模的程序设计了一些扩展实验。最后，通过实验检验恢复出来的类型信息是否有助于恶意软件的检测。以下实验都是在一个英特尔处理器的 PC 机上进行的，其主要配置为 i5-4590（3.30GHz）和 8GB 内存。

5.1 不同分类器的实验结果

由于不同的机器学习算法所训练出来的分类器具有不同的效果，并且对不同的具体问题的适应性也可能不同，因此本文设计了五折交叉验证的实验来评价各个分类器的准确程度。所谓五折交叉验证实验，即先将数据集随机分成相等的五份，每次取其中的一份作为测试集，其余的作为训练集，使用机器学习算法训练分类器后再对测试集上的数据做测试。该过程重复在五份相等的测试集上进行，最后将五次测试的结果汇总作为总的结果。

本文尝试过使用以下算法训练分类器：决策树（DT）[40]、随机森林（RF）[41]、K 邻

[1] The IDA Pro disassembler and debugger: http://www.hex-rays.com/idapro
[2] Hex-Rays decompiler: http://www.hex-rays.com/products/decompiler/index.shtml
[3] Snowman decompiler: http://derevenets.com

近算法（KNN）[42]、支持向量机（SVM）[43][44]和朴素贝叶斯（NB）。其中，决策树分别使用信息增益或最大熵建立；随机森林包含 10 个用信息增益或最大熵建立的决策树；K 邻近算法尝试了 K 的取值为 1、3、5 和 7 时的情况；支持向量机则采用了不同的核函数训练分类器，包括线性核函数、径向基（RBF）核函数和 Sigmoid 核函数；朴素贝叶斯算法则选用了高斯朴素贝叶斯（GNB）、多项式朴素贝叶斯（MNB）和伯努利贝叶斯（BNB）算法。

测评采用的度量标准如下：

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad (5-1)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad (5-2)$$

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}} \qquad (5-3)$$

其中，TP 和 FP 分别表示 true positives 和 false positives，TN 和 FN 分别表示 true negative 和 false negative。Precision 是精确率，Recall 是召回率，$F1$ 值为综合评价指标，其值为精确率和召回率的加权调和平均数。这三个值都介于 0 和 1 之间，值越接近 1 则表示效果越好。表 5-1 展示了不同分类器下的精确率、召回率和 *F1* 值。

表 5-1 不同分类器的表现情况

分类器	**Precision**	**Recall**	***F1***
KNN (k = 1)	0.9462	0.9437	0.9432
KNN (k = 3)	0.9390	0.9368	0.9364
KNN (k = 5)	0.9337	0.9329	0.9325
KNN (k = 7)	0.9296	0.9289	0.9384
DT (gini)	0.9457	0.9447	0.9444
DT (entropy)	0.9434	0.9418	0.9419
RF (10,gini)	0.9461	0.9457	0.9453
RF (10,entropy)	0.9454	0.9447	0.9444
Gaussian Naïve Bayes	0.7661	0.5775	0.6002
Multinomial Naïve Bayes	0.9172	0.9072	0.9052
Bernoulli Naïve Bayes	0.9240	0.9299	0.9264
SVM (kernel = linear)	0.9466	0.9437	0.9432
SVM (kernel = rbf)	0.9440	0.9427	0.9425
SVM (kernel = sigmoid)	0.6619	0.5548	0.4304

从表 5-1 的结果可以发现，除了高斯朴素贝叶斯算法和使用 Sigmoid 核函数的 SVM 算法的效果不佳，其它大部分算法的效果都非常好，精确率、召回率和 *F1* 值都在 0.9 以上。特别是使用线性核函数的 SVM 表现出最高的精确率（94.66%），而使用信息增益建立随机森林表现出最高的召回率（94.57%）和 *F1* 值（0.9453）。关于随机森林的表现稍好的原因可能是因为它是一种并行式集成学习的方法，进一步在决策树的训练过程中引入随机属性选择。而支持向量机通过把原本在低维不可分的数据集映射到高维可分空间中简单线性计算来达到可分的目的。基于以上结果，在机器学习算法的选择上，本文建议采用 SVM 或者随机森林，BITY 工具正是选择了 SVM 算法。

5.2 BITY 与 Hex-Rays、Snowman 的对比实验

本节展示的实验为 BITY 与 Hex-Rays（v2.2.0.15）和 Snowman（v0.1.0）工具的比较，其中 Hex-Rays 是商业工具 IDA Pro 上的一个插件，Snowman 是开源的 C/C++反编译器。

由于这三个工具采用的类型格略有不同，为了能够公正和定量地测评这三个工具的能力，需要使用同一个标准的类型格来评判，且这个类型格应包含三个工具能给出的所有类型。本文将第 3.2 节给出的类型格做了扩展，使其能够适用于 Hex-Rays 和 Snowman。同时，还扩展了第 3.4 节给出的距离函数，使其能在扩展后的格中计算两个类型间的距离。图 5-1 给出了扩展之后的类型格，图中除了 $\bot$ 和 $\top$，Hex-Rays 和 Snowman 考虑到了所有的类型，而 BITY 只考虑了图中用粗体显示的类型。要说明一点，不同工具可能对同一个类型名称叫法不同，例如，Snowman 分别使用 int32_t 和 uint32_t 表示有符号的 32 位整形和无符号的 32 位整形，而 Hex-Rays 分别使用 int 和 unsigned int。但这仅仅是名称叫法有区别，这三个工具给出的结果中，每一个类型都在格中能找到唯一的类型与之对应。要扩展原来的距离函数 d 使其适用于新的类型格并不难，相比 3.2 节的类型格，新扩展的格中包含了更多的子类型关系，例如将一个 int 类型的变量预测为 dword 类型并不精确，但依旧比猜测成 $\top$ 类型要好。无论是 dword 类型还是 $\top$ 类型都比预测成一个完全错误的类型（例如 float）要好。因此，为了更精确地体现这种关系，本文借助了 TIE[16] 提出的“可兼容的类型”。所谓可兼容的类型，即给定两个类型，如果其中一个类型是另一个类型的子类型，那么就说这两个类型是可兼容的。例如 dword 与 int 是可兼容的，但前者不如后者精确。给定任意两个类型 t 和 s，$\mathrm{d}(t,s)$ 的定义如下：

(1) 如果 t 和 s 分别是指向类型 t' 和 s' 的指针，那么 $\mathrm{d}(t,s)$ 的具体的取值要根据 t'

和 s' 的关系来确定，若 t′ 和 s' 不兼容，则 $\mathrm{d}(s,t)=2$（即最大高度的一半乘以 1）；若 t′ 和 s' 是可兼容的，则 $\mathrm{d}(s,t)=1$（即最大高度的一半乘以 0.5）；若 t′ 和 s' 完全一致，则 $\mathrm{d}(s,t)=0$（即最大高度的一半乘以 0）。

(2) 其它情况下，若在格的最顶级中 t 和 s 是可兼容的，则 $\mathrm{d}(t,s)$ 的值为 t 与 s 之间的线段数；否则 $\mathrm{d}(t,s)$ 的值为格的最大高度，即为 4。

例如，$\mathrm{d}(*dword,\ *int)$ 和 $\mathrm{d}(dword,\ int)$ 的距离都是 1，而 $\mathrm{d}(*dword,\ int)$ 的距离是 4。当然，由于 BITY 只包含各种粗体部分的类型，相比原来的格，一旦 BITY 给出错误的答案，其距离函数的值也会更大。

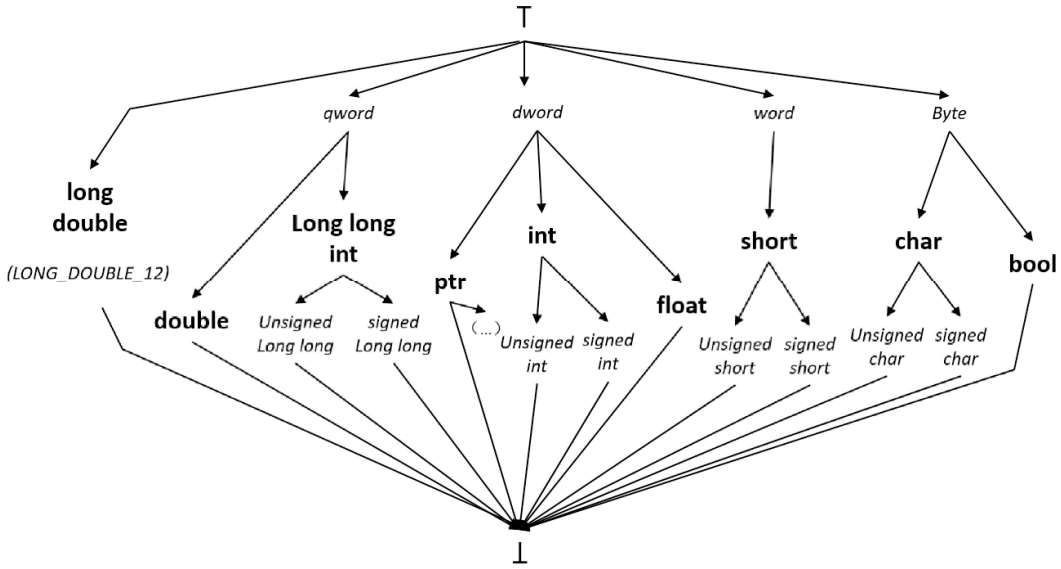

图 5-1 扩展后的类型格

对于测试程序，本文选择了来自 GNU Core Utilities 中的程序，即 *coreutils-v8.4*，这是一个普遍在其它相关工作中[15][16][19][23]使用的一个评价基准。实验的主要流程如下：（1）测试用的二进制程序都使用 *coreutils-v8.4* 的源代码重新编译，生成带调试信息的二进制文件，从调试信息中可以提取出各个变量的数据类型作为评测准确性的正确答案；（2）使用 BITY 分析没有调试信息的二进制文件，将 BITY 恢复出来的变量及其类型与正确答案作对照；（3）使用 Hex-Rays 反编译不带调试信息的二进制文件，从反编译后的代码中提取出变量及其类型与正确答案作对照；（4）Snowman 同 Hex-Rays。由于 BITY 与 Hex-Rays 都使用 IDA Pro 作为反汇编前端，而 Snowman 使用了不同的反汇编前端，因此在无法确定 BITY 与 Snowman 的变量对应关系的情况下，对 Snowman 的变量类型的提取采用了人工查找的方式。

表 5-2 给出了 BITY、Hex-Rays 和 Snowman 对 *coreutils-v8.4* 中部分程序的分析结果。其中，program 这一列给出了 *coreutils* 中的二进制程序的名称，Vars 列表示目标变量的个数，R、C 和 F 分别代表变量类型完全正确的个数、变量类型为可兼容的类型的个数，以及结果不正确的个数。P 列给出的是正确的结果与可兼容的结果占总变量数的百分比。在后中，本文用 proper types 表示正确的类型或可兼容的类型。从表中的结果可以发现：（1）一共列举了 45 个程序，其中的 44 个程序中，BITY 能为 80%以上的变量恢复出 proper types，而 Hex-Rays 和 Snowman 在能为 80%以上的变量恢复出 proper types 的程序数量分别 26 和 19。（2）从整体上看，一共 2333 个变量，BITY 恢复的变量类型中有 1356 个（58.12%）是精确的，729 个（31.25%）是可兼容的，即一共 2085 个（89.37%）变量的类型属于 proper type；Hey-rays 恢复的变量类型中有 1276 个（54.69%）是精确的，589 个（25.25%）是可兼容的，即总共有 1865 个（79.97%）变量的类型属于 proper types；Snowman 恢复的变量类型中有 995 个（42.65%）是精确的，713 个（30.56%）是可兼容的，即一共有 1708（73.21%）个变量的类型属于 proper types。（3）平均来看，对于恢复 proper types 的准确率，BITY、Hex-Rays 和 Snowman 分别为 90.32%、82.96%和 74.28%。由此可见，无论是精确类型上，还是可兼容的类型上，BITY 都比 Hex-Rays 和 Snowman 要准确。

表 5-2 BITY、Hex-Rays 和 Snowman 的对比实验

Program	Vars	BITY				Hex-Rays				Snowman			
		R	C	F	P	R	C	F	P	R	C	F	P
base64	41	20	19	2	95.12%	29	7	5	87.80%	21	4	16	60.98%
basename	22	17	4	1	95.45%	12	4	6	72.73%	11	1	10	54.55%
cat	50	29	19	2	96.00%	18	19	13	74.00%	18	18	14	72.00%
chron	55	39	8	8	85.45%	32	7	16	70.91%	28	12	15	72.73%
chgrp	31	21	4	6	80.65%	17	4	10	67.74%	17	10	4	87.10%
chmod	42	19	20	3	92.86%	20	13	9	78.57%	13	15	14	66.67%
chown	17	10	4	3	82.35%	6	6	5	70.59%	8	7	2	88.24%
chroot	23	12	9	2	91.30%	18	4	1	95.65%	8	7	8	65.22%
cksum	14	6	7	1	92.86%	7	6	1	92.86%	4	5	5	64.29%
comm	20	10	4	6	70.00%	11	1	8	60.00%	10	4	6	70.00%
copy	135	69	48	18	86.67%	50	42	43	68.15%	29	72	34	74.81%
cp	78	46	26	6	92.31%	45	23	10	87.18%	21	43	14	82.05%
csplit	66	27	32	7	89.39%	26	25	15	77.27%	24	16	26	60.61%

cut	47	32	14	1	97.87%	31	15	1	97.87%	21	17	9	80.85%
date	30	18	8	4	86.67%	15	8	7	76.67%	11	10	9	70.00%
dd	128	81	35	12	90.63%	78	30	20	84.38%	63	33	32	75.00%
df	92	51	32	9	90.22%	45	25	22	76.09%	27	35	30	67.39%
dircolors	55	31	23	1	98.18%	26	23	6	89.09%	24	10	21	61.82%
du	68	27	28	13	80.88%	26	15	27	60.29%	17	32	19	72.06%
echo	11	8	3	0	100%	5	6	0	100%	7	3	1	90.91%
expand	25	16	8	1	96.00%	16	9	0	100%	13	6	6	76.00%
expr	85	29	39	17	80.00%	28	35	22	74.12%	23	22	40	52.94%
factor	30	20	9	1	96.67%	22	4	4	86.67%	17	3	10	66.67%
fmt	62	40	15	7	88.71%	40	7	15	75.81%	37	13	12	80.65%
fold	25	17	8	0	100%	20	5	0	100%	14	8	3	88.00%
getlimits	20	17	3	0	100%	17	2	1	95.00%	15	1	4	80.00%
groups	9	5	4	0	100%	5	4	0	100%	3	4	2	77.78%
head	111	63	41	7	93.69%	52	42	17	84.68%	37	36	38	65.77%
id	20	13	5	2	90.00%	12	5	3	85.00%	7	10	3	85.00%
join	106	48	52	6	94.34%	54	24	28	73.85%	44	46	16	84.91%
kill	27	18	6	3	88.89%	15	9	3	88.89%	12	6	9	66.67%
ln	29	23	3	3	89.66%	21	5	3	89.66%	11	13	5	82.76%
ls	352	189	105	58	83.52%	186	73	93	73.58%	156	93	103	70.74%
mkdir	22	15	4	3	86.36%	10	5	7	68.18%	6	6	10	54.55%
mkfifo	10	7	2	1	90.00%	7	0	3	70.00%	6	0	4	60.60%
mktemp	35	23	9	3	91.43%	16	15	4	88.57%	16	13	6	82.86%
mv	35	20	8	7	80.00%	15	8	12	65.71%	7	14	14	60.00%
nice	16	15	1	0	100%	15	1	0	100%	12	1	3	81.25%
nl	18	11	4	3	83.33%	12	3	3	83.33%	8	3	7	61.11%
nohup	22	20	1	1	95.45%	19	1	2	90.91%	13	7	2	90.91%
od	120	88	23	9	92.50%	86	25	9	92.50%	82	18	20	83.33%
operand2sig	13	11	0	2	84.62%	9	2	2	84.62%	9	4	0	100%
paste	35	26	7	2	94.29%	24	9	2	94.29%	16	15	4	88.57%
pathchk	19	15	3	1	94.74%	14	4	1	94.74%	8	8	3	84.21%
pinky	62	34	22	6	90.32%	44	9	9	85.48%	41	9	12	80.65%
总计	2333	1356	729	248	-	1276	589	468	-	995	713	625	-
平均值	-	-	-	-	90.32%	-	-	-	82.96%	-	-	-	74.28%

接下来，针对复合类型，本文还专门统计了 BITY、Hex-Rays 和 Snowman 对于指针变量和指向结构体的指针变量的准确程度。

在所有的变量类型中，指针变量的类型恢复是比较复杂的，但也是至关重要的。指针变量可以看作复合类型，其中包含了指针变量和指针指向的变量两者的类型信息。根据二进制程序的调试信息，表 5-2 的 45 个二进制程序中一共有 2333 个变量，其中有 1021 个变量属于指针类型的变量。表 5-3 给出了 BITY、Hex-Rays 和 Snowman 恢复指针类型变量的准确程度。表中 Num 代表指针变量的数目；R 表示对于指针变量和指针指向的变量都能给出正确的结果；C 代表可兼容的类型，在此处对于指针变量来说，表示能够判断出变量是指针，但对于指针指向的变量的类型判断有误；F 表示推断错误，不能够判断出变量是指针。从数值上看，BITY 精确地恢复出了 444 个（43.49%）变量的类型，397 个（38.88%）变量的类型是可兼容的，即一共 841 个（82.37%）变量的类型属于 proper types；Hex-Rays 精确地恢复出了 397 个（38.88%）变量的类型，203 个（19.88%）变量的类型是可兼容的，即一共有 600 个（58.77%）变量的类型属于 proper types；而 Snowman 精确地恢复出了 238 个（23.31%）变量的类型，399 个（39.08%）变量的类型是可兼容的，即一共有 637 个（62.39%）变量的类型属于 proper type。该实验结果表明，在指针类型的恢复上，BITY 也比商业工具 Hex-Rays 和开源工具 Snowman 要准确。

表 5-3 恢复指针类型的实验结果

Num	BITY			Hex-Rays			Snowman		
	R	C	F	R	C	F	R	C	F
1021	444	397	180	397	203	421	238	399	384

在这 1021 个指针类型的变量中，有 350 个变量指向结构体，即有 350 个变量的类型是 struct*。表 5-4 给出了 BITY、Hex-Rays 和 Snowman 对 struct*类型的恢复结果。在这 350 个 struct*类型的变量中，BITY 给出的结果中有 296 个是指针类型（pointer），其中只有 130 个是 struct*类型；Hex-Rays 给出的结果有 146 是指针类型，其中只有 60 个是 struct*类型；Snowman 给出的结果中有 216 个是指针类型，其中有 182 个是 struct*类型。在指针类型上，BITY 的准确率依旧比 Hex-Rays 和 Snowman 要高，但若考虑到 struct*类型，BITY 恢复的 struct*类型比 Hex-Rays 要多，但少于 Snowman。究其原因可以发现，Snowman 倾向于将指针变量赋予 struct*类型，对于很多不是指向结构体的指针，Snowman 也倾向于推断为 struct*。

表 5-4 恢复 struct* 类型的实验

Num	BITY		Hex-Rays		Snowman	
	pointer	struct*	pointer	struct*	pointer	struct*
350	296	130	146	60	216	182

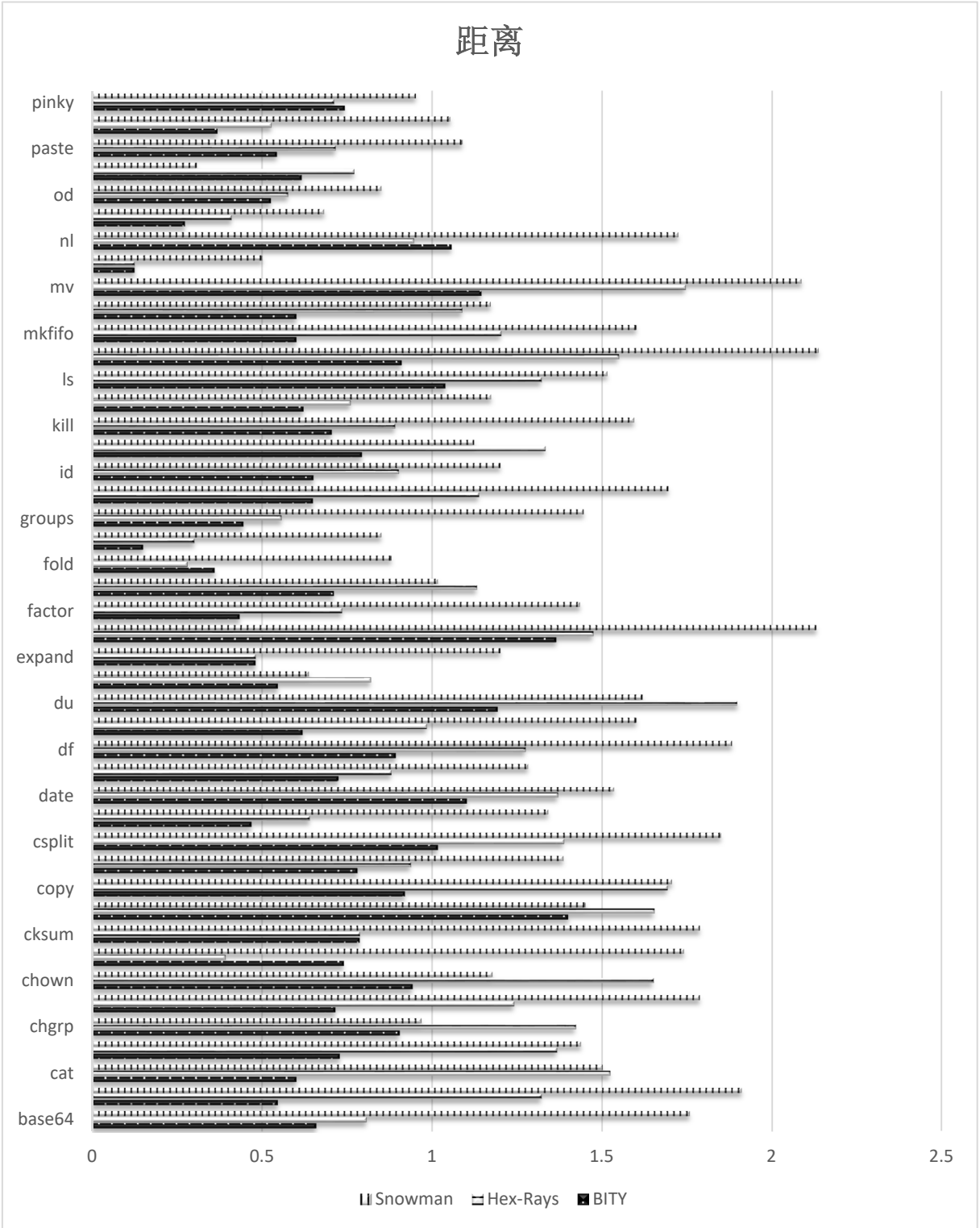

图 5-2 BITY、Hex-Rays 和 Snowman 对于 *coreutils-v8.4* 的平均距离

此外，经过人工分析 BITY、Hex-Ray 和 Snowman 一些判断错误的案例，究其错误的原因主要有一下四点：(1)有些变量使用的频率不高，提取出来的相关指令也特别少，对于这类相关指令特别少的变量容易判断错误。(2)对指针变量的类型恢复准确率不高，由表 5-3 和表 5-4 的数据显示，判断错误的案例中，指针变量占大多数。BITY，Hex-Rays 和 Snowman 的失败案例中，指针变量所占的比重分别为 72.58%、89.96%和 61.44%。（3）对于一些复合类型的恢复并不容易，例如 BITY 没有考虑到数组类型和栈区中的结构体类型。（4）对于 struct*类型，BITY 的判断条件较为严格，只有在保证指针指向一片区域，并且该区域中有多个偏移量能访问时，才判定变量是 struct*类型，因此对于很多 struct*类型的变量，BITY 仅仅判断其为指针类型。

最后，再以平均距离这个指标来分别评测 BITY、Hex-Rays 和 Snowman。对于每一个变量，如果从其调试信息中提取的类型为 t，类型恢复工具给出的类型为 s，距离函数 $\mathrm{d}(t,s)$ 的值就是类型为 s 与类型 t 之间的距离。对于任意一个程序，其各个变量的对应的类型依次为 $[t_1, t_2, t_3, \dots, t_n]$，类型恢复工具给出的对应结果是 $[s_1, s_2, s_3, \dots, s_n]$，则平均距离为：

$$\overline{\mathrm{d}}(t,s) = \frac{\mathrm{d}(t_1,s_1) + \mathrm{d}(t_2,s_2) + \cdots + \mathrm{d}(t_n,s_n)}{n} \qquad (5-4)$$

由于 $0 \le \mathrm{d}(t,s) \le 4$，若平均距离越接近 0，则说明该类型恢复工具的效果越好；若平均距离越接近 4，则说明该类型恢复工具的效果越差。图 5-2 给出了 BITY、Hex-Rays 和 Snowman 对于 *coreutils-v8.4* 中 45 个程序的平均距离。在大部分情况下，BITY 的平均距离要小于 Hex-Rays 和 Snowman。平均来看，BITY、Hex-Rays 和 Snowman 对这 45 个程序的平均距离分别为 0.715、1.014 和 1.372，这意味着 BITY 恢复的数据类型信息比 Hex-Rays 和 Snowman 恢复的数据类型信息要更加精确。

5.3 BITY 对不同规模程序的性能测评实验

本节的实验主要是测试 BITY 对不同规模的程序的性能表现，为此，选用了不同大小的二进制程序来测试 BITY 在运行时的表现。这些二进制程序主要来自日常生活中常用的应用软件。在表 5-5 中给出了 BITY 对于从 7KB 到 1.3G 大小不等的程序所表现的性能情况。在表中，Program 列给出了二进制程序的名称，Size 表示程序的大小，Aloc 代表程序经过 IDA Pro 反汇编器后的汇编代码的行数，Vars 代表目标变量的个数，Time-P 代表预处理所用的时间，其中包括反汇编步骤所用的时间和提取变量及其相关指令所用的时间（秒），Time-L 代表类型预测步骤用到的时间（秒）。

表 5-5 BITY 对不同规模程序的测评实验结果

Program	Size	Aloc	Vars	Time-P	Time-L
strcat	7 KB	508	8	0.187	0.011
Notepad++ 7.3.3_Installer	2.80 MB	12032	113	0.807	0.229
SmartPPTSetup_1.11.0.7	4.76 MB	128381	166	1.156	0.365
DoroPDFWriter_2.0.9	16.30 MB	25910	71	0.629	0.068
QuickTime_51.1052.0.0	18.30 MB	61240	247	2.132	0.607
Firefox Portable	110.79 MB	12068	113	0.906	0.254
VMware workstation v12.0	282.00 MB	39857	352	3.739	0.911
opencv-2.4.9	348.00 MB	61636	287	4.130	0.722
VSX6_Pro_TBYB	1341.44 MB	129803	450	4.762	1.921

从表中的数据可以发现：（1）在整个过程中，预处理所用的时间占很大的比重。随着 Aloc 和 Vars 的增加，预处理所用的时间线性增加。（2）类型预测所用的时间非常短，并且随着目标变量的增加而增加。（3）BITY 在 7KB-1.3GB 的实际应用程序中的类型预测时间非常短，这表明，BITY 的性能表现较优秀、可扩展性较强，比较适合用于实际使用。

5.4 恶意软件检测实验

本文还实现了一款能自动化检测恶意软件的工具。该工具使用 IDA Pro 作为前端来解码二进制程序，使用前文提到的 BITY 工具来恢复二进制程序中的类型信息，并借助 Scikit-learn 提供的机器学习算法实现分类器。实验主要针对 PC 平台 x86 架构的应用程序，在数据集的选择上，样本包括恶意软件样本和非恶意软件的样本。恶意软件的样本主要来自 BIG 2015 Challenge[4]和 theZoo aka Malware DB[5]中 2017 年以前的恶意软件样本。非恶意软件的样本来自 360 软件管家，这可以保证这些样本的安全性较高。本文选取的数据集一共包含 11376 个恶意软件的样本和 8003 个良性软件的样本。

5.4.1 不同特征的实验结果

基于数据集，本文分别以操作码、系统调用库、数据类型，以及它们的组合作为特征，分别使用朴素贝叶斯、K 邻近算法、随机森林、支持向量机这四种机器学习算法进行十折交叉验证实验。表 5-6 列举了实验结果，图 5-3 为对应的 ROC 曲线，其中 O、L 和 T 分别代表操作码、系统调用和数据类型。Accuracy 为分类器正确预测的比例，AUC

[4] Microsoft Malware Classification Challenge: https://www.kaggle.com/c/malware-classification
[5] theZoo aka Malware DB: http://ytisf.github.io/theZoo

（Area Under Curve）是 ROC（Operating Characteristic curve）曲线下的面积。

表 5-6 不同特征的恶意软件检测结果

特征	分类器	Precision	Recall	F1	Accuracy	AUC
O	Multinomial Naïve Bayes	0.9492	0.6128	0.7449	0.8266	0.9790
	KNN (k=3)	0.9521	0.9777	0.9648	0.9705	0.9842
	RF (n=10,gini)	0.9595	0.9775	0.9672	0.9736	0.9947
	SVM (kernel = linear)	0.9462	0.9701	0.9580	0.9639	0.9914
	平均值	0.9518	0.8840	0.9087	0.9339	0.9709
L	Multinomial Naïve Bayes	0.8508	0.7705	0.8087	0.8494	0.9397
	KNN (k=3)	0.7439	0.9892	0.8492	0.8549	0.8878
	RF (n=10,gini)	0.9439	0.8328	0.8872	0.9105	0.9736
	SVM (kernel = linear)	0.9401	0.8014	0.8653	0.8969	0.9661
	平均值	0.8711	0.8485	0.8526	0.8785	0.9497
T	Multinomial Naïve Bayes	0.8346	0.1829	0.3001	0.6476	0.9427
	KNN (k=3)	0.8570	0.9735	0.9115	0.9219	0.9493
	RF (n=10,gini)	0.8751	0.9790	0.9284	0.9336	0.9741
	SVM (kernel = linear)	0.8208	0.9427	0.8776	0.8913	0.9452
	平均值	0.8483	0.7701	0.7544	0.8496	0.9427
O+T	Multinomial Naïve Bayes	0.9492	0.6115	0.7438	0.8260	0.9537
	KNN (k=3)	0.9580	0.9803	0.9690	0.9741	0.9845
	RF (n=10,gini)	0.9643	0.9801	0.9722	0.9768	0.9954
	SVM (kernel = linear)	0.9508	0.9813	0.9659	0.9714	0.9937
	平均值	0.9556	0.8883	0.9127	0.9371	0.9818
L+T	Multinomial Naïve Bayes	0.8533	0.7718	0.8105	0.8510	0.9426
	KNN (k=3)	0.9190	0.9810	0.9490	0.9564	0.9724
	RF (n=10,gini)	0.9447	0.9801	0.9621	0.9681	0.9927
	SVM (kernel = linear)	0.9268	0.9694	0.9476	0.9557	0.9861
	平均值	0.9109	0.9256	0.9173	0.9328	0.9735
O+L	Multinomial Naïve Bayes	0.9461	0.6561	0.7749	0.8425	0.9675
	KNN (k=3)	0.9575	0.9798	0.9685	0.9737	0.9856
	RF (n=10,gini)	0.9683	0.9770	0.9726	0.9773	0.9956
	SVM (kernel = linear)	0.9569	0.9856	0.9711	0.9757	0.9939
	平均值	0.9572	0.8996	0.9218	0.9856	0.9856

O+L+T	Multinomial Naïve Bayes	0.9494	0.6590	0.7778	0.8446	0.9710
	KNN (k=3)	0.9572	0.9798	0.9678	0.9736	0.9859
	RF (n=10,gini)	0.9678	0.9787	0.9733	0.9778	0.9959
	SVM (kernel = linear)	0.9581	0.9858	0.9718	0.9763	0.9948
	平均值	0.9582	0.9008	0.9227	0.9431	0.9869

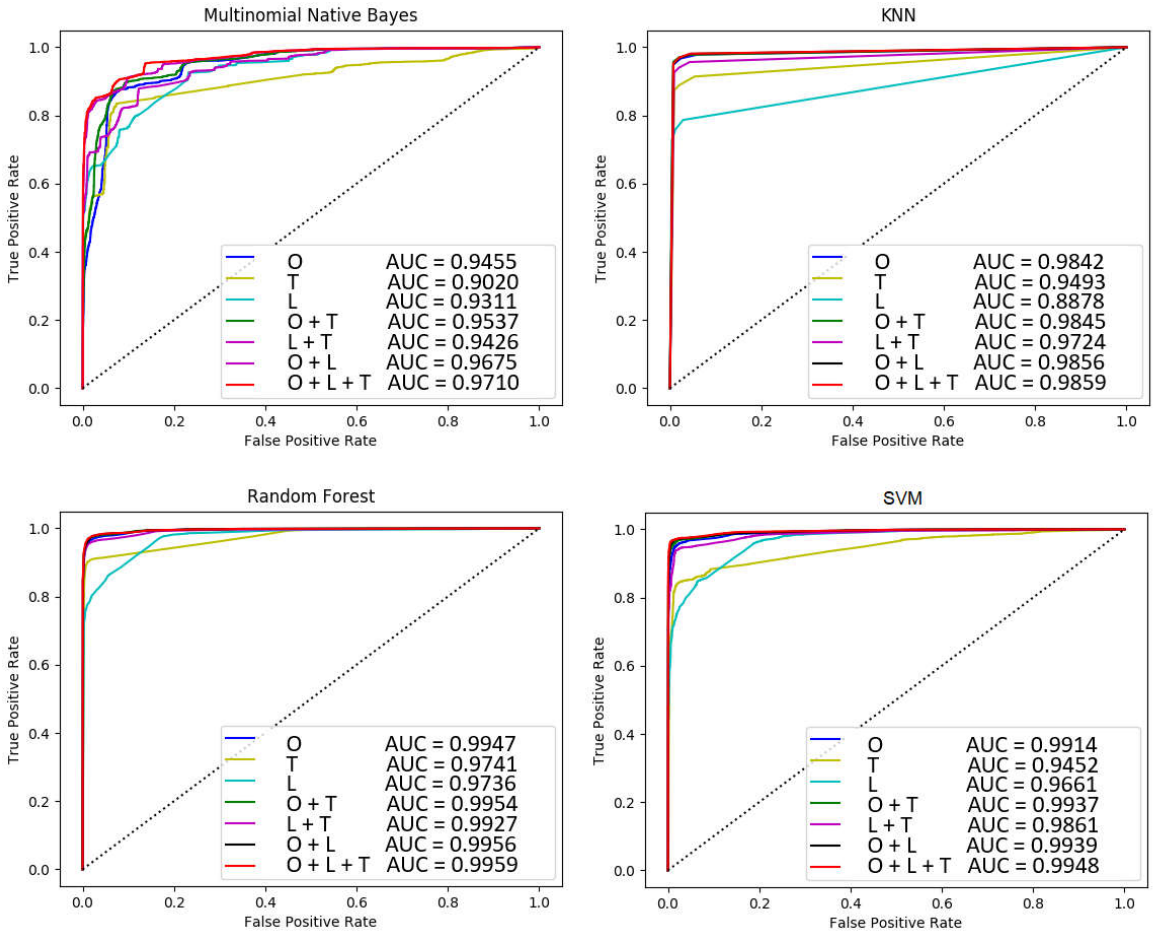

图 5-3 不同特征的 ROC 曲线

由实验结果可以看出，利用类型信息能够有效的检测恶意软件，其平均精确度为 84.83%，平均准确率为 84.96%，平均 AUC 值为 0.9427。相比另外两个特征，数据类型检测恶意软件的能力与系统调用相当，但弱于操作码。从平均准确率来看，使用系统调用和数据类型作为特征的分类器比只用了系统调用作为特征的分类器提高了 5.43 个百分点；使用操作码和数据类型作为特征的分类器比起只用操作码作为特征的分类器提高了 0.34 个百分点。这表明，数据类型信息能够提高反恶意软件的能力。

5.4.2 新恶意软件样本的检测结果

Saxe 和 Berlin 等人[46]指出，对于恶意软件的检测，当前时间点所训练出来的模型对

于恶意软件的检测能力，并不能代表对新版本的恶意软件的检测能力，其能力会随着时间的推移快速下降。为此，本文特意以 2016 年 12 月以前的数据集训练模型，选取了 2017 年 1 月以后的恶意软件作为测试集进行实验，以验证本文工具对较新版本的恶意软件的检测能力。实验选取了 DAS MALWERK[6]上 2017 年 1 月至 2017 年 7 月收集到的恶意软件作为测试样本，一共有 364 个样本。表 5-7 给出了本文工具对这 364 个样本的检测结果，其中 O、L 和 T 分别代表操作码、系统调用和数据类型特征。只使用数据类型作为特征的分类器比没有使用数据类型作为特征的分类器检测出更多的样本。另外，即使是只以数据类型作为特征的分类器也能检测出 287 个（78.8%）样本。由此可见，增加数据类型作为特征可以提高反恶意软件的能力。

表 5-7 新恶意软件样本的检测结果

新恶意软件样本数	O + L + T	O + L	T
364	297	295	287

5.4.3 抗混淆技术测试实验

另一个实验是检测模型对抗混淆技术的能力。混淆技术的使用会让恶意软件的检测更加困难，攻击者可以利用一些已知的恶意软件，通过改变其中的指令顺序、通过添加 jump 指令修改控制流、改变的寄存器的使用方式、插入大量无关指令或者增加大量无关系统调用，产生新的恶意软件的变种。面对这些新的恶意软件的变种，需要恶意软件检测模型具有很强的对抗混淆技术的能力。为此，该实验以增加了混淆技术的恶意软件样本对原有模型进行测试。本文使用了两个代码混淆工具：*Obfuscator*[7]和 *Unest*[8]。

Obfuscator 是一个能混淆汇编程序源代码的工具，它能将输入的源代码转换为受保护的版本，以防止被破解或被逆向工程分析。本文使用的是 *Obfuscator* 的一个免费版本，其只提供了“Change code execution flow”功能。本文先从恶意软件库中随机选取了 50 个样本的汇编源代码，然后使用 *Obfuscator* 对这 50 个样本进行混淆，产生 50 个新的恶意软件变种。最后，使用本文的工具进行测试。

Unest 是一个二进制代码混淆工具，其使用手册上明确描述该工具的功能就是用于规避恶意软件检测。该工具提供四种混淆方式：（1）混淆数字变化的规则算法；（2）混淆输出字符串的构造；（3）混淆目标代码的写入部分，把代码段写入到栈内并跳转过去

[6] DAS MALWERK: http://dasmalwerk.eu
[7] Obfuscator: http://www.pelock.com/products/obfuscator
[8] Unest: an obfuscation engine for binary code

执行；（4）对静态库进行混淆，从而影响所有链接的代码。本文先从恶意软件库中随机选取了 15 个样本，分别以这四种混淆方式对 15 个样本进行混淆，一共产生 60 个新的变种。最后，使用本文的工具进行测试。

测试结果如表 5-8 所示，被 *Obfuscator* 和 *Unest* 混淆后的所有恶意软件都被成功检测出来。这说明本文的工具对混淆技术具有一定的抵抗能力。具体来看这些混淆技术的做法，插入跳转指令、改变指令的执行流等主要是会改变二进制程序的操作码特征，而混淆静态库主要是会改变二进制程序的系统调用特征。实际上，目前大多数混淆工具都主要是修改二进制程序的操作码特征和系统调用特征，而修改数据类型特征的混淆工具不多。因此，在增加了类型信息这种特征后，更多样性的特征能一定程度上提高模型的反恶意软件能力。

表 5-8 抗混淆技术测试实验结果

混淆工具	测试样本数	检测结果	准确率
Obfuscator	50	50	100%
Unest	60	60	100%

5.5 本章小结

本章为实验与结果评价，详细介绍了本文二进制代码类型恢复与恶意软件检测的相关实验。二进制代码类型恢复的实验部分，首先在机器学习算法的选择上，比较了各个算法的效果，实验结果表明 SVM 和随机森林的效果最好。其次，将本文实现的原型工具 BITY，分别与商业工具 Hex-Rays 和开源工具 Snowman 进行了对比实验，就准确率和距离这两个指标进行测评。实验表明，无论是在精确类型上，还是在可兼容的类型上，BITY 都比 Hex-Rays 和 Snowman 要准确。本章还测试了 BITY 对实际应用中不同规模程序的性能表现。实验用 BITY 分析了一些 10KB-1.3GB 大小不等的应用程序，BITY 都在较短的时间内完成分析。因此，本文方法的可扩展性较强，比较适合用于实际使用。最后，本章还展示了第四章中恶意软件的检测部分的实验与分析，实验表明二进制程序的数据类型信息能够提高反恶意软件的能力。

第 6 章 总结与展望

6.1 总结

二进制代码的类型恢复在二进制代码分析中有着非常重要的意义，该技术在不同领域也有着很大的需求。在此背景下，本文提出了一种新的方法来恢复出二进制代码中变量的类型。与现有的一些工作不同，本文没有采用像约束求解这样的分析技术，而是融合了程序分析与机器学习方法，先从二进制代码的指令流和数据流中提取出关键信息，再利用这些关键信息来训练分类器，最后根据分类器的预测结果整合得出变量的类型。

本文实现了一个原型工具 BITY，并且还设计了一系列实验来测评该工具。实验结果表明，本文提出的二进制代码类型恢复方法在一定程度上能够较准确地恢复出二进制程序中变量的类型，本文实现的原型工具 BITY 比商业工具 Hex-Rays 和开源工具 Snowman 要精确，并且其性能表现较优秀、可扩展性较强，比较适合用于实际使用。

最后，本文还将二进制代码类型恢复技术应用于恶意软件的检测。与许多已有的研究工作不同，本文不仅考虑了二进制代码的行为特征，还考虑到了二进制代码的数据类型特征。本文将程序的数据类型信息作为恶意软件检测的重要特征之一，实验表明恢复出来的类型信息对恶意软件的检测有帮助，不仅增加了检测特征的多样性，也增强了反恶意软件的能力。

6.2 展望

本文的研究工作能够在一定程度上恢复二进制代码中的变量类型，但由于时间限制以及作者水平有限，本文还有很多地方可以进一步完善，主要包括以下几个方面：

第一，本文没有考虑类型的量型词（signed 和 unsigned）和常量类型（const），在今后的研究工作中会考虑增加对这两种类型的支持。

第二，复合类型的种类繁多，恢复复合类型也较为复杂，本文的方法目前只支持（多级）指针变量的恢复和通过指针访问的结构的恢复。今后的研究工作中可以考虑结合 VSA 方法和 DIVINE 方法来恢复更多的复合类型，例如不通过指针访问的结构、嵌套结构等。

第三，在应用方面，二进制代码的类型恢复在许多领域都有着很大的需求。Caballero 等人[30]调查报告中列举了许多关于二进制代码类型恢复的应用，作者希望在将来的工作中不断改进本文的方法，并将二进制代码的类型恢复应用于更多的领域。

参 考 文 献

[1] Wang S, Liu T, Tan L. Automatically Learning Semantic Features for Defect Prediction[C]. Proceedings of the 38th International Conference on Software Engineering. 2016: 297-308.

[2] Slowinska A, Stancescu T, Bos H. Howard: A Dynamic Excavator for Reverse Engineering Data Structures[C]. Proceedings of Network and Distributed System Security Symposium. 2011.

[3] Caballero J, Grieco G, Marron M, et al. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities[C]. Proceedings of International Symposium on Software Testing and Analysis. 2012: 133-143.

[4] Jacobson E R, Rosenblum N, Miller B P. Labeling Library Functions in Stripped Binaries[C]. Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2011: 1-8.

[5] Guilfanov I. Simple Type System for Program Reengineering[C]. Proceedings of the 8th Working Conference on Reverse Engineering. 2001: 357-361.

[6] Zeng J, Fu Y, Miller K A, et al. Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming[C]. Proceedings of ACM SIGSAC Conference on Computer and Communications Security. 2013: 487-498.

[7] Edward J. Schwartz, JongHyup Lee, Maverick Woo. Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring[C]. Proceedings of the USENIX Security Symposium. 2013:353-368.

[8] Jiang X, Wang X, Xu D. Stealthy Malware Detection Through Vmm-based Out-of-the-box Semantic View Reconstruction[C]. Proceedings of the 14th ACM conference on Computer and Communications Security. 2007: 128-138.

[9] Cozzie A, Stratton F, Xue H, et al. Digging for Data Structures[C]. Proceedings of the 8th USENIX conference on Operating systems design and implementation. 2008:255-266.

[10] Dolangavitt B, Leek T, Zhivich M, et al. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection[C]. Proceedings of IEEE Symposium on Security and Privacy. 2011:297-312.

[11] Fu Y, Lin Z. Space Traveling Across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection[C]. Proceedings of IEEE Symposium on Security and Privacy. 2012:586-600.

[12] Martín Abadi, Budiu M, Ligatti J. Control-flow Integrity Principles, Implementations, and Applications[J]. ACM Transactions on Information and System Security, 2009, 13(1):1-40.

[13] Schwarz B, Debray S, Andrews G, et al. Plto: A Link-time Optimizer for the Intel IA-32 Architecture[C]. Proceedings of Workshop on Binary Translation. 2001.

[14] Balakrishnan G, Reps T. Analyzing Memory Accesses in x86 Executables[C]. Proceedings of International Conference on Compiler Construction. 2004: 5-23.

[15] Lin Z, Zhang X, Xu D. Automatic Reverse Engineering of Data Structures from Binary Execution[C]. Proceedings of the 11th Annual Information Security Symposium. 2010.

[16] Lee J H, Avgerinos T, Brumley D. TIE: Principled Reverse Engineering of Types in Binary Programs[C]. Proceedings of Network and Distributed System Security Symposium. 2011.

[17] Balakrishnan G, Reps T. Divine: Discovering Variables in Executables[C]. Proceedings of International Workshop on Verification, Model Checking, and Abstract Interpretation. 2007: 1-28.

[18] Robbins E, Howe J M, King A. Theory Propagation and Rational-trees[C]. Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming. ACM, 2013: 193-204.

[19] Yan Q, McCamant S. Conservative Signed/Unsigned Type Inference for Binaries Using Minimum Cut[R]. Technical Report, Department of Computer Science and Engineering, University of Minnesota, 2014.

[20] Raychev V, Vechev M, Krause A. Predicting Program Properties from "Big Code"[C]. Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2015:111-124.

[21] Bielik P, Raychev V, Vechev M. Programming with "Big Code": Lessons, Techniques and Applications[C]. Proceedings of the 1st Summit on Advances in Programming Languages, 2015.

[22] Katz O, Ran E Y, Yahav E. Estimating Types in Binaries Using Predictive Modeling[C]. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2016:313-326.

[23] Shin E C R, Song D, Moazzezi R. Recognizing Functions in Binaries with Neural Networks[C]. Proceedings of the 25th USENIX Security Symposium. 2015: 611-626.

[24] Haller I, Slowinska A, Bos H. Mempick: High-level Data Structure Detection in c/c++ Binaries[C]. Proceedings of the 20th Working Conference on Reverse Engineering. 2013: 32-41.

[25] Noonan M, Loginov A, Cok D. Polymorphic Type Inference for Machine Code[C]. ACM Sigplan

Notices. 2016, 51(6): 27-41.

[26] Caballero J, Grieco G, Marron M, et al. ARTISTE: Automatic Generation of Hybrid Data Structure Signatures From Binary Code Executions[R]. Technical Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, 2012.

[27] Zhang M, Prakash A, Li X, et al. Identifying and Analyzing Pointer Misuses for Sophisticated Memory-corruption Exploit Diagnosis[C]. Proceedings of the Western Pharmacology Society, 2013, 47(47):46-49.

[28] ElWazeer K, Anand K, Kotha A, et al. Scalable Variable and Data Type Detection in a Binary rewriter[C]. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2013:51-60.

[29] Fokin A, Troshina K, Chernov A. Reconstruction of Class Hierarchies for Decompilation of C++ Programs[C]. Proceedings of the 14th European Conference on Software Maintenance and Reengineering. 2011:240-243.

[30] Caballero J, Lin Z. Type Inference on Executables[J]. ACM Computing Surveys, 2016, 48-65.

[31] Pierce B C. Types and programming languages[M]. MIT Press, 2002.

[32] Guide P. Intel 64 and IA-32 Architectures Software Developer's Manual[J]. Volume 3B: System Programming Guide, 2011.

[33] Robertson S. Understanding Inverse Document Frequency: On Theoretical Arguments for IDF[J]. Journal of Documentation, 2013, 60(5):503-520.

[34] Brumley D, Newsome J. Alias Analysis for Assembly[R]. Technical Report CMU-CS-06-180, School of Computer Science, Carnegie Mellon University, 2006.

[35] Xu Z, Wen C, Qin S, et al. Effective Malware Detection Based on Behaviour and Data Features[C]. Proceedings of the 2nd International Conference on Smart Computing and Communication. 2017:53-66.

[36] Ye Y, Li T, Adjeroh D, et al. A Survey on Malware Detection Using Data Mining Techniques[J]. ACM Computing Surveys, 2017, 50(3): 41.

[37] Mohamed G, Ithnin N B. Survey on Representation Techniques for Malware Detection System[J]. American Journal of Applied Sciences, 2017, 14(11):1049-1069.

[38] Pedregosa F, Gramfort A, Michel V, et al. Scikit-learn: Machine Learning in Python[J]. Journal of Machine Learning Research, 2012.

[39] Xu S. Commonly Used Algorithm Assembly (C Language Description)[M]. Tsinghua University Press, 2004.

[40] Quinlan J R. Induction of Decision Trees[J]. Machine learning, 1986, 1(1): 81-106.

[41] Liaw A, Wiener M. Classification and Regression by RandomForest[J]. R news, 2002, 2(3): 18-22.

[42] Arya S, Mount D M, Netanyahu N S, et al. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions[J]. Journal of the ACM, 1998, 45(6):891-923.

[43] Burges C J C. A Tutorial on Support Vector Machines for Pattern Recognition[J]. Data Mining and Knowledge Discovery, 1998, 2(2):121-167.

[44] Smola A J, Schölkopf B. On a Kernel-based Method for Pattern Recognition, Regression, Approximation, and Operator Inversion[J]. Algorithmica, 1998, 22(1-2): 211-231.

[45] Dai J, Guha R, Lee J. Efficient Virus Detection Using Dynamic Instruction Sequences[J]. Journal of Computers, 2009, 4(5): 405-414.

[46] Saxe J, Berlin K. Deep neural network based malware detection using two dimensional binary program features[C]. Proceedings of the 10th International Conference on Malicious and Unwanted Software. 2015:11-20.

致　　谢

在这篇文章即将完成的时候，心中充满了感慨。回首过去的三年，我在研究生学习期间学到了很多东西，得到了老师和同学们无微不至的关心和帮助，在此我衷心地对所有帮助过我的人表示感谢。

首先感谢我的导师秦胜潮教授，在生活和学习都上给了我很多帮助和鼓励，促使我不断进步。还为我创造了良好的学习和研究环境，给我提供了丰富的资源，使我能够获得国内著名专家教授的讲座授课，对我的学习和研究起了极大帮助。

特别感谢许智武老师对我的教导。从论文的选题、构思、撰写到最终的定稿，许智武老师都给了我悉心的指导和热情的帮助。这篇论文的每个篇章、每个细节都离不开许老师的细心指导。感谢许老师在百忙之中抽出时间，为我指明前行的方向，在我遇到困难时帮助我不断进步。从许老师那学到的治学精神和工作态度是我研究生期间的重要收获。

感谢我的同学们，我在学习和生活上都得到了大家的关心和帮助，让我学到了很多东西，与他们的深刻友谊是我的宝贵财富。

感谢我的父亲母亲，感谢他们给了我最无私的爱，没有他们对我学业上的支持和生活中的关心鼓励，就没有我今天的一切。他们给予的最宝贵的亲情和毫无保留的爱是我前进的动力和力量的源泉，我爱他们。

再一次感谢所有关心和帮助过我的人。

文　成

2018 年 5 月于深圳大学

攻读硕士学位期间的研究成果

[1] Zhiwu Xu, **Cheng Wen**, and Shengchao Qin. Learning Types for Binaries[C]. Proceedings of the 19th International Conference on Formal Engineering Methods, 2017:430-446.

[2] Zhiwu Xu, **Cheng Wen**, and Shengchao Qin and Zhong Ming. Effective malware detection based on Behaviour and data features[C]. Proceedings of the 2nd International Conference on Smart Computing and Communication, 2017:53-66.

[3] Zhiwu Xu, **Cheng Wen**, and Shengchao Qin. State-taint analysis for detecting resource bugs[J]. Science of Computer Programming, 2017:168-175.