申请上海交通大学工程硕士专业学位论文

基于数据流跟踪和库函数识别检测溢出攻击

学校代码：　　10248

作者姓名：　　周　侃

学　　号：　　1080379072

导　　师：　　戚正伟

学科专业：　　软件工程

答辩日期：　　2011 年 1 月 19 日

上海交通大学软件学院

2011 年 1 月

A Dissertation Submitted to Shanghai Jiao Tong University

for Master Degree of Engineering

DETECTING OVERFLOW BY COMBINING TRACKING DATA FLOW AND IDENTIFYING LIBRARY FUNCTION

University Code：	10248
Author：	Zhou Kan
Student ID :	1080379072
Mentor ：	Qi Zhengwei
Field：	Software Engineering
Date of Oral Defense：	2011/1/19

School of Software

Shanghai Jiao Tong University

Jan. 2011

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，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，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。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。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本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和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

基于数据流跟踪和库函数识别检测溢出攻击

摘 要

当前计算机已经成为日常生活中不可缺少的工具，其广泛应用在我们的生活中，有很多工具利用各种方式来进行漏洞检测或者攻击防御，但是利用各种漏洞来进行攻击的恶意行为并没有因此而减少。缓冲区溢出是一种非常常见的攻击方式。目前存在的缓冲区溢出检测工具有很多都是需要源代码的。它们虽然能够比较成功的防御攻击，检测漏洞，但是它们不适用于现在的软件程序的源代码大部分都不容易获得的情况，特别是商用软件。所以针对二进制的方法来进行程序分析是比较实用的方法。并且当前的缓冲区溢出工具大部分都是针对一种缓冲区溢出类型来工作的，那就有了很大的局限性，就不具有普遍适用的特点，人们使用起来就非常的不方便。动态二进制的数据流跟踪分析技术，就很好的解决的源代码不容易获得的问题，并且使用动态二进制数据流跟踪分析技术能够很好的解决检测类型单一这个问题。数据流跟踪将来自不安全途径的数据打标签，并跟踪观察其在内存中的行为，只要增加探测的规则就可以广泛的探测各种攻击行为，对于当前的攻击探测防御是非常好的。动态数据流跟踪分析技术对多种攻击方式都能够有效地检测到，是比较实用的攻击防御、漏洞探测方式。动态二进制检测分析的技术现在越来越流行，在各个领域显示它的优势，特别是在安全领域。

但是动态二进制数据流跟踪分析技术也有一些问题需要解决，比如在缓冲区溢出的探测方面，常常只能探测比较严重的溢出，一般只有在类似返回地址之类的重要数据被改变时才能探测到，对于一些轻微的溢出，虽然没有改变程序的控制流，但是对于程序的数据还是有影响，所以还是应该警惕，而且这种情况也说明此处能够发生溢出，这里有漏洞。我们应该注意此类情况，所以我们加入了库函数识别来增加动态二进制数据流跟踪分析的准确度。

本文引入了库函数识别到动态二进制数据流跟踪分析技术来解决提到的问题。本文构建了动态二进制数据流跟踪分析系统，并在此系统框架的基础上结合库函数识别技术，改进了数据流跟踪分析的准确性，不但能够在没有源代码的情况下探测攻击，而且比其他系统改进了准确性，并且实验验证了这些。总结为以

下三点：

在开源二进制翻译平台 DynamoRIO 上面实现动态二进制数据流跟踪分析，构建了动态二进制数据流跟踪分析系统。

引入库函数识别到动态二进制数据流跟踪分析技术，在动态二进制数据流跟踪分析系统上实现库函数识别，增强系统的准确度。

本系统在增强了准确度的基础上，同时又有比较小的性能开销。在标准的实验平台上，实验本系统，并验证了系统的功能以及性能。

关键字：动态二进制，数据流跟踪，库函数识别，攻击检测

DETECTING OVERFLOW BY COMBINING TRACKING DATA FLOW AND IDENTIFYING LIBRARY FUNCTION

ABSTRACT

Nowadays computers are widely used in almost everywhere in our life. At the same time, the number of tools to detect attack has been increasing rapidly in recent years. But the number of the attacks does not decrease. Based on the program source code, there are already some techniques and tools that can successfully detect buffer overflow. While this condition is not available in most cases, the techniques have to rely on binary to detect the malicious behaviors. And most of the existing tools can only detect one kind of attacks, which restricts the use of the tool and makes the tool not so practical. In order to address these problems, binary-based information flow tracking is introduced and applied to my work to analyze the program to detect kinds of attacks. Generally this technique labels the input data from unsafe channels as "tainted", then any data derived from the tainted data are labeled as tainted. In this way the behavior of a program can be analyzed and presented. Binary-based information flow tracking follows the malicious data, and monitors the behaviors in memory. By adding detecting kinds of rules, we can detect kinds of attacks, and discover kinds of vulnerabilities.

While the regular binary-based information flow tracking systems meet a problem when they applied to detect the malicious behaviors. When operations result in a value greater than the maximum value which causes the value to wrap-around, the overflow happens. The existing systems only concern whether the control flow is modified or not, while ignore the modifications of data flow. Thus a detection gap

comes up. Obviously not only the modifications of the control flow should be taken into account, but the ones of the data flow should be considered as well.

In our work, we did the following works.

Our work presents an effective and practical method to implement the information flow tracking. The system is implemented on binary translation platform DynamoRIO

Function recognition is introduced into information flow tracking to address the related problem. When a function is called, the function will be identified and the function's behavior will be recognized. Function recognition is implemented on the information flow tracking system to improve the system. In our work, function recognition is the strategy we applied to address the detection gap.

Our work presents an effective and practical method to enhance the information flow tracking with function recognition, and the experimental results are shown to manifest the effectiveness and practicability of our tool.

Key words： Dynamic analysis, information flow tracking, function recognition, attack detect

目 录

图目录

表目录

第一章 绪论

本章是全文的基础。本章首先阐述了本课题的研究背景与意义，之后简要介绍了本课题所涉及的背景知识以及国内外研究现状，包括缓冲区溢出、动态二进制探测技术和数据流跟踪分析技术，最后介绍了本课题的研究内容和本文的组织结构。

1.1 研究背景与意义

1.1.1 现状

随着计算机在各个领域的大量运用，计算机的各项运用越来越影响我们的社会。由于软件本身是脆弱的，现在随着软件业的发展，软件的代码量迅速增加，软件的规模越来越大，这使得软件的质量很难得到保证。安全隐患在软件中到处可见，这些漏洞使得黑客或者其他恶意用户有可乘之机，以致经济或者其他损失。同时攻击方式也从原来的简单变得越来越复杂。随着计算机技术的发展，攻击手段也趋于多样化，黑客利用各种各样的方式侵入软件，甚至一些大公司也在一些软件中嵌入了一些隐藏的功能。这些恶性行为有的侵犯了人们的隐私，有的甚至造成了经济损失或者其他的各种影响。计算机的安全问题在计算机科学中占据了重要的地位。安全问题越来越受到人们的关注，其在计算机应用中的地位也越来越高，然而漏洞检测却越来越难。

人们采用多样化的信息安全技术希望解决日益严峻的安全问题。网络、系统、设备或主机（甚至管理）中存在各种安全漏洞是产生安全攻击的根源。早期漏洞挖掘主要集中在操作系统、数据库软件和传输协议。目前有两大方向来探测漏洞。一种是用静态分析来去除漏洞，但是去掉程序所有的漏洞是不可行的[1]。一种是用动态检查数据流跟踪来预防漏洞。数据流跟踪预防的方式主要有安全语言或者非安全语言扩展[2-6]，数据流跟踪[7-10]和硬件支持[11]三种方法。但是从语言方面只是针对特定的语言，其没有动态信息，所以某些攻击无法探测。而硬件支持的方法成本很高，而且不适用与现有的系统。而基于源代码跟踪数据流[10]不容易获得，所以基于二进制的数据流跟踪是可行的方法。然而普通的基于二进制的数据流分析不能够准确的探测缓冲区溢出。

利用数据流跟踪技术来进行程序分析来进行缓冲区溢出的检测是一个很好

的解决方案。目前国际上已经有一些数据流跟踪技术。但是还有很大的发展空间。

1.1.2 目前的问题

目前程序分析分为两大方向，静态和动态。静态程序分析在很多方面有着局限性，动态程序分析借着他本身的优势，取得了很多成果，但是目前动态程序分析往往基于源代码，这就使得其实用性不足，因为源代码的获得并不易。同时单纯的基于数据流分析程序有时候又不能够做到比较准确，往往漏掉很多溢出漏洞。库函数的识别能够在很大程度上加强准确性。在这个方面还有很多工作可以做，仍然可以有比较大的发展空间。

1.2 相关理论与技术基础

1.2.1 缓冲区溢出

缓冲区[12]是指内存当中一块连续的并且有限的存储空间，高级语言中的数组就是一种典型的缓冲区。当向缓冲区当中写入过多的数据，超过了缓冲区的存储空间的边界，就会发生缓冲区溢出。缓冲区溢出[13]能够改变其相邻空间的一些数据，有时会造成程序执行异常。在过去的 20 年中，缓冲区溢出是最常见的安全漏洞[14,15]，很大一部分的攻击行为都是利用缓冲区溢出漏洞进行攻击的，很多论文中都分析了这种类型的攻击[16]。缓冲区不管是在内存的哪里都有非常可能因为数据过多而溢出到相邻的内存结构中，这样就会重写存在那里的任何内容。

缓冲区溢出往往会导致程序执行流程异常或者系统瘫痪，缓冲区溢出会改写缓冲区相邻一段空间的内容，从而被一些恶意攻击者利用对计算机系统进行恶意攻击[16]。例如，在 Linux 系统之下，恶意攻击者通常会先在内存中加入一段攻击代码，他的功能与系统调用 execve 类似，之后攻击者会选择攻击一个以管理员权限(root 权限)运行的程序，就可以通过缓冲区溢出将地址相关的程序状态改变成为上述攻击代码的首地址，从而使程序转向执行这段恶意攻击代码，获得以管理员权限来运行的命令解释脚本。此时恶意攻击者就可以进行任何特权操作，窃取计算机系统的有用数据。

究其根本造成缓冲区溢出的原因[13]，在于程序在内存的使用上没有边界保护机制。这往往被用来重写一个代码指针，改变控制流，或者直接把代码指针溢

出，或者先溢出另外一个指针然后是这个指针指向代码指针。如果攻击者能够向缓冲区输入内容，那么他就可以引导返回值指向到攻击代码。造成缓冲区溢出的具体原因则多种多样，包括：编程语言没有边界检测机制；编译器没有提供边界检测机制；用户为了追求性能在编写程序时也没有进行数组边界的检查等等。

利用缓冲区溢出可以篡改多种程序状态，最为常见的可以归为以下三类。

(1) 攻击函数的返回地址[17]

程序进行调用函数时，会在栈空间上面记录函数的返回地址，当被调用的函数执行完毕时，程序会根据函数的返回地址继续执行调用函数之后代码。被调用函数当中的局部变量也同样是分布在程序的栈空间上，和程序返回地址是十分接近的。如果被调用的函数当中包括局部的数组变量，恶意攻击者可以利用本地数组溢出覆写程序的返回地址，从而改变程序执行流程，使其转向恶意攻击者事先准备的攻击代码。篡改函数的返回地址的这种攻击占据了所有利用缓冲区溢出漏洞的攻击的很大一部分的比例。

(2) 攻击函数指针[18]

函数指针是一个指针变量，它指向函数，内容是函数体可执行代码的起始地址，可以被用来调用函数。函数指针可以分布在程序的堆空间、栈空间、静态数据区域上。攻击者需要在函数指针附近找到一个可能溢出的缓冲区并加以利用，从而篡改函数指针的内容，即被调用函数的地址。这个缓冲区通常是与函数指针具有相同作用域的数组变量。在函数指针的内容被篡改后，程序通过函数指针调用函数时，程序计数器将指向被篡改后的地址，从而改变程序的执行流程，转向执行攻击代码。

(3) 攻击长跳转缓冲区

在 C 语言中包含了一个简单的检验/恢复系统，称为 setjmp/longjump。意思是在检验点设定“setjmp(buffer)”，用 “longjump(buffer)”来恢复检验点。然而，如果攻击者能够进入缓冲区的空间，那么“longjump(buffer)”实际上是跳转到攻击者的代码。就像函数指针一样，longjump 缓冲区能够指向任何地方，所以攻击者所要做的就是找到一个可供溢出的缓冲区。一个典型的例子就是 Perl5.003 的缓冲区溢出漏洞；攻击者首先进入用来恢复缓冲区溢出的 longjump 缓冲区，然后诱导进入恢复模式，这样就使 Perl 的解释器跳转到攻击代码上了。

1.2.2 数据流跟踪分析技术

a) 动态二进制探测框架

探测[19]是一种在程序中加入额外代码来收集程序行为信息的方法，被加入程序中的额外代码称为探测代码。探测代码可以是高级语言，也可以是二进制代码，其主要功能是收集程序行为信息，例如指令的执行次数、内存操作信息等。探测技术可以在程序的源代码开发阶段，编译阶段，链接阶段和运行阶段使用。

动态二进制探测是指在程序的运行时动态地加入二进制探测代码的一种方法。目前，动态二进制探测作为一种常见的程序行为分析方法，在查找程序漏洞、验证程序安全性等方面有着广泛的应用。应用动态二进制探测方法的程序行为分析工具通常称为动态二进制探测工具。相比于传统的程序行为分析方法，动态二进制探测方法有两个优点：

1) 动态二进制探测方法作用在二进制代码的级别[8]，不需要源代码的支持，因此能够很好地应用于遗留系统和商用软件。

2) 相比于静态方法[20]，动态二进制探测方法作用在程序运行时，可以覆盖到所有被执行的客户代码，同时可以很容易地区分程序的数据和代码。

在程序行为分析领域，程序的安全性是十分重要的一部分内容[16]。目前有许多分析程序安全性能的动态二进制探测工具。动态二进制探测工具通常以动态二进制翻译为基础，因此实现上存在工作量大、复杂度高、开发周期长等特点。但是程序安全性的需求是多种多样的，如果针对不同的程序安全性需求分别实现一个动态二进制探测工具是无法令人接受的。如何以较少的资源在较短的时间内实现一个动态二进制探测工具，就成为了研究的热点。在这种情况，动态二进制探测系统也就应运而生。

动态二进制探测框架存在的前提是大多数动态二进制探测工具都以动态二进制翻译为基础，因此它们包含着许多共通的部分，而动态二进制探测框架正是这些共通部分的集合。动态二进制探测框架最少包含一个动态二进制翻译模块和一个探测模块，还可能包含内存管理模块、代码缓存模块、优化模块等，为动态二进制探测工具的开发提供一系列的基本功能[7]。在动态二进制探测框架的基础上，工具开发者只需要实现探测工具的特殊功能，就可以便捷的构建一个动态二进制探测工具，缩短了开发时间，降低了开发难度，提高开发效率。

b) 动态二进制的数据流跟踪分析技术

数据流跟踪分析[9]是一种在程序安全性检测领域常见的技术。它通过对程序中使用的数据进行分析，将程序的数据标记为“被污染的”（Tainted）和“未被污染的”（UnTainted）两类，同时在程序的执行过程中控制污染属性的传播，当非法使用被污染的数据时，即可断言攻击行为的存在；同时，通过分析非法使用数据污染属性的传播路径，可以发现程序的漏洞所在[21]。这里的数据是广义上的数据，包括可执行文件的代码、操作数等所有内容。

数据流跟踪分析可以分为静态数据流跟踪分析[20]和动态数据流跟踪分析[9]两类。静态数据流跟踪分析是在程序执行之前进行的，它将数据流跟踪过程嵌入到类型分析中，将污染信息视为扩展的类型信息，通过编译器的类型分析，获得丰富的类型信息[22]，从而在类型推断和类型限定中完成污染属性传播的功能，达到分析程序安全性的目的。这种方法利用编译器类型信息，分析较复杂，需要较强的类型分析和推断算法，因为无法确定程序的执行流程。

动态数据流跟踪分析是在程序执行的过程中进行的，它通过跟踪变量、存储单元、寄存器的值，并依据执行流程跟踪数据污染信息的传播，检测污染数据的非法使用，从而达到跟踪攻击路径、获取漏洞信息的目的。动态数据流跟踪分析能够准确获得程序的执行流程，有效提高了数据流跟踪分析的精确度，错误率较低，在组织攻击的同时获得程序的漏洞所在，应用性较强。

数据流跟踪分析的过程一般可以分为三个步骤[22]：标记“被污染的”数据，污染属性的传播，“被污染的”数据非法应用的判定。

数据流跟踪分析可以针对源代码中的变量、数据，也可以针对二进制指令中的操作数。本文的研究方向是在动态二进制探测的基础上，结合数据流跟踪分析的技术，检测程序缓冲区溢出漏洞，防御缓冲区溢出的攻击。在后面章节所提到的数据流跟踪分析中，主要是指针对二进制代码的数据流跟踪分析分析技术。

1.3 国内外研究现状

1.3.1 缓冲区溢出

缓冲区溢出的检测和防御在二十世纪就是年代就受到了广泛的关注，各种各样的防御方法层出不穷。缓冲区溢出防御技术主要分为静态防御技术和动态防御技术[23]。静态防御技术是指在源程序编译的不同阶段，依据静态库进行程序更

改的代码修正技术，它需要高级语言源代码的支持；动态防御技术是对应用软件的运行环境进行修正的上下文检测和更改技术。

静态防御技术针对编译不同阶段，通过模式匹配和静态库进行防御。典型防御技术包括词法分析阶段防御的 ITS[24]词法分析工具，它是利用一个包含潜在不安全函数的数据库（如：strcpy/strcat/gets/fgets 等）来搜索安全问题。该工具实现比较简单、检测速度快，但是没有考虑语法和语义，检测效果差。LCLint[25]和 SPLinit 是在语义分析阶段进行防御的语义注解工具，其通过向源代码和标准库的源代码中添加语义注解，进行局部的、过程内属性检查，用注解来辅助静态分析，从而快速有效的检测安全漏洞。语义注解方案简单、高效、快速、可扩充，它可以简单精确的描述程序员对程序的意图和假定，但该工具受程序员分析方法的影响，分析是不可靠的。

动态防御技术多数借助运行库的支持在程序运行阶段进行部署防御。动态防御技术主要分成 3 类：地址保护、堆栈不可执行以及边界检查[23]。

(1) 地址保护方案指的是为防止地址和状态信息，尤其是可执行代码的地址信息被恶意修改而设置的防御方法。典型的防御包括在函数地址附近设置效验值的 Stack-Guard[26]方法、对地址进行加密的 PointGuard[27]方法以及对保存返回地址副本的 RAD[28]方法。但是，攻击者在不破坏效验值的情况下仍能修改地址，或者生成一个有效的效验值来修改地址；此外，PointGuard 在对指针地址进行加密操作时需要源代码的支持，并且在保护加密密钥和加密算法时存在一些问题。地址保护方案仅仅是削弱了攻击的可能。

(2) 考虑到恶意攻击者在内存空间中注入的攻击代码通常都是在程序的堆和栈上，设置堆栈不可执行可以阻止攻击者注入的攻击代码的执行。但是该方法并不能够阻止 Return-into-lib 攻击，并且阻止 trampolines 通道的正常执行，存在一定的兼容性问题。

(3) 边界检查方案从指针中引入一个基指针，一个指针值仅对一个内存区域的引用是有效的。每次指针引用利用基指针强行进行边界检查，检查引用范围是否在相同的区域内。边界检查技术虽然是在程序运行的过程当中检测缓冲区溢出，但是需要缓冲区边界相关信息的支持[22]，这些信息要么以解释的方式存在于源代码中，要么在编译阶段生成，而目前除了特定的编译器，一般的编译器都不支持在二进制代码中嵌入边界检查信息和代码，因为边界检查将会带来性能上的严重损失。因此，边界检查技术需要源代码的支持。现在，安全语言诸如 Java、C#，也会提供数组

边界检查等保护机制，但是提供这种保护机制的平台——虚拟机，仍然存在很多不安全的因素，所以这些安全语言只是减少了被攻击的可能性。除此以外，大量的遗留代码仍然使用 C、C++等不安全的语言编写的[9]。一方面，重新编写这些不安全的代码的工作量十分大；另一方面，用户无法获得一些遗留系统或商业软件的源代码，从而也无法重新编写这些软件。因此，动态防御技术中不需要源代码支持的技术，例如地址保护、堆栈不可执行等，更容易得到广泛的应用。下面将具体介绍国内外比较有代表性的缓冲区溢出检测工具。

在缓冲区溢出检测方面，国内外具有代表性的工具包括：Stack-Guard[26]，Stack Shield[29]，ProPolice[30]，CCured[31]，Chaperon[32]，Libsafe and Libverify[33,34]，Memcheck[35]等。

- Stack-Guard[26]

Stack-Guard 是对编译器 gcc 的扩展，它需要源代码的支持，但是不需要修改源代码。在 Stack-Guard 的支持下重新编译源代码得到的可执行程序能够很好的阻止利用栈空间上缓冲区溢出的攻击行为。它的主要思想就是缓冲区溢出攻击改写指向他们目标的任何东西。目标是返回值的缓冲区溢出为例，攻击者会将缓冲区填满，然后改写下面的任意的本地变量，然后改写旧的基指针直到达到返回值。如果放置一个测试值在返回值到栈值之间，然后在使用这个返回值之前检查这个值是否被改写了，我们就可以探测到这类的攻击并且预防它。

针对修改栈空间上函数返回地址的攻击，Stack-Guard 给出两种方式阻止攻击：1.在函数返回前检测函数返回地址是否被修改；2.阻止任何程序对函数返回地址的修改。第一种方法更加高效并且更容易移植，第二种方法则更加安全。对于第一种方法，Stack-Guard 在函数返回地址下一个字节中放入已知的安全字(Canary Word)，在函数返回时检测安全字是否被修改。如果安全字被修改，则认为函数返回地址收到攻击，因为考虑到缓冲区溢出在空间上的连续性，通常情况下利用栈空间上的缓冲区溢出在修改函数返回地址之前，会先修改安全字。但是也存在例外情况，比如拷贝到缓冲区中的内容中有内存漏洞，或者攻击者可以很容易的猜出安全字。针对这些问题，Stack-Guard 在每一次程序运行的时候都随机生成一系列的安全字以供使用，防止攻击者因猜测出安全字而绕过 Stack-Guard 的检测方法。对于第二种方法，Stack-Guard 在 MemGuard 的基础上进行扩展，使得函数返回地址不可写，或者产生一个中断使得程序能够扑捉到对函数返回地址的修改，从而阻止了利用修改函数返回地址的攻击。

数据表明，Stack-Guard 可以有效的阻止栈空间上的缓冲区溢出，但是他需

要源代码的支持；如果攻击者知道其采用的防御方法，仍然有方法进行攻击。除此之外，Stack-Guard 只能检测修改函数返回地址的攻击行为，不适用于其他类型的攻击手段。这种技术只适用于防止顺着栈改写所有东西的攻击。攻击者还是可以使用指针使其指向返回值，然后写一个新的地址到这个地址中。这个缺点被 Mariusz Woloszyn 发现，并被 Bulba 和 Kil3er 提出。然后 StackGuard 的团队用同时保存 canary 值和 canary 和正确返回地址的 XOR 值来解决这个问题。这样即使 canary 值未被改变，但是 XOR 值的改变会被探测到。如果 XOR 模式要使用那么 canary 需要时随机的。但是 StackGuard 不支持随机 canary。

- Stack Shield[29]

Stack Shield 是由 Vendicator 给 GCC 的一个编译器的补丁。他实现了三种保护，两种是用来防止返回地址的重写，另外一种是用来防止函数指针的重写。

函数指针应该指向文本段。如果进程确定没有函数指针指向其他段，那么攻击者就不能指向注入的代码。Stack Shield 在所有的函数调用前增加了代码检查。在数据段定义一个全局变量，它的地址被用作界限。检查函数确认任何被废弃的函数指针都在这个地址之下。

Global Ret Stack 是一个单独的栈用来保存被调用的函数的返回地址。如果攻击者改写了普通的返回值，那么通过比较 Global Ret Stack 和普通的栈攻击就会被发现。

Ret Range Check 用一个全局变量来存储当前函数的返回值。在返回前，比较全局变量的值和返回值，如果不同停止执行。

- ProPolice[30]

ProPolice 的主题思想和 StackGuard 很相似，他们也是使用 canary 值来检测堆栈攻击。他们的创新之处在于他们重新安排本地变量使得 char 缓冲区一直是在顶部来保护栈，这样其他的本地变量就不会被溢出破坏。

- CCured[31]

CCured 是 C 中加强类型及边界检查的研究，他们都是采用动静结合方式。CCured 是一个 C 语言的延伸，它基于他们的使用区别这种指针。这种辨别的目的在于防止指针的不恰当时使用，并且可以防止程序进入其不应该进入的内存区域。

CCured 通过静态分析的方法决定指针的类型：SAFE，SEQ，WILD。不同类型的指针所能够执行的操作不同。CCured 用适当的元数据标识每一个指针：

SEQ 类型的指针会包括它所指向区域的上下界，WILD 指针则会包含类型标签。根据这些元数据，会在可执行文件中插入适当的检测代码。CCured 作为一个基于编译器的缓冲区溢出检测工具，需要源代码的支持，这限制了其引用的范围。

- Chaperon[32]

Chaperon 是由 Parasoft 公司开发的安全检测工具之一。Chaperon 不需要源代码的支持，针对二进制代码检测缓冲区溢出。它可以截取 malloc 和 free 系统调用，检测堆访问的合法性；并且可以检测出堆上的内存泄漏；以及对未初始化内存的访问。相比于堆上的缓冲区溢出检测，Chaperon 对栈上的缓冲区溢出检测比较简单粗糙。探测代码有时也会因为缓冲区溢出而被覆盖，导致最后给出错误的缓冲区溢出信息。Chaperon 作为一个商用软件，源代码不公开，也无法对其进行扩展和改进。

- Libsafe and Libverify[33,34]

Libsafe 实际上是一个动静结合的工具。静态的方面，他为 C 中的库函数中一些潜在的缓冲区溢出漏洞打补丁。在实际的函数调用前做一个边界检查可以保证返回值和基指针不被重写。之后 Libverify 使用一个和 StackGuard 类似的动态方法来增强保护。

- Memcheck[35]

Memcheck 是在 Valgrind[]基础上构建的内存探测工具。Valgrind 作为一个开源的动态二进制探测框架，其关于内存检测的核心技术是“影子内存”。Memcheck 通过为内存的每个比特位建立影子内存从而更准确的发现比特级别的内存错误。对每一次内存操作 Memcheck 都需要进行探测并对影子内存进行相应的修改。由此带来的是性能上的严重损失，通常会比正常运行程序慢 20-30 倍。Annelid 是在 Valgrind 基础上构建的类似指针边界检查工具。

总结

这些工具或者说技术都能够探测到一些攻击，但是他们都是针对特定的某些类型的软件漏洞设计的，所以对于那些未知的软件漏洞他们都不能够有效的探测出来。上述的五种工具像 LibSafe 就能够探测 18 种预先设计的漏洞类型，但是把他们所有的工具都加起来仍然会有 30%的攻击不能被覆盖到。

1.3.2 数据流跟踪分析技术

a) 动态二进制探测框架

关于动态二进制探测框架的研究在 2002 年之后才开始逐渐兴起，之前更多的是静态的二进制探测工具或基于源代码级别的探测工具。比较著名的动态二进制探测框架包括利用数据流跟踪技术来进行程序分析对于漏洞挖掘是一个很好的解决方案。目前国际上已经有一些数据流跟踪技术，有很多二进制动态分析工具，比如 Dynamo [36]，DynamoRIO [37]，Pin [38]，PinOS[39]，Valgrind[40]，StarDBT[41]等等。他们可以用来 profiling，改善可靠性和探测软件错误。也有很多工具用来进行反编译，像 IDA Pro。我们主要介绍几个工具。

- Valgrind[40]

Valgrind 是一个开源的动态二进制探测框架，适合用来构建重量级动态二进制探测工具。它采用动态二进制翻译的方法，将 x86 指令翻译成一种 RISC 结构的中间代码，并加入探测代码，之后再将中间代码翻译成为目标平台的二进制代码，加以执行，从而达到分析程序行为的目的。Valgrind 的核心技术是“影子内存”(Shadow Memory)。“影子内存”是指用额外的数据记录每个寄存器和正在使用的内存值的相关信息，并可以提供精确到比特位上的“影子值”(Shadow Value)。一系列的程序行为分析工具在 Valgrind 基础上构建起来，包括 Memcheck[37]，Taintcheck[9]，Hobbes[42]，Redux[43]等。由于 Valgrind“影子内存”的特性，使得 Valgrind 在性能上的损失较大。

- DynamoRIO[37]

DynamoRIO 是由 Dynamo 扩展而成的动态二进制探测框架。Dynamo[36]是一个动态二进制优化器，DynamoRIO 在 Dynamo 的基础上加入了探测功能，形成现在的动态二进制优化及探测框架，目前支持 x86/Linux 和 x86/Xin32 平台的客户程序。DynamoRIO 提供了丰富的 API 接口，便于工具开发者修改客户程序二进制代码或加入分析代码。它通过在内存中保存寄存器中的拷贝，避免客户代码和分析代码的冲突。但与 Valgrind 不同的是何时将寄存器值拷贝到内存中需要由工具开发者来决定。DynamoRIO 的 API 的实现是平台相关的，它与 x86 体系结构紧密相关。基于 DynamoRIO 构建的工具包括一些优化器以及轻量级的动态二进制工具，例如检测客户程序中跳转指令的地址是否安全等。

- Pin[38]

Pin 是 Intel 公司开发的一个动态二进制探测框架，具有易用、透明、高效、可移植、健壮等特点，目前支持 IA64/Linux、x86／Linux、x86-64／Linux、ARM／Linux 平台的客户程序。Pin 的易用性体现在 Pin 提供了丰富的 API，工具开发者不需要了解客户代码指令集的细节知识，也不需要了解探测框架的实现细节，就可以轻松构建各种探测工具。Pin 的 API 设计成平台无关，使用 Pin 构建的程序行为工具具有很好的兼容性和移植性。Pin 使用动态编译的方法插入探测代码，配合使用函数内联、寄存器重分配、指令调度等优化方法，使得基于 Pin 开发的动态二进制探测工具运行效率高。例如，使用 Pin 统计程序的基本块比 Valgrind 快 3.3 倍，比 DynamoRIO 快 2 倍。基于 Pin 开发的动态二进制探测工具不会改变源程序的执行结果，搜集的程序信息只涉及客户程序，对用户来说具有透明性。

- PinOS[39]

PinOS 是在 Pin 的基础上进行扩展的动态二进制探测框架。PinOS 是在虚拟机 Xen 和 Intel 虚拟技术（Intel VT technology）基础上构建的。它作用于硬件和操作系统之间，因此 PinOS 能够对整个系统的代码进行探测，包括用户空间的代码和内核空间的代码，这也是 PinOS 的特点。PinOS 继承了 Pin 丰富的 API 接口，并引入了系统级的探测接口，使得工具开发者可以便捷地构建探测系统级程序的动态二进制探测工具。

b) 基于二进制的数据流跟踪分析

数据流跟踪技术是一个比较可信有效的漏洞探测技术。总体来说，数据流跟踪技术就是将来自不安全渠道（比如一些网络连接）的输入信息标记为“不安全”，然后通过计算来将这些数据标记进行传播，也就是从不安全数据得来的数据也会被标记为不安全[21]。然后探测不安全数据的不当使用，即当不安全数据会将程序的控制权导向不安全的数据。同时，数据流跟踪技术也能够用来探测数据泄漏。

目前，数据流跟踪技术已经用三种不同的方式实现。第一种是为特定的编程语言在编译时进行跟踪数据流[2-6]。这种方法能够在没有实时开销的情况下让程序符合数据流安全策略，但是这种只是适用于特定的语言。更重要的是，因为几乎没有实时信息，大部分这种类型的工具都是被设计用来探测敏感数据的泄漏问题而不是用来探测安全攻击。比如如果要探测改变间接分支的目标这种攻击，没有实时信息是不行的，这种方式就很难探测此类攻击。

第二类方法就是通过源代码[10]或者二进制代码[7-9]跟踪数据流并且探测恶意行为。使用源代码的方式相较二进制代码的方式有比较小的开销，但是不能跟踪

到库代码中，这有可能就漏掉了一些。相比较而言，基于二进制代码的能够准确的跟踪库函数中的数据流，但是他却有一个很明显的缺点，就是开销问题，大概要减慢 37 倍，这个开销太大以致用来探测攻击是很不实用的。

第三种方法是使用硬件支持[11]。这种方法已经在一些前期工作中实现，这种方法在对于用低开销有效地探测攻击是不错的。但是这种方法需要硬件扩展。因此这种方法的成本就比较高，并且不适用于现有系统。

下面介绍几个现有的系统。

Taintcheck[22]是一个基于 Valgrind 的数据流跟踪工具，它通过实时重写二进制代码来探测重写。但是他的开销大概有 30 几倍。同时他的检测局限在重写函数的探测。

CCured 和 Stack-Guard 在上面已经介绍过。

GIFT[22]为 C 程序提供了一个数据流跟踪的框架。GIFT 通过控制并且检查标签来进行数据流跟踪。

LIFT[44]是一个基于二进制代码的低耗的数据流跟踪工具。他通过过滤掉一些数据流无关的数据提高了效率，同时又不会降低其准确性。

1.4 本课题研究内容

图 1-1 中通过一个例子来说明缓冲区溢出的原理。现在动态二进制数据流跟踪分析技术被用来解决一些安全问题，比如探测一些恶意攻击行为。同时本图说明当前大部分传统工具与本文工具之间的探测鸿沟。在 strHandling 函数中，缓冲区的大小比复制到此缓冲区的字符串要小，这时就产生了所谓的缓冲区溢出。在传统的方法中，只有 ebp 值或者返回值被改变时才会被探测到。通常的数据流跟踪分析系统只关注控制流是否被改变，而忽略了数据流是否被改变，这是就出现了图中所示的探测鸿沟。图中是一个缓冲区溢出的例子，这个例子是使用 C 代码来表示的，这只是为了说明工具是如何工作的，本文的工作是基于二进制代码进行的。

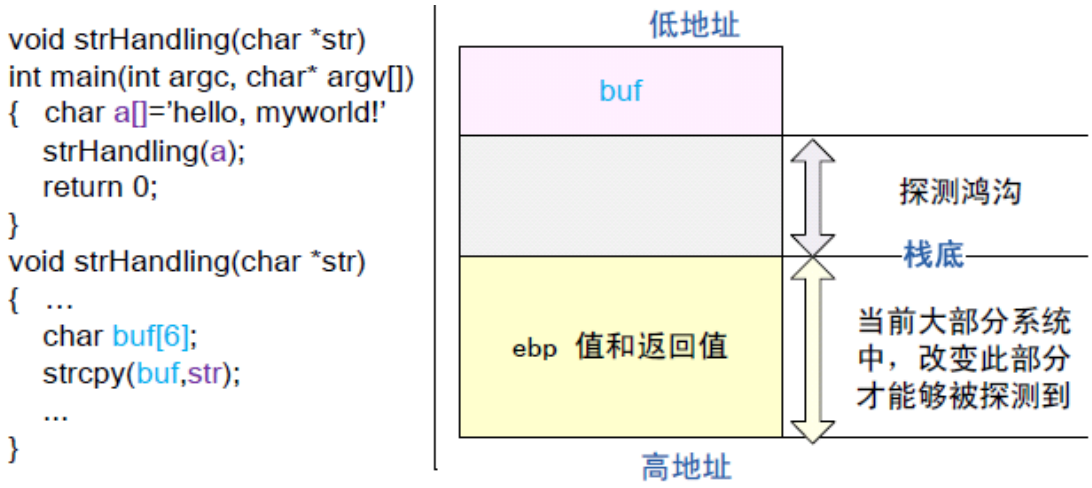

图 1-1 缓冲区溢出例子。

Fig.1-1 Example of buffer overflow.

本课题调研现有的缓冲区溢出技术以及动态二进制数据流跟踪分析技术，尝试设计开发一个有效并且实用的漏洞探测框架，并在该框架的基础上加入函数识别功能，使该框架在有效使用的基础上增加其精确度，以期帮助程序员或者普通用户查找并定位缓冲区溢出漏洞，提高程序的安全性。具体的研究内容包括以下的四个部分：

(1) 调研并分析缓冲区溢出检测技术以及动态二进制探测框架的背景知识、研究现状和特点；

(2) 设计一个动态二进制数据流跟踪分析系统，并在开源二进制翻译平台 DynamoRIO 的基础上，实现动态二进制数据流跟踪分析工具，基本可以用以防御利用缓冲区溢出的明显的攻击行为。

(3) 引入库函数识别的技术，在上述动态二进制数据流跟踪分析系统框架的基础上加入实现库函数识别，用以增强缓冲区溢出的准确度。

(4) 分析并评估动态二进制数据流跟踪分析系统以及库函数识别的功能和特点。

1.5 论文组织结构

论文共分为六章，本文具体章节组织结构安排方式如下：

第一章，将课题研究的背景以及大概的内容作阐述。

第二章，首先简单的介绍了数据流跟踪分析技术的工作原理，并对其优势与不足就行了分析。之后对动态二进制数据流跟踪分析系统进行了介绍，包括动态二进制数据流跟踪分析系统的系统架构、实现平台、关键技术模块等，并对本系统进行了性能分析。

第三章，首先简单介绍了库函数识别技术，之后介绍了利用库函数识别技术的动态二进制数据流跟踪分析工具 FRDIFTS 的设计思路和实现。

第四章，利用第三章中的 FRDIFTS 系统进行一些功能和性能上的实验，并分析结果，以确定其功能上的优势与不足，并给出结论。

第五章，结对全文的工作进行了总结，并指出了 FRDIFTS 的优势与不足之处，并且对未来工作进行了展望。

第二章 数据流跟踪分析系统设计与实现

第二章首先简单介绍了动态二进制数据流跟踪分析技术的原理以及优缺点，然后介绍了数据流跟踪分析系统的设计思路，平台以及性能。

2.1 动态二进制数据流跟踪分析技术的原理以及优缺点[45]

在前面我们提到在各种攻击方式中，恶意攻击者为了实施恶意行为必须改变程序执行的正常的流程，使得程序来执行非法的恶意代码。为达到这一目的，攻击者通常采用重新赋值某一些敏感数据，比如返回地址，跳转地址，而这些数据原本是在程序编译结束后就已经确定的。攻击者常常通过用重新覆盖这些敏感数据来达到恶意攻击的目的。缓冲区溢出在绪论中已给出例子和解释，其主要通过向缓冲区中写入过多的数据从而覆盖缓冲区周围相邻的内存空间中的数据，是一个非常常用的覆盖敏感数据的手段。那么也就是说对于程序来说，来自用户的数据有一部分数据值是不安全的，我们将来自外部的不安全数据称作“污染源”。数据流跟踪分析正是利用恶意攻击的重新赋值敏感数据的特性来防御攻击的。

传统的缓冲区溢出检测方法有一些局限性，相比于传统的缓冲区溢出检测方法，数据流跟踪分析的方法具有以下几个优点：

(1) 动态二进制数据流跟踪分析的方法可以检测出多种形式的漏洞的攻击，具有很大的普遍性。现存的大部分缓冲区溢出检测工具，特别是二进制工具，有很大一部分都只能针对一些特殊特定类型的溢出形式来进行处理。比如说 Stack-Guard 虽然可以有效的检测栈上面的的溢出，比较擅长的检测的是利用缓冲区溢出来改写函数返回地址的这种恶意攻击，但是它却不能够检测堆上的缓冲区溢出。而数据流跟踪分析的方法则是维护整个用户的内存以及寄存器的污染属性状态，所以它能够有效地检测出缓冲区溢出。

(2) 动态二进制数据流跟踪分析的方法还可以检测出一些利用其他漏洞的恶意攻击行为。大部分攻击的关键就在于它改变了程序正常的执行流程，我们通常将这个成为控制流，数据流跟踪分析的方法就是根据是否非法的改变了程序的控制流来断言是否发生了攻击。所以，从理论上来讲，除了检测利缓冲区溢出漏洞的攻击以外，数据流跟踪分析还可以检测到

其他多种不同特征的恶意攻击。

(3) 动态二进制数据流跟踪分析可以确定“污染源”的位置和攻击在内存和寄存器的传播过程，这就可以为防御攻击，并从根本上消除漏洞提供了很多有用的信息。数据流跟踪分析的方法可以很精确的定位“污染源”的来源，恶意攻击的发生地点，在局部范围之内甚至还可以追踪到污染属性传播的路径。这些信息都可以直接或者间接的帮助定位漏洞，为消除漏洞提供很多有用的信息和帮助。

(4) 动态二进制数据流跟踪分析的方法可以直接针对动态二进制代码，不需要源代码的支持，也不需要对源代码进行重新编译。在绪论的介绍中我们能够看出，通常的缓冲区溢出检测工具很多都需要源代码的支持，这些工具所采用的方式要么是对源代码进行分析；要么就是重新编译源代码，在可执行文件中当中加入边界检查的代码。对于无法获得源代码的遗留系统或者是商用软件来说，这些方法都是不实用的。

通过以上的分析明显的，相较现有的传统缓冲区溢出防御工具，数据流跟踪分析不需要源代码的支持，并且能够多元化的防御缓冲区溢出攻击，以及其他类型的攻击，在程序分析方面特别是程序安全方面具有很强的优势。

当然数据流跟踪分析也有一些缺点存在，具体包括以下三点。

(1) 数据流跟踪分析的技术，只能作为防止恶意攻击的攻击防御工具，却不能作为漏洞的发现检测工具。数据流跟踪分析技术在非法的使用“被污染的”数据的情况之下才会判定是发生了攻击行为，只有在这样情况之下，才能够将安全漏洞给暴露出来。同样的道理，对于那些没有检测到恶意攻击行为的程序来讲，也并不能够说明程序当中不存在某个安全漏洞。并且一些由于编程人员疏忽而导致的安全漏洞，数据流跟踪分析的技术也没有办法预防由于这些而可能产生的严重后果，比如程序崩溃等等。

(2) 数据流跟踪分析的方法一般都需要维护整个内存以及寄存器的污染属性，这就需要占用非常大的内存，这个问题在目前的数据流跟踪分析工具当中是普遍存在的，这个问题也是目前比较难解决的。

(3) 大部分数据流跟踪分析工具由于需要维护对污染属性大量的访存操作，所以都会有性能上面的瓶颈。本文中提出一些优化手段来尽量降低性能上的开销。

针对内存占用以及性能上面的问题，存在着一些优化的方法，在本节中将会

提到。

2.2 系统框架

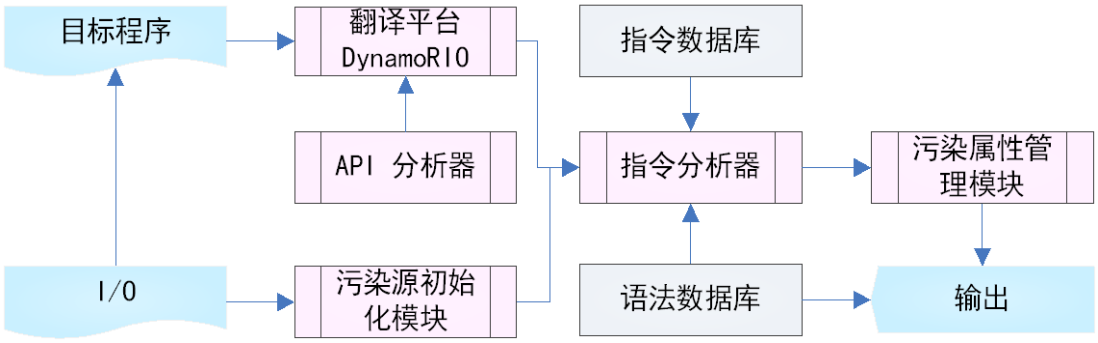

图 2-1 数据流跟踪分析框架

Fig. 2-1 Architecture of Dynamic Information Flow Tracking System.

在图 2-1 给出我们的数据流跟踪分析系统的框架。这个框架主要介绍了各个模块之间的交互，以及运作方式。其中“污染源初始化模块”是用来初始化其他模块并启动系统。“指令分析模块”是用来分析污染的传播以及消亡，是指令级别的数据流跟踪，并且其与“污染属性管理模块”交互来查询或者设置某块内存或者寄存器的污染属性。API 分析器是借助二进制翻译平台做一些 API 的工作。“语法数据库”用来存放漏洞探测规则。为了效率优化的目的，使用不相关 API 过滤来过滤掉一些不相关的 API。此系统框架中的二进制翻译平台我们选用 DynamoRIO 来完成对目标程序数据流跟踪分析并与其他模块交互。

2.2.1 污染源初始化模块

污染源初始化模块的主要作用是用来确定“污染源”（Taint Source）的位置，为动态二进制翻译平台准备初始化环境，包括数据结构的初始化、探测环境的准备工作等等。在数据流跟踪分析系统中，“污染源”被定义为来自不安全的文件的不被信任的数据。在数据流跟踪分析中，所有的源自不可信的数据源头或者不可信的数据的操作之后得出的结果都应带被标记为“被污染的”（Tainted）。在这里的不可信的数据源头就是“污染源”。在通常情况下，相对于程序本身包含的数据来讲，来自于程序外部的输入就会被认为是不可信的数据，包括读入的文件、键盘的输入的数据等。当一个来自外界污染的目标文件被打开时，其文件的内容就被映射到内存中。在我们的数据流跟踪分析系统中，将这些被映射到的内存定

义为“污染源”。污染源初始化模块用来查看污染源内存的起始地址以及所占大小，并且之后通知污染属性管理模块将这块内存标志为“被污染的”。

考虑到扩展性，本文中“污染源”是可以手动的标记某一块内存或者寄存器作为污染源，并且污染源初始化模块会查找这部分的地址。在污染源被初始化之后，污染源初始化模块会打开开关，然后数据流跟踪分析(Information Flow Tracking)模块就开始工作了。

2.2.2 数据流跟踪

指令分析器和 API 分析器在二进制翻译平台上共同完成数据流的跟踪这一功能。在程序运行的过程当中，假如机器指令的操作数当中包含被污染的操作数，那么得到的结果也就很可能是被污染的数据，我们称此过程为污染属性的传播。程序当中的二进制指令大致都分为三类，大致就是以下三类：算术逻辑运算指令，访存指令，控制转移指令。其中访存指令包括存操作数（Store）、取操作数（Lord），算术逻辑运算指令当中包括逻辑运算指令（与或非等）、算术运算指令（加减乘除等)以及移位运算指令等等，控制转移指令主要包含跳转指令、分支指令等等。会引起污染属性传播的指令主要集中在访存指令以及算术逻辑运算指令。

大部分的二进制指令的操作数都是存储在寄存器当中的，因此在数据流跟踪分析时不仅仅需要维护内存当中数据的污染属性，还需要维护寄存器当中数据的污染属性。污染可以内存与寄存器之间分别传播，也可以在内存和寄存器之间传播。

本文中数据流跟踪模块是由两层实现的，第一层是给予指令的，也就是指令分析器完成的，和其他系统相类似，目前我们不考虑 SSE/MMX 指令，同时 Ring0 指令也不考虑。剩下的指令被分为三组。一类我们将其称之为传播类，它们会传播污染，如 mov 指令、push/pop 指令。另一类称为净化类，它们会将污染的数据净化。另外的指令被归类为第三类，它们完全不会影响数据流跟踪分析的过程，我们称其为无关类，比如 cmp 或者 test 指令，我们在分析时就跳过这些指令。

还有一种很重要的情况也要考虑。如下的例子：

```
mov     r32, r/m32/imm32
xor     eax,    eax
```

第一条 mov 指令，我们不关心源操作数如何复杂。其中源操作数被分为通

用寄存器，内存和立即数。立即数如果被用在源操作数时那么接过奖杯认为是“未被污染的”，然而通用寄存器或者内存被用作源操作数时会将污染属性传播。在第二条 IA-32 指令中，不管 eax 的状态是什么，都会被设置为“未被污染的”，所以我们将认为在这条指令之后 eax 不包含污染信息。

第二层是基于 API（Application Programming Interface,应用程序编程接口）以及库函数，主要是 API 分析器完成的。数据流跟踪分析应该基于这层有两个原因。一个原因是如果只有指令级别的分析的话，一些不是直接污染的情况就不会被发现，因为非直接污染的情况只是直接的数据转换的副产品。Hooks 会被设置在这些函数中来获得返回值。第二个原因主要是为了优化，快速的过滤掉与数据流跟踪分析无关的会很大程度的降低实时开销。比如一些程序花费很多时间在 GUI（Graphical User Interface，图形用户接口）事务上，如果我们跟踪 GUI 相关的 API 的指令，那么目标程序往往将不能被分析出结果。所以 GUI 相关的 API 不应该被分析，应该被略过。同样的很多其他不使用污染数据的 API 也会被略过。在之后的章节中，会详细表述 API 过滤。

2.2.3 污染属性管理模块

这个模块记录所有受污染内存的内存地址和寄存器的污染状态。这个模块分为两部分，一部分是内存污染管理，一部分是寄存器污染管理。内存管理模块记录目标程序地址空间内所有内存状态是“污染的”还是“未被污染的”（Tainted or UnTainted）。实现它要注意的是为目标程序开辟的内存空间最大可能为 4G，其中 2G 为用户空间，2G 为内核空间，之前的工作，比如 LIFT[44]，保存一个与内存一对一的标签，类似于我们常见的影子内存的概念。如采用一一映射的方式记录所有内存的状态则分析工具本身就需要耗费相当大的固定空间，这对一个程序分析软件来说是不可取的。考虑到随机查询和污点维护是相对频繁的操作，因此需要采取合适的数据结构以减少这些操作的时间消耗。

在分析过程中，大部分的内存空间都是“未被污染的/干净的”，那我们就可以利用只记录污染的内存空间的方式来大大的降低空间的耗费。考虑到污染点在内存分布情况多样的特点以及查询、修改状态为主要操作,以及兼顾到时间和空间效率因素，借鉴实际内存管理中的段页式结构，选择散列表的链表实现方式为内存映射表的数据结构实现方式。在我们的数据流跟踪分析系统中，我们使用链表哈希表来记录内存标签，这些标签只标记了被污染的内存。表中的每一个节点都保存了目标程序虚拟内存空间中一个字节的污染的内存。

开始的时候，哈希表是空的，因为还没有污染的内存空间，并且所有的寄存器状态也都是“干净的”，所以标记为 0。当污染属性管理模块从污染源初始化模块获得信息，污染属性管理模块根据得到的信息向哈希表中插入节点。当程序执行，如果污染的数据被传播，那么更多的节点就可能会被插入到哈希表中。如果污染的数据被净化，那么一些表中的节点则会被删除。

寄存器管理模块记录所有(EIP 和 EFLAGS 除外)的寄存器状态(“被污染的”或者“未被污染的”)。在寄存器管理模块中，我们用 1 和 0 来表示“被污染的”还是“未被污染的”。实现它要特别注意一些寄存器之间的内在联系。比如 ax 是 eax 的低 16 位，而 ax 又由 ah 和 al 组成，因此某一个寄存器状态的变化可能影响多个寄存器的状态。一些寄存器是另外的寄存器的一部分，那当其中的一个寄存器空间的状态被改变时，另外一个也会被改变。如果一个寄存器由多个寄存器组成，则它为未受污染的条件是所有组成它的寄存器都是未受污染，而且其它组成部分也未受污染。以 eax 为例。如果 eax 被设为“被污染的”，则 ax、ah、al 都应该被设置为“被污染的”。如果这时 ax 被设置为“被污染的”，ah、al 为“未被污染的”，但是 eax 的高 16 位仍可视为“被污染的”，所以 eax 为“被污染的”。

表 2-1 污染状态描述

Table 2-1 Description of Taint

Bits	eax	ax	ah	al
000	0	0	0	0
001	1	1	0	1
010	1	1	1	0
011	1	1	1	1
100	1	0	0	0
101	1	1	0	1
110	1	1	1	0
111	1	1	1	1

本文中将寄存器分为三类，一种是 eax、ebx 这种包含 16 位寄存器，而这个 16 位寄存器又由 2 个 8 位寄存器组成。一种是包含 16 位寄存器，但这个 16 位寄存器为一个整体，如 esp、ebp。第三种是段寄存器。

对于 eax、ebx 这类，对每个寄存器采用三个 bit 来记录状态，1 表示“被污染的”，0 表示“未被污染的”。第一位为 eax 高 16 位的状态，第二位为 ah 状态，第三位为 al 状态。将这 3 bits 作一个三位二进制数字，判断 eax 的状态的时候计

算这个二进制数是否为 0，如是则为“未受污染的”，否则是“被污染的”。ax 的状态由后两位决定，ah 由第二位决定，al 由第三位决定。如表 2-1 所示。

同理，对于 esp 这类，采用 2 位表示。段寄存器只用 1 位表示。

2.2.4 攻击检测，漏洞分析

对于不同的数据流跟踪分析的应用，所谓的非法使用被污染数据都有不同的定义。那么漏洞分析（Vulnerability Analysis）也就需要不同的规则。在本章中认为恶意攻击行为的关键是改变程序正常的控制流，所以当“被污染的”数据被用作改变程序的控制流时，那就可以被认为是一个恶意攻击行为。在二进制级别，控制转移指令的结果的是程序计数器，用来标识接下来一条将要被执行指令的地址。所以，当“被污染的”数据被用作控制转移指令的操作数的时候，然后根据污染属性传播的规则，程序计数器的内容就将会被标记为“被污染的”，也就是说程序的控制流被非法的改变了，那么就可以判定这里发生了攻击行为。

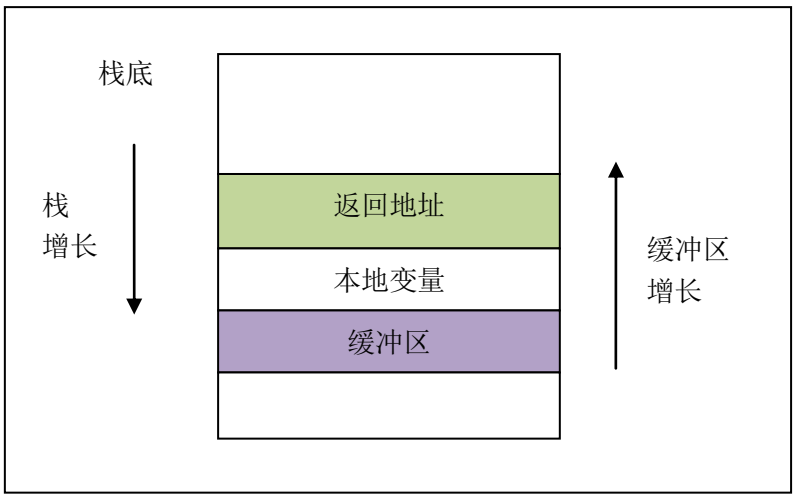

图 2- 2 函数调用栈空间布局图

Fig.2-2 Stack layout with function evocation

在那些使用缓冲区溢出漏洞来进行的攻击中，恶意攻击者经常是通过改变函数返回地址或者函数指针来进行的攻击，这里函数返回地址和函数指针都是作为控制转移指令的目标地址。还有对于分支指令，假如用于条件判断的某个操作数或者条件码被污染了，那么这也应该被该指令是不安全的。

主要的规则如下：

规则一：输入数据控制跳转

描述：恶意代码攻击程序的一种常见方法是试图获取系统的控制权。主要方法是利用输入数据改变系统的控制流，比如返回地址被改写，函数指针被覆盖，或是打开文件之后跳转到其中的恶意代码等等。另外如果存在缓冲区溢出等错误，会导致返回地址等被改写，从而改变系统的控制流。LIFT 采用这种脆弱点检测规则，这也是二进制代码脆弱点检测工具最常用的检测规则。

检测方法：针对 call，ter 以及各种跳转指令（jmp，jne，jb 等），如果源操作数为受污染的地址或使用了受污染的寄存器，则说明系统存在脆弱点。

规则二：长字符串拷贝。

描述：串传送指令(movsb，movsw，movsd)，用于将 DS：ESI 中的内容复制到 ES：EDI 中。经常与前缀 rep 连用，重复复制，这时 ECX 控制复制的次数。

我们将总结的规则定义在我们的语法库中。当有一条指令被怀疑触犯漏洞，那么指令分析模块会对这条指令进行上下文分析。如果有一条指令触犯了其中的所有规则那么会输出一个输出。

2.3 优化

这部分描述数据流跟踪分析系统的基本流程以及优化。和其他的数据流跟踪分析系统类似，基础数据流跟踪系统也会引起很高的实时开销。为了降低实时开销，我们的数据流跟踪分析系统在基础数据流跟踪系统引入了 API 过滤优化，就是非相关 API 过滤来过滤掉不必要的数据流跟踪分析。

API（应用程序编程接口）是为了让其他程序能够简单的与本程序交互而设计的程序接口。API 是由应用程序，库和操作系统来定义其他软件可以如何调用它们或者从它们获取服务。某些 API 的源代码是可以获得的，那么它们的属性以及调用之后的影响都可以预测，那我们就可以通过 API 的一些分析来大大的降低实时开销。

快速的过滤掉与数据流跟踪分析无关的会很大程度的降低实时开销。比如一些程序花费很多时间在 GUI 事务上，如果我们跟踪 GUI 相关的 API 的指令，那么目标程序往往将不能被分析出结果。所以 GUI 相关的 API 不应该被分析，应该被略过。同样的很多其他不使用污染数据的 API 也会被略过。

具体的说来就是如果某个 API 所有变量都是干净的，并且没有其他的副作用，那么就没有必要进一步对这个 API 进行内部的数据流跟踪分析。如果某个 API 的某个变量使用了污染的变量，那么就会对这个 API 进行数据流跟踪分析。比如 NtSingnakAndWaitingForSingleObject 函数就是给一个对象（Object）发信号并且等待第二个对象，这个函数不会传播污染，所以它被归类为不需要数据流跟踪分析的。但是有的函数就不能简单的被略过，比如 strcpy 就和我们的漏洞分析规则有着紧密的联系。

不相关 API 过滤主要就是基于上面的观点。我们手动建立一个小的数据库用来存放 API 相关的信息。当有一个 API 被调用时，数据流跟踪分析系统会在这个数据库中查找其相关的信息，然后系统会根据信息自动的处理这个 API，是跳过还是跟踪分析。如果完全与污染信息无关，这个 API 的跟踪分析就会被跳过。因为我们的数据库并不完备，所以有一些 API 在数据库中是找不到的，那么我们也会对这些 API 进行数据流跟踪分析。

2.4 实现平台

我们目前基于 DynamoRIO 平台已经实现了一个数据流跟踪分析系统来完成这个想法。

首先先简单的介绍一下动态二进制翻译平台。动态二进制翻译平台系统允许你通过添加、修改或者变换指令逻辑来在每一个基本块的内部插装单独的指令。通常可以通过将执行控制流从初始指令集转换到一个修改过的代码缓存来实现这一功能。在某些系统中，可以使用更高层次的、类 RISC 的伪汇编语言来实现指令级的插入。这就使得开发者能够更加简单在动态二进制翻译平台上开发工具，并且试着使该工具能够跨平台使用。动态二进制翻译系统所提供的 API 可以为我们提供范围很广的应用，从切片和最优化到机器翻译以及安全性监视。

利用动态二进制翻译就使得我们可以开发监视工具以进一步提高错误检测的能力，即从检测一个错误的发生到潜在的发现错误的来源。

使用基于调试器的监视方法，我们将无法检测到实际的溢出，因为它发生在对函数的调用过程中。对于前面所讨论的栈的溢出而言，情况也是如此。在栈的溢出的情形下，当受影响的函数返回时，可能会生成并且检测到一个异常。对于这种情形以及其它的堆的溢出而言，将不会生成一个异常，直到后续的堆的处理操作，比如对 free()的调用出现在前面的代码片段中。基于动态二进制翻译平台系统，我们就能够在运行的时候跟踪并记录所有的内存分配。术语 fence 表示标

记或者记录每一个堆块的起始地址和结束位置的能力。通过注入所有的内存写入指令，就可以进行一些相应的检查以确保任何单独的堆块的边界都没有被超越。一旦一个单一的字节超越了一个所分配的堆块的边界，那么就会发出一个报告。成功的实现这样一种技术可以为用户节省大量的时间、财力以及精力。

目前存在有许多的动态二进制翻译系统，其中的例子包括 DynamoRIO，DynInst 以及 Pin 。我们在本文中使用由马萨诸塞技术研究所和惠普公司联合开发的开源项目的 DynamoRIO 系统，主要有以下几个原因。

(1) DynamoRIO 是在 IA-32 框架中实现的，它支持微软的 Windows 操作系统以及 Linux 操作系统。

(2) DynamoRIO 非常的稳定，并且具有很高的性能，这已经通过其在商业软件工具如 Determina 的内存防火墙中的应用得到了证实。

(3) DynamoRIO 是一个动态二进制分析平台，支持二进制分析技术，能在不影响程序执行结果的前提下，通过在程序动态执行过程中插入额外的分析代码，动态监控程序的执行过程。DynamoRIO 工作在操作系统和应用程序之间，它采用了代码缓存技术，通过将程序的代码拷贝进代码缓存的方法来对目标程序进行模拟执行。在模拟执行过程中，用户可以对目标的二进制代码进行修改，实行指令级分析。

(4) DynamoRIO 具有良好的扩展机制，用户可以利用 DynamoRIO 提供的接口编写各种二进制分析工具。

所以本文中选择使用此工具作为二进制分析平台。

2.5 性能

表 2-2 实验平台

Table 2-2 Experiment paltform

CPU	Intel Core2 3.00GHZ
内存	DDR2 3G
硬盘	Seagate SATA 1TB
操作系统	Windows XP SP3

我们在表 2-2 的实验平台中实验我们的系统，并得到图 2-3 的结果。

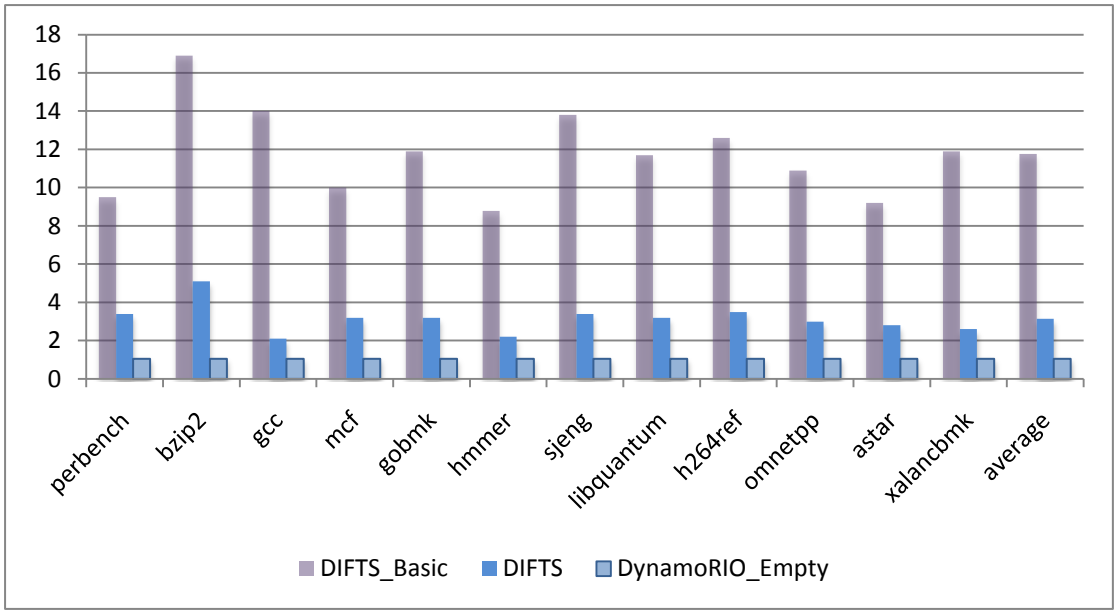

图 2-3 数据流跟踪分析性能实验结果

Fig.2-3 The performance of Dynamic Information Flow Tracking System

上图给出了使用 SPEC CINT2006 测试数据流跟踪分析系统的性能，结果表明本系统有较小的实时开销，主要是因为使用了优化手段。数据流跟踪分析的开销大概是 1.9 到 6.9 倍，平均 3.3 倍。和其他的一些工具相比，我们的系统有比较小的实时开销（LIFT 平均 3.5 倍，Dytan50 倍，Panorama20 倍，SeeC15.2 倍）。

图中给出了优化在基础数据流跟踪分析系统（图中 DIFTS_Basic 表示基础数据流跟踪分析系统的性能，也就是尚未优化的系统）基础上所减少的开销。没有任何优化的情况下，系统就不同的目标是程序会减慢执行时间 6.8 倍到 23.5 倍，平均 12.2 倍。通过优化之后目标程序的平均实时开销减少许多。这个的最主要原因是我们使用了一些优化手段，最有效的优化手段是非相关 API 过滤，单独使用这一优化就可以平均减少实时开销 5.8 倍。主要是因为我们的系统在处理 API 时使用更有效地处理模式，我们检查 API 的参数是否是“被污染的”，跳过检查 API 主体的指令，而不是一条一条的检查每一条指令。

无论是在实时开销还是占用空间方面，我们的系统都是比较有效的。我们不使用内存映射，而是使用哈希表来跟踪污染信息，这在很大程度上减少了所占用的内存空间。Dytan 工具有空间耗用比较大的缺陷，其空间耗用近似达到 240 倍。LIFT 以空间开销来换取实时开销，也就是说他的空间开销问题也没有解决。而我们的系统将污染信息保存在哈希表中，从而成功的将空间开销降低，目前系统的空间开销大约在 10 倍。

2.6 小结

本章主要是写上海交通大学软件学院嵌入式实验室程序分析小组共同完成的数据流跟踪分析系统的设计思路以及一些比较重要的问题的解决思路。此系统是在动态二进制翻译插装平台 DynamoRIO 上面完成的动态二进制数据流跟踪分析系统，主要用来当攻击发生时，探测攻击，发现漏洞。

本章开始就数据流跟踪分析技术的优缺点做了分析。然后主要介绍了我们的数据流跟踪分析系统的设计框架，并对框架中的各个模块做了详细的阐述。数据流跟踪分析系统的框架主要是从数据流跟踪分析的三个步骤出发来设计的。三个步骤主要是“污染源”的确定，污染属性的传播，规则的制定以及使用。“污染源初始化模块”初始化其他模块并且启动整个系统，对应“污染源”的确定的步骤。“指令分析模块”分析数据污染属性的传播或者消亡并且与“污染属性管理模块”交互来设置以及查询某一内存或者寄存器的污染状态对应污染属性的传播的步骤。系统使用“语法数据库”里存放的规则来发现漏洞，对应规则的制定以及使用的步骤。并在之后在标准测试程序上进行了性能测试，我们的实验结果用数据证明了我们系统性能上面的优越性。

第三章 库函数识别

第三章介绍了反编译技术的基本知识以及库函数识别在反编译中的重要作用，然后描述了库函数识别的重要算法。

3.1 反编译与库函数识别

反向工程是指通过对别人软件可运行程序系统进行逆向分析、研究的工作，从而推导出他人的软件所使用的思路原理、结构算法、以及处理过程、运行方法等要素从而加以利用参考的过程。反编译是编译的反过程，属于计算机软件反向工程（Reverse Engineering）的一部分，它能够将低抽象层次的可执行程序转换为高抽象层次的源代码。

在维基百科中，反编译(Decompilation)被定义为：

“Translates a file containing information at a relatively low level of abstraction (usually designed to be computer readable rather than human readable) into a form having a higher level of abstraction (usually designed to be human readable).”

也就是从较低抽象层次文件出发从而得到与之对应的较高抽象层次语言文件的过程。此定义可能稍显宽松，因为大家都知道在大多数情况下反编译一般都是以机器语言可执行文件为起点，并且以特定的高级语言源代码为终点。但是我们同样应当注意到的是，在某些反编译的过程当中，反编译的起点也有可能是中间代码，比如对 Java 语言的反编译，并且反编译的终点也非常有可能是到达流程图等比源代码更高的抽象层级。

自从 1960 年，美国海军电子实验室搭建第一个反编译器以来，反编译技术已经在二进制程序修改、源代码重建、程序移植等方面得到了广泛应用，并在软件维护以及技术发展方面有着重要意义。但是与编译技术发展的一帆风顺不一样的是，反编译技术的发展前景始终面临着困境。其中最主要的就是反编译技术在法律上面的争议。20 世纪 90 年代，反编译技术中的法律的障碍被扫除之后，反编译迎来了第二次的大发展。在这一时期中出现了为数不少的试验性的 C 反编译器[46]，但对于 C++人类到目前为止应用最为广泛的语言的反编译技术成果却是乏善可称的。这是由于在现代编译器的编译过程中丢掉了很多信息，这自然就给反编译技术带来了很多很难去克服的困难。

现代的反编译技术过程可以细分为以下几个阶段：句法分析，语义分析，中间代码生成，控制流向图生成，数据流分析，控制流分析，代码生成[47]。在具体实现的时候，为了将机器和语言之间依赖的特征分离，他们常被封装为三个模块：前端，UDM(通用反编译机器)，后端。前端是由那些机器依赖的阶段组成的，它产生一个与机器无关的、中间的程序表示法。UDM 则是一个完全独立于机器和语言的中间模块，也是进行反编译分析的核心。后端是由那些目标语言依赖的或高级语言依赖的阶段组成。

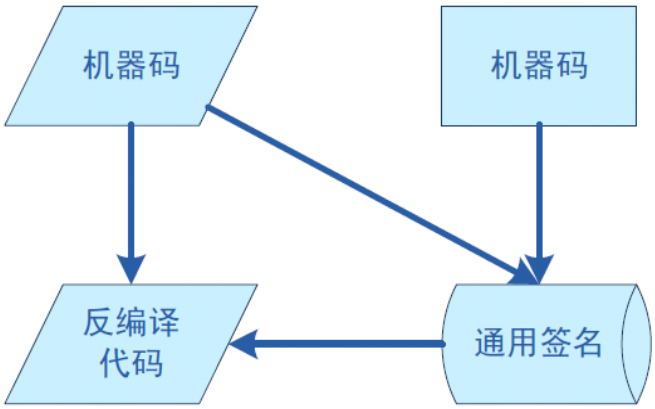

图 3-1 库函数识别简图

Fig.3-1 Function Recognition

库函数识别过程位于反编译技术的前端，是反编译技术的重要组成部分。识别后的库函数能够提供很多非常有用的信息，比如返回值，参数类型，控制流等等，他们对反编译中的控制流的恢复与数据流的恢复有重要作用，它可以很好的帮助提高数据流跟踪分析准确度。

一般来说，库函数的识别分为三个阶段，前期处理、签名生成和库函数识别。前期处理阶段的最主要任务就是自动地或人工地得到生成签名所需要的目标代码，这一个阶段牵涉到的技术主要包括：编译环境的穷举，编译器版本的识别，自动代码生成等。签名生成的过程利用上一个阶段获取的目标代码，设计并且提取出能够用于区分目标函数的签名，这一过程是整个库函数识别的重点。库函数识别过程相对而言比较简单，它是利用得到的通用签名匹配识别目标代码的过程。一般来说，函数的识别过程和反编译器的具体实现联系非常紧密。

下面来介绍一下反编译基本模型与研究方法[47]。

反编译是编译过程的逆过程。由于编译过程当中不可避免的会丢掉很多的信息，所以反编译不可能完全的实现逆映射编译过程。在反编译的各个环节当中，

很多都是使用特征提取以及模式识别的手段，从较低抽象层级的文件当中提取特征推断出较高抽象层级的代码，实质上这是一个信息重建的过程。

常见的反编译器是一个分层结构:

$$Dec = D \cdot C \cdot L$$

公式当中的 D 代表的是数据流的恢复，C 代表的是控制流的恢复，而 L 代表的是库函数的识别和恢复，这就是反编译器的三层结构模型。加上预处理(Pre-process)以及后处理(Post-process)过程就构成了反编译器的五层结构模型。

3.2 库函数识别在反编译流程中位置

库函数的恢复是库函数连接过程的逆过程。具体来说，库函数恢复开始的时间往往是数据流恢复以及控制流恢复之前，并且在中间代码生成结束之后的。在反编译技术中库函数的识别过程往往是一个需要攻克的难点，其识别的成功率将直接影响到之后的数据流的恢复以及控制流恢复的效果，同时还将直接表现在反编译最终代码的可读性上面。

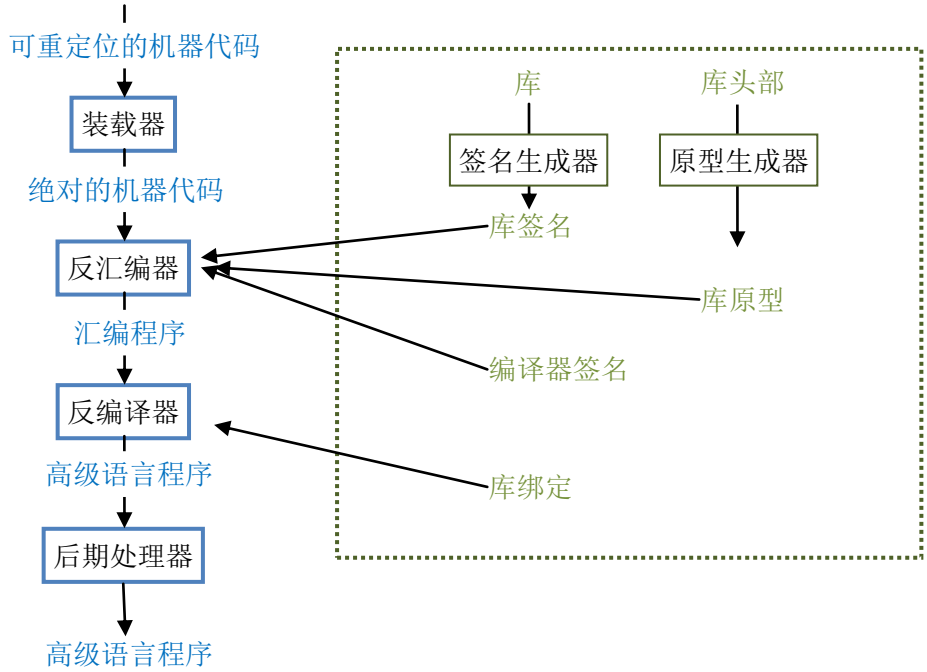

图 3-2 库函数识别在反编译中位置

Fig.3-2 Location of function recognition

我们首先通过DCC反编译器的上下文(图3-2)对库函数识别过程在整个反编译的流程中的位置做大体的了解。

图3-2中的DCC反编译上下文只是用作参考，实际在工作当中库签名的生成等许多技术和环节都会发生不同程度的改变。但我们还是会很明显的发现库函数识别的准备工作已经早在对具体代码的反编译开始之前就已经开始了，反编译器则需要预先针对函数库进行一些处理，将可识别的特征抽取出来储存在“签名库”当中，等到在中间语言(一般情况是指汇编代码)的生成阶段，反编译器将利用预先生成的库签名通过比对算法进行库绑定，完成库的识别工作。我们提取这一部分的工作帮助进行数据流跟踪分析。

由此可见，总体而言，库签名识别的准备工作基本覆盖了整个反编译的前端，而识别的过程位于前端与UDM的连接阶段。不过在个别反编译器的实现当中，库签名的识别过程也可能会访问中间代码之前的数据，但是这种情况是比较少见的，而且影响也不大。特别是对我们的数据流跟踪分析几乎是没有影响的。

3.3 库函数识别的重要性

如前面的文章阐述，库函数识别是库函数连接的逆过程，它对整个库函数的识别的影响也很大，甚至都可以说库函数的识别的效果能够直接影响到整个反编译器的成败。

在本小节当中，将会通过以下三个方面对其进行讨论：

(1) 指导数据流的恢复和控制流的恢复

在前面的章节当中我们曾经分析过，反编译的过程在本质上是一个从不完全的信息推断出完全信息的过程。由于在编译过程，特别是C／C＋＋的编译过程当中丢掉了许多的信息，因此这一逆过程也就非常的困难。然而库函数识别的技术就为补全遗失的信息提供了一条非常便捷的途径。

由于库函数一般都是公开的，一旦反编译器通过库函数识别技术手段识别出了一段程序块，那么很方便就可以通过该库函数的内部控制流、入口参数等信息协助确定控制流、变量的类型等信息从而协助数据流以及控制流的恢复过程。我们正式利用了这一点原理来将库函数识别技术应用到数据流跟踪分析系统中，希望能够加强数据流跟踪的准确度。

非常值得一提的是，由于C＋＋与C相比做了非常多的变化，导致一些在C

反编译时代广泛采用的技术手段受到了限制，比如说 C 反编译中常常利用库函数参数类型来确定变量类型，然而在 C＋＋当中库函数多用模板完成，那就无从确认参数的类型。然而这并不影响库函数的反编译过程，它仍然通过各个途径指导、辅助着 UDM 过程(数据流恢复和控制流恢复)。

(2) 减轻数据流恢复和控制流恢复的工作量

统计显示，在一个真实程序中，平均有多达半数以上的函数为库函数。在各种窗口应用程序以及侧重业务服务的程序当中这一比例就更高。一个非常典型的例子是，编译一个“Hello World！”程序得到的可执行文件中甚至都包括了 58 个库函数！

这样一来，假如没有库函数识别的帮助，我们将不得不花费至少 50%的时间和运算资源来对库函数代码进行毫无意义的分析与生成。而相对的，一个成功的库函数识别的模块将极大地减少降低数据恢复和控制流恢复的压力。这样一来，我们的数据流跟踪分析也就能够通过库函数识别的思路来优化。

(3) 提高高级语言代码的可读性

到目前为止，人们已经在反编译领域做出了半个世纪的努力，也先后实现了很多可以运行的反编译器。然而随着高级语言便得越来越灵活，程序结构变得越来越复杂，生成代码的可读性差的问题就显得越来越严重。

随着各种功能、结构高度复杂的库函数的引入，这一问题愈加深重，也越来越严重的影响到反编译的根本目的：也帮助人们理解与机器代码同质的高抽象层的设计思路与程序意图。这个虽然对于我们的数据流跟踪分析技术没有什么意义，但是作为，库函数的一个重要意义，我认为有必要在此阐述清楚。

我们都知道一个仅仅包括寥寥数行语句的程序在调用了库函数并经过编译器编译之后很有可能就成为一个庞然大物,更何况那些本身就异常庞大而且频繁调用库函数的程序。一个脱离了库函数识别的反编译程序生成的高级语言代码，必将是极其难以理解的。对于程序员来说，得到这种无法理解的高级语言程序也对于他们理解源程序的设和意图完全没有帮助，自然而然的这将会影响到他们对该反编译器的使用欲望。很多时候，生成代码的可读性是否良好将决定一个反编译器的生死。

由此可见，库函数识别的成功与否对于整个反编译过程来说都是非常重要的。当然对于我们数据流跟踪分析技术的改进也是非常新颖以及有帮助的。

3.4 库函数识别的难点[48]

3.4.1 所占内存空间

在函数识别中最大的困难就是所需识别的函数数量以及所占据内存空间很大。库函数数量巨大，而每个库签名往往又要包括对应库函数的全部操作码序列，再加上函数名、函数长度等等相关信息，整个签名库所占用的空间会相当庞大。而当反编译器能对应多个编译平台时，该签名库的大小更是倍增。如果我们考虑各种编译器针对不同内存所产生的所有版本的库，那么这种数量级将无法轻易地达到，很容易超出当前计算机能力所能达到的范围。但是对于一般的个人用户来说，很难说服他们仅仅为了一个小小的反编译器或者攻击探测系统，就耗费如此之多的存储空间以存储这些签名库。所以如何存储这些签名就成了一个需要解决的问题。MFC（Microsoft Foundation Classes），是一个微软公司提供的类库（Class Libraries），以 C＋＋类的形式封装了 Windows 的 API，并且包含一个应用程序框架，以减少应用程序开发人员的工作量。其中包含的类包含大量 Windows 句柄封装类和很多 Windows 的内建控件和组件的封装类。如果考虑 MFC 库和其他通用的库，那么所需要的内存空间要求将是非常巨大的。这些要求对于现在的个人电脑来说是不能负担起来的。

在 IDA 的实现中，Ilfak GUIlfanov 提出了一种解决思路，就是使用 CRC(冗余循环校验码)来压缩操作码序列。由于操作码序列可想而知有很多字节是雷同的，所以这种手段得到的效果相当好，而且压缩过的签名不仅利于存储，还利于查找过程的快速识别。

当然压缩操作码序列的手段还有很多，比如说使用散列函数或者使用树状结构存储也是不错的解决手段。

在本文中使用了一个优化策略来减少识别函数所需要的信息以减少其所占用的内存空间。该策略是并不是将所有的函数都识别出来，而是只有与程序行为相关的函数体会被识别出来，并对识别出的函数进行分析。这样就大大的降低了所需要识别分析的函数，降低了所需占用的内存，又提高了效率。

3.4.2 签名冲突

签名冲突是指，在确定了签名的设计以及相应的签名生成方式之后，该方法对至少两个库函数生成的签名相同。

签名冲突产生的原因一般有两种：

(1) 所采用签名方法有缺陷

我们曾经说过，评判一个签名方法有两个标准，一个是精确性即其区别性的识别出目标代码的能力；另一个是冗余度，也即签名本身所携带信息的多少。通常而言精确性与冗余度是此消彼长的关系，研究人员很难使二者都达到最好。因此有些时候，即使明知签名方法有缺陷，无法精确区分少数库函数，但为了不增加其冗余度，很多研究人员还是沿用有缺陷的签名方法。同时采用特别处理的手段对待少数签名未能区分的库函数。

(2) 库函数本身过于近似

在函数库中存在着这么一些库函数：他们的操作码序列完全相同。比如 getx()和 gety()、fread()和 fwrite(). 基于操作码序列设计的签名方法对于这些函数完全束手无策。不过庆幸的是这些函数只是极少数，我们完全可以采用特殊处理的手段将他们识别出来：比如我们注意到可执行程序中 getx()后总是紧跟着 gety()，而 gety()之后为其他库函数。因此，当反编译器识别出函数为 getx()或 gety()时，可以通过对其上下文的函数的识别结果来确定该库函数。

此外，我们还可以采用分级扩展的方法识别此类库函数。即当我们发现目标代码对应 2 个以上签名时，可以顺次打开其中 call 跳转指向的相关代码，然后通过对下一级的代码的识别结果来判断该函数。

还有一种情况也很重要，重定位在 32 位得代码当中，涉及到直接寻址指令都需要重定位。明显可以看出来，由于重定位信息的存在，所以不能利用函数模板来做逐字节的对比。更为糟糕的是有些库函数在 LIB 文件里面逐字节对比全部都是一样的，两个只是重定位符号不一样。比如当用函数提取的方式来识别函数的时候，一般的方法是将调用指令的重定位信息用“00 取代，那这样就有一个问题，如果两个函数的大部分都是相同的除了调用指令那一条指令，那么这两个函数会被生成相同的签名，这样也会形成所谓的签名冲突。这种情况下，为了解决函数冲突问题，原始的通用签名会和被调用的函数的机器码用特殊的标志，比如“&”，联系到一起，那么被调用的函数地址就可以在相关联的.obj 文件中

找到。这样一个唯一的新的签名就生成了。

3.5 库函数识别的主要步骤

在本小节当中，我们将对库函数识别中的主要步骤与问题作些介绍。

首先库函数的识别过程可以简单的分为两个大的步骤，库函数特征的提取和库函数的识别[49]。

3.5.1 环节一:库函数特征的提取

库函数特征的提取是库函数识别的先决条件。特征提取的精确性将影响库函数识别的效果，而特征的冗余度也决定了识别过程的效率和耗费资源的大小。这里用精确性衡量特征正确表达目的函数并区分任意其他函数的能力，而用冗余度来衡量特征本身携带信息量的大小。显然，很难做到精确性和冗余度共同的最优化，大多数情况下我们必须试图在二者之间寻找一个平衡点，迄今为止，国内外已经提出了相当多的技术和算法来解决这一问题。

须注意，这里的库函数特征通常是指由库函数特征提取方法所决定的一系列用来限定库函数的特征信息的集合，这些特征被提取出来之后按照特定的手段被存储起来。在后文中，为方便读者阅读，我们将把这些被提取出来的信息集合称作“签名”，而用库函数特征代表包含所有目标库函数信息的信息集合。

在确定了签名提取的方式之后，其精确性与冗余度已经基本确定。然而还有相关工作需要处理。

(1) 目标函数代码的穷举

我们曾提到，对于同一高级语言，各个编译器在实现时会各有不同。同样，对于同一标准的标准函数库，各个平台在支持时也会有微小的不同，甚至作一定程度的自行扩展。

此外，随着编译器优化技术的进步，在不同环境下，或在不同编译选项下，相同库函数所生成的机器码也会有所不同。

特别的，对于 C＋＋标准函数库中的很多函数，由于采用模板写成，他们根据特定参数类型实例化之后的结果再经过优化编译所生成的最终机器码更是不

一而足。

这样一来，为了使最终所得到的函数签名尽可能的覆盖所有变化，我们在生成签名之前势必要对大多数可能性进行一定程度的穷举。

通常情况下，在穷举过程中至少要包括对多个编译平台的穷举、对不同编译选项的穷举等。当然，具体的穷举过程依赖于特定高级语言的特性以及所采用的签名生成方法，在这里难以赘述。

(2) 生成通用签名

经过上面所述的穷举环节之后，我们已经得到了目标函数在各种环境下的目标代码。然而，通过这些代码或其分别生成得到的签名识别目标代码还是一项极其繁重的工作。为此，我们有必要通过一定的手段，将这些代码/签名汇总成为一个或少数几个通用签名以加速识别过程。有许多相关算法被研究出来以解决这个问题。

3.5.2 环节二:库函数的识别

经过环节一之后，我们已经得到了一个或数个通用签名。在这些通用签名之内包含了可以用来确认库函数的一些特征。在识别环节的主要任务就是根据先决条件，比如所采用的编译器机器版本等信息，来适当的利用这些签名以完成库函数的识别。

当然，在这一环节也有很多相关的算法存在，但由于这些算法大多数都依赖于所采用签名生成算法，故而很难在本章节中对此作详细介绍。不过在后文中，当讨论到具体的库函数识别方法时自然会对此作简单介绍。

3.5.3 库函数识别研究的主要内容

通过对上述流程的描述我们可以轻易看出，库函数识别的主要内容都集中于所采用的签名生成算法以及与之相配套的签名识别方法。根据所采用签名生成方法的不同甚至会影响库函数识别的流程。

所以库函数签名的设计，其生成方法、识别方法以及相应的储存方法当之无愧的成为库函数识别研究的关键点和重心。

3.6 库函数识别算法[47]

在 C＋＋库中，依赖于参数类型与编译器的优化，函数代码的长度以及操作码都有可能发生改变，也即操作码序列的长度以及序列中的个别值有可能发生改变。所以 C＋＋库识别中不能直接采用现在比较多的 C 库识别中使用的通用签名识别方法。但是同一个库在各种条件下生成的操作码序列应当具有相似性。而且我们认为这种相似性应当大于不同库函数的操作码序裂中的相似性。那么基本的思路就是通过适当的忽略操作码序裂中的部分操作码，来得到一个更通用的操作码序列，作为函数识别的核心。

在本文中抽取机器码的共同的部分将被作为通用签名。签名的抽取是一项有难度的工作，我们通过以下的算法来完成签名的抽取。

3.6.1 签名的设计

本文中定义库函数特征签名为如下形式：

Func_pattern = (Length, Maskcode, Func_info)

其中 Length 代表的是指令条数， Maskcode 掩码用于指定操作码序列中的可变/不变部分，Func_info 指的是库函数其它信息如函数名、参数个数等。Func_info 不仅有用于协助识别，其储存的信息也对成功识别后的后续反编译步骤有重要作用。

3.6.2 签名生成算法

签名生成前，首先需要采集一些“样本”，通常是目标库函数在不同参数类型下生成的目标代码。

算法主要思想如下：

假设条件：设 Sig＝｛S_0，S_1，S_2，S_3,...｝为某个库函数 Func 的样本集，$Code_i$为第 i 次训练后的签名（也就是指令序列)。

1) $Code_0 = S_0$
2) 从 Sig 中取出样本S_k并将其从 Sig 中删除
3) $Code_{i+1}$=**Mix(**$Code_i$，S_k**)**
4) 若 Sig 为空或达到中止条件，goto **6**）

   //本步骤的中止条件可以设为(i > max AND $code_i$ == $code_i$-max). max 可自行调整，通常设为 10~20.
5) goto **2**）
6) Func.pattern=($Code_n$,Length,$Code_n$,Func.other_info)

   //最终的 $Code_n$就是函数的不变加上可变指令序列。本步骤可以根据这个指令序列和其它信息生成用于识别的库函数特征 Func.pattern。

其中 Mix 函数的描述如下：

Mix(Code, Sig)

设一个空的比较结果，单位是指令，长度为 Code 的指令条数

设置标 p 指向 Code 的第一条指令

如果当前指令为 X，则设比较结果中对应的指令为 X, goto **5)**

//X 为通配符

比较 p 所指向的指令与 Sig 中对应的指令。若相同则设比较结果中对应指令为此指令，否则设对应指令为 X

p 移动到下一条指令

如果超过 Code 的末尾则 goto **7)**,否则 goto **3)**

返回已填充好的比较结果

在此过程中，使用如“HHHH”以及“& &”的特殊符号将签名分成几个子序列。鉴于我们需要考虑不同参数类型在原始签名中会有不同的长度，像“00”这样的特殊符号就会被插入到较短的签名中，其中它们不同的部分也会被“00”取代来提取原始签名的共同的部分。

生成的过程如下：首先，一个包含所有相关函数.cpp 文件会被各种编译器使用这种编译选项编译，那么一些.obj 文件会生成。然后每个.obj 文件会被分析，并且这些函数的机器码会被用来生成签名。

算法用伪代码完整表述如下：

输入：某函数样本集合：Sig＝｛S_0，S_1，S_2，S_3，...｝，

输出：通用签名 Func.pattern

Procedure GeneralExtract

$Code_0 = S_0$；

While (Sig≠NULL & *未到终止条件*)

//本步骤的中止条件可以设为(i > max AND $code_i$ == $code_i$-max). max 可自行调整，通常设为 10~20.

{Random select S_k from Sig;

$Code_{i+1}$=**Mix($Code_i$，S_k)**；

Delete S_k from Sig；}

Func.pattern=($Code_n$,Length,$Code_n$,Func.other_info)

//最终的 $Code_n$就是函数的不变加上可变指令序列。本步骤可以根据这个指令序列和其它信息生成用于识别的库函数特征 Func.pattern。

Procedure Mix($Code_i$，S_k)

设空的比较结果 M;

//长度为 Code 的指令条数，单位是指令。

p->$Code_0$;

while（Code≠NULL）

{if(p->X)

{设比较结果 M 中对应的指令为 X; //X 为通配符

p->p.next; }

elseif(p->Sig)

{if (相同) 设 M 中对应指令为此指令;

if（不同）设 M 中对应的指令为 X;

p->p.next;}}

return 比较结果;

3.6.3 签名识别算法

基本流程是，根据采集得到的签名库Ｐ＝｛Ｐ$_1$，Ｐ$_2$,...｝,对目标中间代码依次进行模式匹配以识别出库函数。

Procedure Recognize

从反编译其前一阶段获得带识别函数集合Ｆ；

识别出目标代码中的函数入口和出口；

分解出单个函数列表 Ｆ＝｛Ｆ$_1$，Ｆ$_2$,...，Ｆ$_n$｝;

//其中Ｆ$_i$的特征包括：Length 指令条数，asMcode 函数汇编代码 (中间代码)

If（Ｐ$_i$.Length = Ｆ$_i$.Length & Match(Ｐ$_i$.Maskcode = Ｆ$_i$.asMcode)）

Ｆ$_i$．Func_info= Ｐ$_i$.Func_info；

Procedure Match（Maskcode, asMcode）

```
p->Maskcode 第一条；
while(Code≠NULL)
    {if(Ｐ$_i$所指向指令与 asMcode 中对应指令不同)
        {return False；
        P->p.next;}
    if(当前指令为 X) p->p.next; //X 为通配符}
return True;
```

通过签名进行函数识别时，当一个函数调用发生了，那么调用指令会被和签

名比对，如果结果匹配则被认为是被识别的函数，主要通过比较调用指令的机器码和签名来识别函数调用。要确定一个函数调用必须比对所有的签名。

3.6.4 算法的不足

利用这种算法进行库函数识别，我们已经可以成功的识别出一些 C++中的库函数。但是该方法的有一个先决要求：需要事先采集样本进行训练，而且其生成的签名对于训练的样本非常敏感。因此，对于那些参数类型不仅仅是常用数据类型的库函数(比如容器类)，由于无法事先采集相应的样本进行训练，其识别的效果比较差强人意。

比如对于模板库函数 foo<T>(), 如果提取的训练样本为 foo<int>,foo<char>,…，那么当面对 foo<UserType>时，还是难免会无法正确识别。

3.6.5 模板识别法

a) 基本思想[48]

上面的算法失败的主要原因就在于它还是试图识别模板库函数的实体，于是当面对未知类型参数对应的未经训练过的实体时就超出了它的能力范围。

而模板识别法则将专注的对象从实例化之后的实体代码上转移到了对模板函数本身的识别上。当然，由于模板函数本身不能被编译，所以模板识别法关注的是模板函数代码在目标代码中的映射：指令块。

模板识别法认为，既然模板库函数的代码不会发生改变，那么其实体代码编译后得到的目标码中对应代码块之间的顺序关系理应遵循模板库代码的顺序关系，模板识别法的基本思想就是通过利用这种内在的相似性来进行模板库的识别。

b) 签名的设计

模板识别法的签名被定义为一种变长模糊指令序列：

变长模糊指令序列为一有序集合 Seq={S_1，S_2，…S_n},其中 S_i 有以下几种可能：

1) 单条汇编指令，包括操作码和操作数；

2) 单条掩码指令，包括可变的操作码或可变的操作数。

故而最终得到的签名是一个由通配符与机器码构成的，类似于正则表达式的签名。相应的签名识别过程也可以采用模式匹配技术直接实现。目前我们完成了这个功能，但是此方法还没有在我们的系统中实现。

3.7 FRDIFTS 系统设计

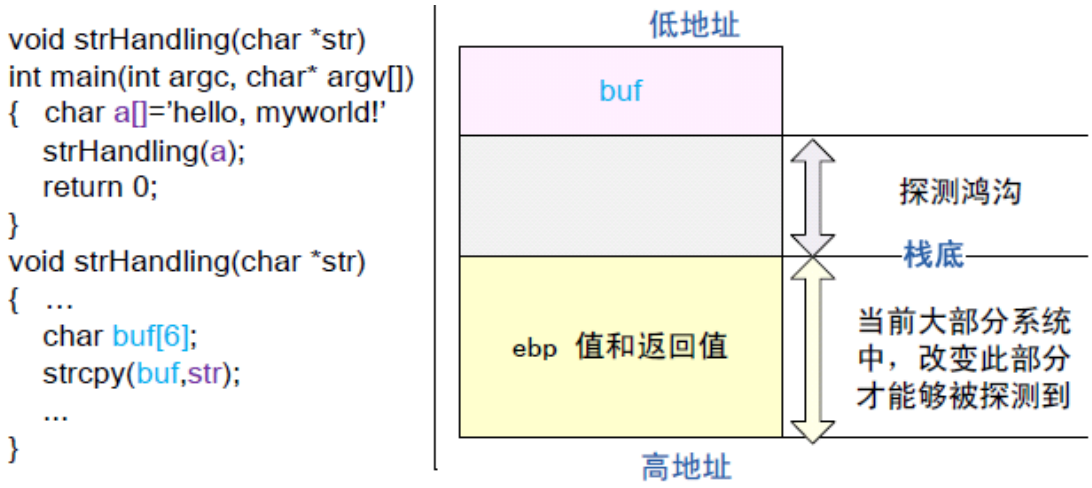

图 3-3 缓冲区溢出的例子。

Fig.3-3 An example of buffer overflow.

以此图继续说明库函数识别对于数据流跟踪分析的改进方法，当某个函数被调用时，函数会被识别出来，那么函数的行为也就被识别。以图为例，在 *strHandling* 函数中，缓冲区的大小比复制到此缓冲区的字符串要小，这时就产生了所谓的缓冲区溢出。在传统的方法中，只有 ebp 或者返回值被改变时才会被探测到。传统工具与本文工具之间有一个探测鸿沟。在 FRDIFTS 中，只要简单的比较 strcpy 函数两个参数的大小，这一类的缓冲区溢出漏洞就可以很容易的被探测到。C＋＋中有很多库函数调用，对库函数的识别能够明显的改善数据流跟踪分析的准确度。

数据流跟踪分析技术通常跟踪数据流来分析控制流是否被输入改变。一般来说，数据流跟踪分析将标记从不安全途径输入的信息为“污染的”，然后跟踪数据流，如果有信息是从数据流来的，那么这些数据也被标志为“污染的”。通过

这种方式可以分析一个程序的行为并将其呈现出来。

但是通常的数据流跟踪分析技术大部分关注的是控制流的改变。但是数据流的改变经常会被忽视。在本文的系统中，在数据流跟踪分析技术中引入了函数识别技术来解决上面提到的问题。下图是改进的数据流跟踪分析系统结构，系统的大多数部分都和上面一章提到数据流跟踪分析系统是相同的。不同的在本章的系统中我们引入了函数识别技术，以完成 FRDIFTS（Function Recognition Dynamic Information Flow Tracking System）。

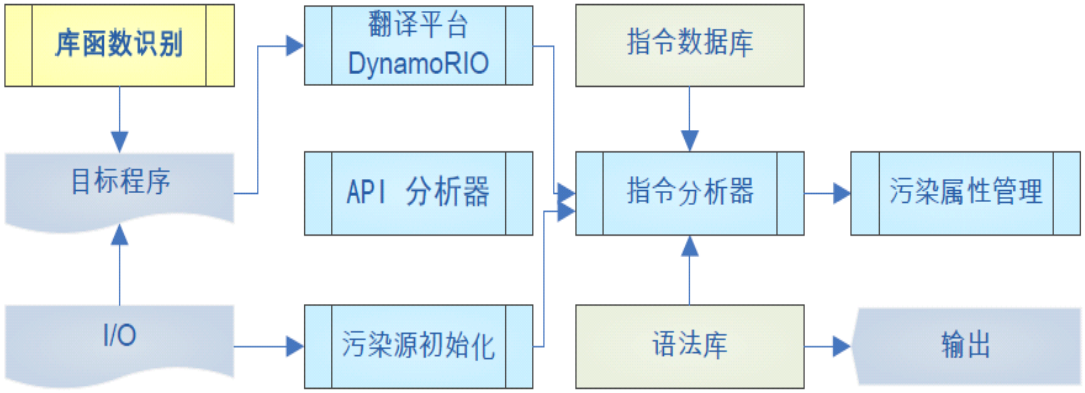

图 3-4 FRDIFTS 系统结构。

Fig.3-4 Architecture of FRDIFTD

函数识别模块是用来识别函数体的模块。和上面系统结构一样，污染源初始化模块用来初始化其他模块并启动系统。指令分析模块是用来分析污染的传播以及消亡，并且与污染属性管理模块交互来查询或者设置某块内存或者寄存器的污染属性。语法数据库用来存放漏洞探测规则。

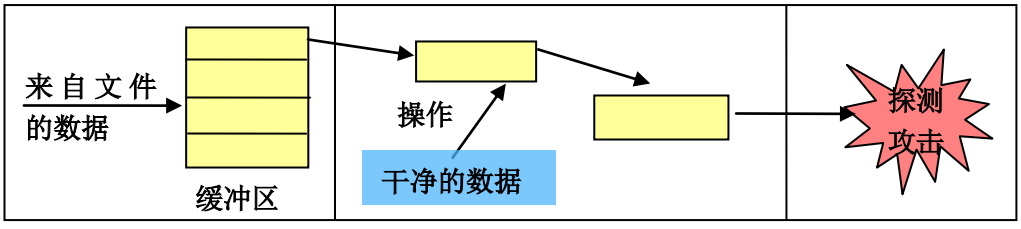

图 3-5 FRDIFTS 分析防御攻击

Fig.3-5 Detection of an attack with dynamic information flow tracking

3.8 小结

目前动态二进制数据流跟踪分析技术已经相对应用比较广泛，但是它也有一些问题需要解决，比如在缓冲区溢出的探测方面，常常只能探测比较严重的溢出，对于一些轻微的溢出，虽然没有改变程序的控制流，但是对于程序的数据还是有影响，所以还是应该警惕，而且这种情况也说明此处能够发生溢出，这里有漏洞。我们应该注意此类情况，所以我们加入了库函数识别来增加动态二进制数据流跟踪分析的准确度。

本章主要写如何库函数识别，引入库函数识别来到动态二进制数据流跟踪分析技术来解决提到的问题。在前一章的系统框架的基础上结合库函数识别技术，改进了数据流跟踪分析的准确性，不但能够在没有源代码的情况下探测攻击，而且比其他系统改进了准确性。

第四章 实验与结果

在这一部分，将给出数据流跟踪分析系统的实验结果，共包括两个部分：首先我们测试了我们的数据流跟踪分析系统的有效性；然后我们测试了我们数据流跟踪系统的性能。

4.1 实验平台

下表中为本文实验的实验平台的配置。

表 4-1 实验平台

Table 4-1 Experiment Platform

CPU	Intel Core2 3.00GHZ
内存	DDR2 3G
硬盘	Seagate SATA 1TB
操作系统	Windows XP SP3

4.2 函数识别结果

表 4-2 函数识别的结果

Table 4-2 Result of Function Recognition

应用程序	源代码 UDF 数	识别出的 UDF 数	源代码 API 数	识别出的 API 数	fp% / fn%
Win32.exe	4	4	17	17	0% / 0%
Fibo.exe	0	0	0	0	0% / 0%
BenchFunc.exe	4	4	11	11	0% / 0%
Valstring.exe	0	0	0	0	0% / 0%
StrAPI.exe	4	4	11	11	0% / 0%
Hallint.exe	4	4	11	11	0% / 0%
Notepad_prime.exe	52	52	62	62	0% / 0%

我们使用了下表 4-2 中的 7 个应用程序来测试我们的库函数识别，并在表中给出了库函数识别的结果。所有函数中出现的代码可以被分为两类，一种是用户

定义的函数（UDF，User-Defined Function），另外一种是系统的 API。在我的实验结果当中，误报（False Positive）比率和漏报（False Negative）比率均为 0%。在表中的 Notepad_prime.exe 是由 Visual C++ 2008 编译的第三方程序。他和微软提供的 Notepad.exe 相同的功能和类似的界面。表中的 fp%表示误报率，fn%表示漏报率。

4.3 FRDIFTS 示例程序

```
01. movzx ecx, word ptr [eax+8]
02. movzx edx, word ptr [eax+6]
03. shl   ecx, 10h
04. or    ecx, edx
05. mov   edx, [esp+2Ch]
06. mov   eax, [ecx]
07. push  edx
08. mov   edx, [esp+18h]
09. push  esi
10. push  ebp
11. push  edx
12. mov   edx, [esp+2Ch]
13. push  edx
14. call  dword ptr [eax+60h]
```

我们来看上面的例子，我们的系统发现了这个漏洞。具体我们分析我们的系统是如何发现这个漏洞的。在实验中发现在(6)中的 mov 指令中使用的通用寄存器 eax 指向含有“被污染”信息的内存。所以也就是污染被传播到了 eax 当中，那么 eax 现在就是“被污染的”。因为最后(14)当中 eax 被作为跳转的地址，那么

在这里就会出现一个提醒这里有一个漏洞。尽管这里 eax 的值并不是直接从 I/O 来的，但是这种情况也需要考虑。

4.4 功能分析

FRDIFTS 能够检测出栈上面利用缓冲区溢出重写函数返回地址的恶意攻击行为，也可以检测出针对数据段上修改函数指针的恶意攻击行为。实际上，与其他缓冲区检测工具相比较， FRDIFTS 能够检测出多种类型的恶意攻击，具有普遍使用的特点，比如 FRDIFTS 能够检测出利用格式化字符串进行的攻击。这是很大的优势

另外，FRDIFTS 能够给出攻击的相关信息：污染发生的地址和污染源。对于普通的数据流跟踪分析有一个问题，就是在遇到函数调用的时候，就会造成无法定位信息，但是由于我们的 FRDIFTS 有库函数识别的能力，也就会给我们定位攻击信息提供了更多的方便。

对于某些控制依赖的问题，我们通过分析数据流图，可以辅助加强我们的实验结果，在图 4-1 中是我们分析 notepad.exe 得到的数据流图。

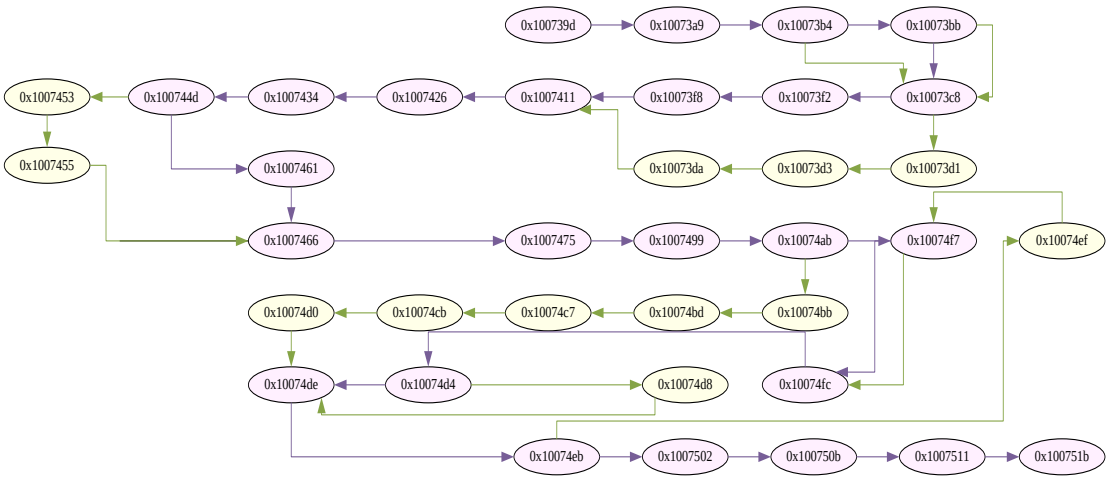

图 4-1 Notepad.exe 的数据流跟踪分析数据流图

Fig.4-1 Data Flow Graph of Notepad.exe

4.5 发现的漏洞

使用数据流跟踪分析系统共发现了 7 个来自文档处理程序的漏洞。表中给出的是发现漏洞的结果。总共我们的系统发现了 7 个确定的漏洞和 57 可能是漏洞的。

表 4-3 使用数据流跟踪系统所发现的漏洞

Table 4-3 discovered vulnerability by DRDIFTS

Word Processor	exploitable	maybe exploitable
MS Word 2003	0	1
MS Word 2007	0	0
MS PowerPoint 2003	0	1
MS PowerPoint 2007	0	0
Kingsoft wps	1	12
justsystem ichitaro	3	1
hangul HWP	3	42
Total	7	57

4.6 性能分析

图 4-2 中说明的是当使用我们的工具分析 SPEC CINT2006 标准测试程序时所得到的性能结果。结果表明函数识别模块的增加引起了比较小的开销。DynamoRIO 是前面提到的二进制翻译平台。从图中我们可以比较清楚的看出函数识别模块的增加并没有很大的增加程序的分析时间，也就是说我们的程序是比较实用的。主要的原因还是我们并没有识别所有的函数，而是将与我们的数据流跟踪分析相关的函数识别出来。

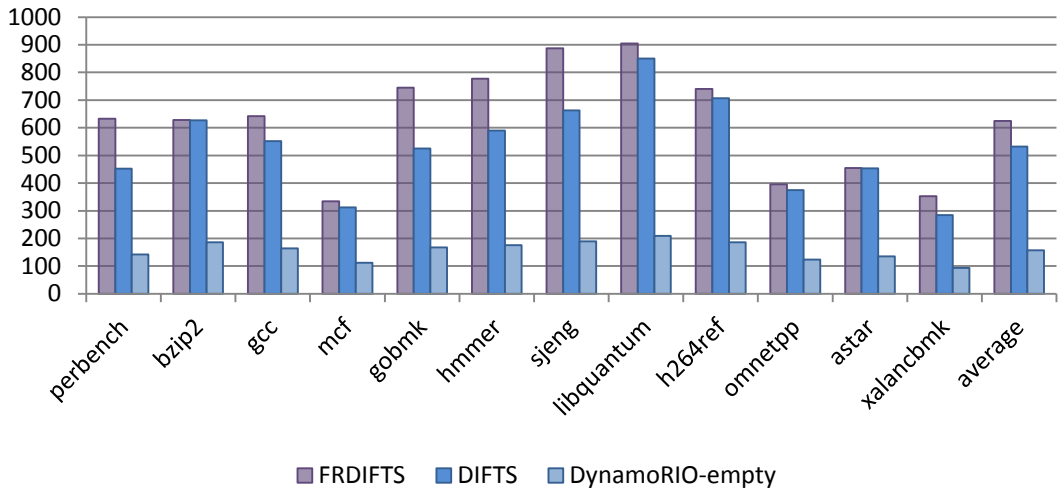

图 4-2 FRDIFTS 的性能分析。

Fig.4-2 Performance of FRDIFTS.

图中，FRDIFTS 表示的是 FRDIFTS 的各测试程序运行时间，DIFTS 表示的是数据流跟踪分析系统的各测试程序运行时间，DynamoRIO-Empty 表示的是没有其他功能的 DynamoRIO 跑这些测试程序的运行时间。

4.7 小结

本章在标准实验平台上，实验我们的系统，开始单独实验库函数识别的功能，然后给出一个例子具体说明系统工作的流程。然后对系统做出了功能分析，以及性能分析。我们的系统在标准试验平台上，性能表现良好，比其他系统的性能优越，并发现了真实的漏洞，并不只局限于缓冲区溢出一种漏洞，说明了系统的有效普适性。从实验结果看，本文的系统是有效实用的系统。

第五章 结论

本章总结全文对基于数据流跟踪分析框架的函数识别技术的研究及其优缺点，并给出了对未来工作的展望。

5.1 全文总结

当今有很多利用数据流跟踪分析来进行漏洞检测或者攻击防御的例子，但是利用缓冲区溢出来进行攻击的恶意行为仍然还是有很多。现在的缓冲区溢出检测工具有地需要源代码，不适用于现在源代码大部分都不容易获得的情况。动态二进制的数据流跟踪分析技术，就很好的解决的源代码不容易获得的问题。并且当前的缓冲区溢出工具大部分都是针对一种缓冲区溢出工具来工作的，那就有了很大的局限性，他只能检测一种缓冲区溢出，那么就不具有普遍适用的特点，人们使用起来就很不方便。而使用动态二进制数据流跟踪分析技术就很好的解决了这个问题，他跟踪不安全的数据，只要增加探测的规则就可以广泛的探测各种攻击行为，对于当前的攻击探测防御是非常好的。动态数据流跟踪分析技术不针对一种攻击方式，但是却能够有效地检测多种攻击形式，是比较实用的攻击防御，漏洞探测方式。动态二进制检测分析的技术现在越来越流行。

但是动态二进制数据流跟踪分析技术也有一些问题需要解决，比如在缓冲区溢出的探测方面，常常只能探测比较严重的溢出，对于一些轻微的溢出，虽然没有改变其中的控制流，但是对于程序的数据还是有影响，所以还是应该警惕，而且这种也说明能够发生溢出，这里有漏洞。我们应该注意此类情况，所以我们加入了库函数识别来增加动态二进制数据流跟踪分析的准确度。

现在有很多动态二进制工具，都是在一些动态二进制探测框架上面完成的。本文基于开源二进制翻译平台 DynamoRIO，构建了动态二进制数据流跟踪分析系统，并在此系统框架的基础上结合函数识别技术，改进了数据流跟踪分析的准确性。

5.2 未来工作的展望

动态二进制数据流跟踪分析系统提供了基本的数据流跟踪分析功能，但是很多地方还可以改进，比如目前小组正在进行数据流跟踪分析系统的继续改进。由于库函数识别功能的加入给系统在性能上面带来了一些损失，特别是在函数签名的提取上，耗费了大量的时间和空间。虽然我认为库函数识别在数据流跟踪分析方面的引入时非常有意义的，但是我目前做的工作还远远不够，还没有把库函数识别做到最符合数据流跟踪分析的方式，特别是动态二进制的。所以我再次提出一个想法就是利用二进制比对的方式来进行库函数识别，这里虽然想法很不成熟，但是我认为之后可以一试。为什么想要进行二进制比对法，因为对于上面签名提取的方法对于反编译是非常实用的，但是根据我们的目的数据流跟踪分析的目的，我们更倾向于方便快捷的方法来完成我们的函数识别。所以一些比较简单而且省时的方法，比如二进制比对的方法，就很适用于动态二进制数据流跟踪分析。

参考文献

[1] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In Proceedings of the 2001 USENIX Security Symposium, Washington DC, USA, August 2001.

[2] D. E. Denning. A lattice model of secure information flow. Communications of the ACM, 19(5):236–243, 1976.

[3] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. Communications of the ACM, 20(7):504–513, 1977.

[4] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In POPL, 1998.

[5] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, Omri Weisman. TAJ: effective taint analysis of web applications. PLDI, 2009.

[6] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. ACM Transaction on Software Engineering and Methodology, (4):410–442, 2000.

[7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In USENIX Security, 2004.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In SOSP, 2005.

[9] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proceedings of NDSS'05, San Diego, California, USA, February 2005.

[10] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In USENIX Security, Aug 2006.

[11] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In MICRO-37, Dec 2004.

[12] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, 49(14), 1998.

[13] Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J. 2000b. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability

Conference and Exposition, 119—129, 2000.

[14] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. IEEE Software, 19(1):42–51, February 2002.

[15] D.Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Proceedings of Network and Distributed System Security Symposium, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.

[16] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In Proceedings of the 2001 USENIX Security Symposium, Washington DC, USA, August 2001.

[17] Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In Proceedings of the USENIX Annual Technical Conference, 2000.

[18] Jedidiah R. Crandall , Frederic T. Chong, Minos: Control Data Attack Prevention Orthogonal to Memory Model, Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, p.221-232, Portland, Oregon., December 04-08, 2004.

[19] N. Nethercote. Dynamic Binary Analysis and Instrumentation. PhD thesis, University of Cambridge, United Kingdom, November 2004.

[20] Yin Liu, Ana Milanova: Static analysis for inference of explicit information flow. PASTE 2008.

[21] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, Youfeng Wu: LIFT: A low-overhead practical information flow tracking system for detecting security attacks. MICRO 2006.

[22] Lap-Chung Lam, Tzi-cker Chiueh: A General Dynamic Information Flow Tracking Framework for Security Applications. ACSAC 2006.

[23] Xia Bing. The Research and Implementation of Defense of Buffer Overflow Virtual Machine. Master’s thesis, Guangxi Normal University, 2003.

[24] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary Mc-Graw. "ITS4: A Static Vulnerability Scanner for C and C++ Code." In 16th Annual Computer Security Applications Conference (ACSAC 2000), December 2000.

[25] David Evans , John Guttag , James Horning , Yang Meng Tan, LCLint: a tool for using specifications to check code, ACM SIGSOFT Software Engineering Notes, v.19 n.5, p.87-96, Dec. 1994.

[26] Crispin Cowan , Calton Pu , Dave Maier , Heather Hintony , Jonathan Walpole , Peat Bakke , Steve Beattie , Aaron Grier , Perry Wagle , Qian Zhang, StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks, Proceedings of the 7th conference on USENIX Security Symposium, 1998, p.5-5, 1998, San Antonio, Texas, January 26-29,.

[27] Cowan, C., Beattie, S., Johansen, J., and Wagle, P. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In Proceedings of the 12th USENIX Security Symposium, Washington, DC. 91--104. 2003.

[28] RAD: A Compile-Time Solution to Buffer Overflow Attacks, Proceedings of the The 21st International Conference on Distributed Computing Systems, p.409, April 16-19, 2001.

[29] Vendicator. Stack Shield technical info file v0.7. http://www.angelfire.com/sk/stackshield/, January 2001.

[30] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2000.

[31] George C. Necula , Jeremy Condit , Matthew Harren , Scott McPeak , Westley Weimer, CCured: type-safe retrofitting of legacy software, ACM Transactions on Programming Languages and Systems (TOPLAS), v.27 n.3, p.477-526, May 2005.

[32] Parasoft. Insure++: Automatic runtime error detection. http://www.parasoft.com, 2004.

[33] Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In Proceedings of he USENIX Annual Technical Conference, 2000.

[34] Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. White Paper http://www.research.avayalabs.com/project/libsafe/, December 1999.

[35] Julian Seward , Nicholas Nethercote, Using Valgrind to detect undefined value errors with bit-precision, Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference, p.2-2, Anaheim, CA. April 10-15, 2005.

[36] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In PLDI, Jun 2000.

[37] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators, 2003.

[38] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S.Wallace, V. J. Reddi,

and K. M. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In PLDI, Jun 2005.

[39] Prashanth P. Bungale , Chi-Keung Luk, PinOS: a programmable framework for whole-system dynamic instrumentation, Proceedings of the 3rd international conference on Virtual execution environments, San Diego, California, USA. June 13-15, 2007.

[40] N. Nethercote and J. Seward. Valgrind: A program supervision framework. Electroic Notes in Theoretical Computer Science, 89(2), 2003.

[41] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In CGO, 2006.

[42] M. Burrows, S. N. Freund, and J. Wiener. Run-time type checking for binary programs. International Conference on Compiler Construction, April 2003.

[43] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. ENTCS, 89(2), 2003.

[44] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, Youfeng Wu: LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. MICRO 2006.

[45] 宋奕青，基于动态二进制探测框架的缓冲区溢出检测研究，上海，上海交通大学，2010

[46] J. K. Donnely , A decompiler for the countess computer, Navy electronics laboratory technical memorandum 427,1960

[47] Cristina Cifuentes , Reverse Compilation Techniques, Ph.D. thesis, Queensland University of Technology, Australia, 1994

[48] 胡政, C++逆编译中模板库函数识别研究,中国科学技术大学计算机科学与技术系,2006

[49] Ilfak Guilfanov, Fast Library Identification and Recognition Technology,1997

致谢

首先感谢我的导师戚正伟老师。在我攻读硕士研究生的两年半时间里，戚老师给了我多方面的指导和关心。戚老师不仅在专业知识、科研方法上言传身教，更以他自己的行为作为榜样，帮助我树立了正确的治学态度乃至对待人生的态度在两年的时间里，戚老师给我们学术上的指导，使我们的学术视野开阔了许多。在平时的项目中，戚老师在每次的例会中都常常提出一些很有见地的想法，并且鼓励大家开放思路积极思考，是我们整个小组的学术氛围十分浓厚，并且戚老师还鼓励我们小组之间多交流，我们在小组之间的交流过程中也产生了很多有意义的想法。在项目进展不顺利的时候，戚老师并不会严厉的指责我们，而是和我们一起想办法解决，在戚老师鼓励以及指导下，我们走过了很多困难的时刻，目前小组进展良好，很多良好的传统都继续了下去。而且在我开题的迷茫阶段，在我们的论文未能投中的时候，戚老师始终坚信我们的能力，始终支持我们，并提出了很多宝贵的修改意见和建议；在我们的项目取得一点点进展的时候，戚老师给了我们最及时的勉励，让我在研究生阶段取得了很大的进步。

值此学业完成之际，我谨向戚老师表示我最最深切的敬意和感谢，没有戚老师的指导，我不能如此顺利的完成我的研究生生涯，更我不能这么顺利的完成我的毕业设计。在项目组中，戚老师给我留下认真、严谨、思路清晰灵活的印象，给我的学术研究以很多指导。在我的科研工作中，戚老师给了我很多宝贵意见，不管是学术进展的大方向还是某些技术细节，戚老师都能够给出非常有价值的指导性意见，令我受益匪浅。戚老师的学术研究也非常广泛，在很多领域，戚老师都有比较深的研究，在我进行毕业设计的过程中，给了我很多指导和启发，使得我能够顺利的毕业设计。

梁老师在我的学习、工作中都给予了很大的帮助，他清晰的思路，严谨认真的态度都给我以很大的指导以及启发，最深刻的印象就是梁老师的课堂上，不光学到了学术知识，而且同时也学到了人生的道理，令我受益匪浅。

同时感谢项目组的所有同学，包括我的前辈学长们，在我刚进入项目组的时候，是你们给了我很多的指导和帮助，使我的科研渐渐走入轨道。

当然我还要感谢王卓同学、张若愚同学、高尚同学、林芊同学、俞培杰同学、朱旻同学、倪康奇同学、黄山同学、黄实秋同学，他们在我两年的研究生生涯中给予的很多的帮助，刚进入项目组时，我们互相帮助，建立的共同进步的学术氛

围，在整个研究生生涯中，同学们都很热情的帮助我。在我们的项目开发及论文写作的整个过程中，我们互相讨论，在我的毕业论文写作过程汇总，你们不断地给我提出宝贵的意见，给予了理论和技术上的强大支持，这种互相探讨、共同进步的研究氛围，给我创造了进步的空间。

感谢嵌入式实验室、程序分析小组以及 Z0803791 班级的所有同学，在两年半的学习生活中，你们给予的我很多帮助与支持，很高兴我能融入这样的集体，在这样的氛围中，我拥有了很多的快乐和美好的回忆。

最后感谢我的父母和家人以及朋友，他们永远是我最坚实的后盾，他们一直无怨无悔的从各个方面支持我的学习，并且在我迷茫的时候支持我走过低潮期，在我有所进步的时候，他们告诉我不要骄傲，要继续努力，以取得更大的进步，所以我非常的感谢他们，我学业上的所有成绩都离不开他们对我的支持与鼓励。

周侃

2011 年 01 月

工程硕士期间发表论文

[1] Kan Zhou, Shiqiu Huang, Zhengwei Qi, Jian Gu, Beijun Shen, Enhance Dataflow Analysis with Function Recognition, E-Forensics 2010（EI）, 上海, 中国, 录用.

[2] Kan Zhou, Shiqiu Huang, Shan Huang, Zhengwei Qi, Jian Gu, Haibing Guan, Enhance Dataflow Analysis with Function Recognition, China Communications(SCIE 源期刊), China, 2010 年 12 月第 6 期. ISSN1673-5447. 发表.

[3] Gengbiao Chen, Zhuo Wang, Ruoyu Zhang, Kan Zhou, Shiqiu Huang, Kangqi Ni, Zhengwei Qi, Kai Chen, Haibing Guan, A Refined Decompiler to Generate C Code with High Readability, 17th Working Conference on Reverse Engineering (WCRE 2010). 录用.