北京大学

博士学位论文

题目：面向二进制程序的漏洞挖掘关键技术研究

姓　　名：王铁磊

院　　系：信息科学技术学院

专　　业：计算机应用技术

研究方向：Internet与信息安全

导师姓名：邹维研究员

二零一一 年 四 月

版权声明

摘　要

随着计算机在国民经济和国防建设各个领域的广泛应用，作为信息系统智能载体的计算机软件的安全性变得尤为重要，软件安全漏洞已经成为信息安全风险的主要根源之一。由于软件安全漏洞的危害性、多样性和广泛性，在当前网络空间（Cyber Space）的各种博弈行为中，漏洞作为一种战略资源而被各方所积极关注。如何有效的发现漏洞、消除或减少漏洞对社会生活、国家信息安全的负面影响，即漏洞挖掘和防护工作已经成为世界各国在信息安全领域的研究重点。

面向源程序(Source Code)漏洞挖掘的研究历史已经相当悠久并取得显著进展。然而，出于商业利益、知识产权保护等原因，大部分软件厂商不对外提供源程序；有些厂商虽然提供了源程序，但源程序和实际系统之间的对应关系无从验证。另外，源程序级的漏洞分析也无法发现编译、链接过程引入的漏洞。因此直接面向二进制程序的安全漏洞挖掘研究具有重要的意义。

然而，面向二进制程序的安全漏洞挖掘面临更多的挑战。一方面，对于复杂机理的安全漏洞，基于静态程序分析的漏洞挖掘误报率高，往往需要大量人工分析来验证挖掘结果的正确性，严重制约了静态挖掘技术的应用范围。虽然以符号执行(Symbolic Execution)为代表的路径敏感分析方法可以缓解高误报率问题，但是又面临执行路径组合爆炸这一公认难题。

另一方面，基于模糊测试(Fuzz Testing)的动态漏洞挖掘技术虽然误报率低，但是其有效性完全依赖于测试数据的生成。尽管研究人员相继提出各种测试数据生成方法，对于复杂的未公开数据格式，特别是含有完整性校验信息的数据格式，现有测试数据生成方法存在严重不足，无法对目标程序有效测试。

本文围绕二进制程序漏洞挖掘问题，深入分析了动静态漏洞挖掘技术的优劣及面临的主要挑战，重点研究了动态漏洞挖掘过程中如何加强畸形样本在程序执行中的纵深传递和静态漏洞挖掘过程中漏洞建模与空间遍历方法。本文主要贡献如下：

1. 首次提出校验和（checksum）感知的模糊测试方法。模糊测试作为软件安全

漏洞挖掘的重要方法，在遇到校验和检测机制时往往无能为力。本文综合运用细颗粒度污点分析、程序执行路径差异定位、离线符号执行等方法，首次提出一种绕过校验和检测机制的模糊测试方法。该方法不需要访问程序源代码，也不依赖于具体数据格式及校验和算法，为运用模糊测试发现深藏的软件漏洞扫除了障碍，拓宽了传统模糊测试的应用范围。

2. 提出了一种基于roBDD的离线细颗粒度污点分析方法。过大的内存消耗和低下的分析效率制约了细颗粒动态污点分析的应用。本文分析了细颗粒动态污点分析的瓶颈所在，提出了一种基于roBDD （Reduced Ordered Binary Decision Diagram）的离线细颗粒度离线污点分析方法，提高了细颗粒度污点分析的性能，降低了内存需求，为在畸形测试例生成和校验和检查穿透等方向应用细颗粒度污点分析奠定了基础。

3. 为进一步提高模糊测试的挖掘效率，提出了一种基于细颗粒度污点分析(Fine-grained Taint Analysis)的导向性样本生成方法。该方法自动识别影响安全敏感操作的输入数据片段，进而对这些片段加以扰动生成畸形测试数据。与传统模糊测试技术相比，该方法避免了对目标程序整体输入空间的盲目枚举，生成的测试用例能够直接影响安全敏感操作，提高了动态漏洞挖掘的效率。

4. 为进一步加强模糊测试的安全分析能力，提出了一种基于混合符号执行(Conclic Execution)的智能样本生成方法。基于混合符号执行和约束求解技术，该方法不仅能够判断执行轨迹上是否存在潜在的安全漏洞，提供了对执行路径深度分析的能力，同时也可以对轨迹上的约束条件逐一取反并求解生成遍历不同执行轨迹的新测试样本，为提高模糊测试的代码覆盖率提供了支撑。

5. 提出了一种面向脆弱性包络(Vulnerable Component)的整数溢出漏洞静态挖掘方法。针对整数溢出漏洞的特点，提出了一种整数溢出漏洞模型，并以该模型为指导提出了二进制程序中脆弱性包络自动识别方法。与传统静态挖掘工具力求遍历所有执行路径不同，该方法重点分析脆弱性包络中的程序路

径，显著降低了路径总数，有效缓解了路径爆炸问题。

6. 设计实现了软件安全漏洞动态挖掘系统TaintScope和整数溢出漏洞静态挖掘系统IntScope，并应用这些系统在Microsoft、Adobe、Google等著名IT公司的产品中发现数十个零日漏洞，大部分漏洞已被中国国家信息安全漏洞库(China National Vulnerability Database of Information Security)、国际安全漏洞机构CVE (Common Vulnerabilities and Exposures)等权威漏洞管理组织收录。

关键词：漏洞挖掘，程序分析，信息安全

Research on Binary-Executable-Oriented Software Vulnerability Detection

WANG Tielei (Computer Science)

Supervised by **ZOU Wei**

Abstract

As computers have been widely used in the national economy and defense, software systems, the intelligent components in computers, play a more and more important role in information security; in fact, software vulnerabilities have become a root cause of various threats to information security. Since vulnerabilities widely exit in diverse software and can cause significant damages, software vulnerability has been considered an important kind of strategic resources and attracted the widespread attention during the competition game in cyberspace. How to identify and eliminate vulnerabilities and reduce their negative influences on social life, i.e., vulnerability detection and protection, have been the focus of security research.

Source-code-oriented vulnerability detection has been researched for dozens of years and made great achievements. However, due to the protection of commercial benefits and intellectual properties, most vendors do not provide the source code of their productions. Even that a few vendors may provide source code, end-users cannot verify whether the executables are compiled from the source code. More importantly, source-code-oriented vulnerability detection cannot identify vulnerabilities introduced during compiling-time and linking-time. In a word, binary-executable-oriented vulnerability detection is more meaningful and useful.

However, detecting vulnerabilities on the binary program level faces more challenges. On one hand, for the complex vulnerabilities, static-analysis-based detection techniques usually generate too many false positives, and need manual inspection to verify the detection results, which significantly constrains the scope of their application. While path-sensitive analysis methods such as symbolic execution can reduce false positives, they have to deal with the path-explosion problem when analyzing large applications. On the other hand, dynamic vulnerability detection methods such as fuzz testing rarely have false positives, but the effectiveness of these methods depends on how to generate test cases. For complicated file formats, especially, which employ checksum-based integrity verification mechanisms, existing test case generation methods have serious drawbacks and cannot effectively test the target applications.

This dissertation focuses on binary-executable-oriented vulnerability detection techniques. The contributions of this dissertation can be summarized as follows:

1. A checksum-aware fuzzing method is proposed. Fuzzing is an important kind of vulnerability detection methods. However, when encountering checksum-based integrity check mechanisms, traditional fuzzing does not work well any more. This dissertation synthetically employs fine-grained taint analysis, execution trace differences identification and offline dynamic symbolic execution, and then proposes the checksum-aware fuzzing method. Checksum-aware fuzzing can address the issues caused by checksum mechanisms for traditional fuzzing tools. The maid advantage of this method is that it depends neither on program source code, nor on specific checksum algorithms.

2. A roBDD-based fine-grained offline taint analysis algorithm is developed. Extremely high memory usage and low analysis performance severely limit the use of fine-grained taint analysis. The bottleneck of fine-grained taint analysis is figured out and a roBDD-based fine-grained off-line taint analysis approach is proposed. The experiment results show that our approach can significantly

improve the performance of fine-grained taint analysis, and reduce the memory usage.

3. A taint-based directed fuzzing method is designed and implemented. Directed fuzzing automatically tracks the propagation of input data, identifies the input bytes which flow into security-sensitive operations. Furthermore, directed fuzzing modifies such input bytes to construct malformed inputs. The mutations preserve the syntactic structures of original inputs, but hold malformed values in some important fields. Compared with traditional fuzzing methods, directed fuzzing can avoid enumerating the whole input space and improve the effectiveness of fuzzing.

4. A dynamic symbolic-execution-based fuzzing method is developed. Symbolic-execution-based fuzzing first records the execution trace when the target program runs with well-formed inputs; then, symbolic-execution-based fuzzing treats input data as symbolic values, replays the trace and gathers trace constraints on input data; furthermore, symbolic-execution-based fuzzing checks whether the trace constraints can protect the trace from stack overflows, heap overflows and integer overflows. Symbolic-execution-based fuzzing provides a deep analysis on a single trace and nicely complements directed fuzzing

5. A static-symbolic-execution-based integer overflow vulnerability detection technique is designed. The dissertation summarizes the features of integer overflow vulnerabilities and proposes a source-sink model to describe them. According to the model, the dissertation designs an algorithm to automatically identify vulnerable components in a binary program. Based on static-symbolic-execution and constraint solving, the dissertation proposes an algorithm to detect integer overflows in these vulnerable components. Compared with traditional path-sensitive analysis methods, our algorithms only traverses the paths in the components instead of the whole execution space, and significantly

alleviates the path-explosion problem. Meanwhile, since infeasible paths are pruned, this method has very low false positives.

6. The details of dynamic fuzzing platform TaintScope and integer overflow detection system IntScope are described. We apply the two systems to dozens of widely used applications, and we have detected a number of zero-day vulnerabilities. Most of these vulnerabilities have been confirmed and collected by CVE (Common Vulnerabilities and Exposures) and China National Vulnerability Database (CNNVD).

Keywords: Vulnerability Detection, Program Analysis, Information Security

目 录

插　图

表　格

第一章　绪论

1.1　课题背景与研究意义

随着计算机系统在国民经济和国防建设各个领域的广泛应用，作为信息系统智能载体的计算机软件的安全性变得尤为重要，已经成为信息系统安全的基础。但是，计算机软件系统的规模和复杂性都快速增长的同时，不可避免地存在安全漏洞。这些安全漏洞给国民经济和社会生活带来深远影响。例如，美国国家标准技术研究院（NIST）在2002年的研究报告中指出 [155]，软件缺陷给美国造成的经济损失达595亿美元（占美国当年GDP的0.6%）；到2007年，这一数字已经增长至1800亿美元 [137]。更为严重的是，因软件漏洞导致的蠕虫、网马(driven-by downloads)、僵尸网络(Botnet)等各种网络攻击事件给互联网甚至给国家信息安全带来严重危害。

由于软件安全漏洞的高危害性、多样性和广泛性，在当前网络空间（Cyber Space）的各种博弈行为中，软件漏洞作为一种战略资源而被攻防双方积极关注。漏洞挖掘和防护已经成为世界各国在信息安全领域工作的共识和重点。2003年，美国政府在发布的国家安全战略报告《The National Strategy to Secure Cyberspace》中指出了网络安全防御面临的诸多问题，并重点指出由软件安全漏洞问题引发的严重后果，特别强调了网络和软件安全漏洞问题的重要性。美国早在上世纪九十年代就提出了信息对抗和网络中心战的概念，以高校、工业界和军方为主要研发力量，在软件漏洞分析领域开展了大量相关研究工作。

软件安全漏洞正在成为我国的一种重要信息战略资源。2006年，中共中央办公厅、国务院办公厅发布《2006-2020年国家信息化发展战略》，在第四章“我国信息化发展的战略重点”中明确指出“积极跟踪、研究和掌握国际信息安全领域的先进理论、前沿技术和发展动态，抓紧开展对信息技术产品漏洞、后门的发现研究，掌握核心安全技术，提高关键设备装备能力，促进我国信息安全技术和产业的自主发展”。2009年10月中国信息安全测评中心宣布国家漏洞库投入运行，1个月内

有关机构和组织又相继发布了中国信息安全漏洞库、国家安全漏洞库，表明我国对安全漏洞的认识和管理进入了一个新阶段。

软件安全漏洞挖掘技术是漏洞发现、消除及防护工作的核心所在。在程序语言设计、程序分析、软件工程等领域的推动下，面向源代码的软件漏洞挖掘技术取得显著进展。然而，面向源代码的软件漏洞挖掘技术存在诸多不足 [27]。出于商业利益保护等原因，大部分软件厂商不对外提供源程序，有些厂商虽然提供了源程序，但和实际系统之间的对应关系无从验证。另外，源程序级的漏洞分析也无法发现编译、链接过程引入的漏洞。结合我国软件产业现状，软件安全漏洞挖掘技术不仅要研究自主开发的软件，也要研究系统中的第三方软件及构件；不仅要研究软件设计中的安全隐患，更重要的要研究软件编码实现阶段引入的安全性问题；不仅要对源码进行研究，也要对最终的可执行码进行研究。简言之，直接面向二进制程序的漏洞挖掘技术具有重要意义。

1.2 漏洞挖掘研究现状

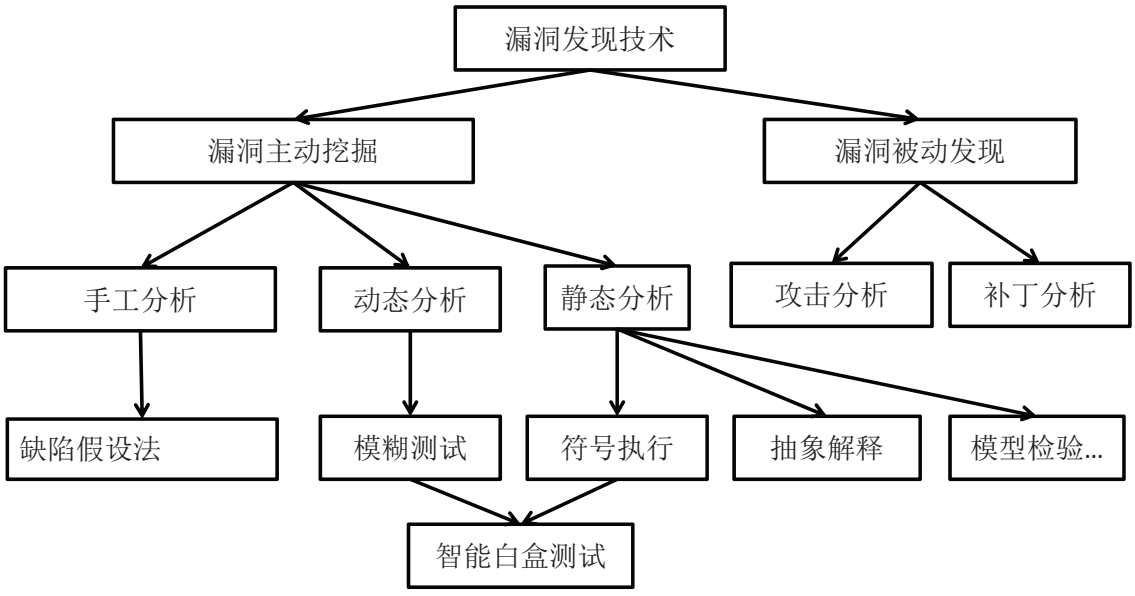


图 1.1: 漏洞挖掘技术概览

漏洞挖掘技术与软件安全测试、渗透攻击与防护及程序分析等领域密切相关。从研究客体上可以将现有漏洞挖掘技术分为两类：以软件为客体的漏洞挖掘，

即从软件中挖掘未知漏洞，本文称之为“漏洞主动挖掘技术”；其次是以漏洞为客体的漏洞分析技术，即根据漏洞的部分相关信息，进一步复原漏洞的完整信息，本文称之为“漏洞被动发现技术”。图1.1给出了漏洞挖掘技术的概览。

1.2.1 漏洞被动发现技术发展

漏洞被动发现技术指的是根据部分漏洞信息恢复漏洞全部信息的技术。常见的场景是根据捕获到了攻击样本或者开发方发布的安全补丁进而及时有效地分析漏洞的详细信息，由此发展出基于攻击分析(Attack Analysis)和基于补丁分析(Patch Analysis)的漏洞发现技术。

1. 基于蜜罐/蜜网捕获攻击样本是攻击分析的基础手段。蜜罐技术在90年代起开始在计算机安全界推广应用。2001年，L. Spitzner[148]给出了蜜罐的定义：蜜罐是一种安全资源，其价值在于被扫描、攻击和攻陷。蜜罐的核心价值就在于对这些攻击活动进行监视、检测和分析。2005年，J. R. Crandall等人提出了Epsilon-Gamma-Pi网络攻击模型，并设计了Minos[68]，它可利用虚拟机蜜罐捕获蠕虫的攻击。2006年，Yi-Min Wang等人提出了HoneyMonkeys[161]，利用浏览器蜜罐来捕获针对浏览器的攻击，进而发现浏览器的漏洞。他通过长时间监视挂马网页，发现挂马组织会迅速利用最新的零日漏洞（zero-day vulnerability）进行攻击，因此通过监控挂马页面发现零日漏洞，是一种非常有效的方法。2007年，C. Leita等人发布了SGNET[105]，系统化的利用蜜网来捕获零日漏洞。

使用蜜罐捕获攻击后，恢复零日漏洞核心信息的关键步骤是针对漏洞攻击的逆向分析。2004年，G. Edward Suh等人提出了基于动态污染跟踪技术的漏洞攻击分析方法[151]。2005年，J. R. Crandall等人继Minos之后提出了DACODA[67]，利用符号执行技术分析Minos捕获的攻击中所利用的漏洞。2006年，G. Portokalidis等人提出了Argos系统[134]，利用动态污染跟踪和shellcode替换等技术对漏洞利用进行分析。在上述针对漏洞攻击的逆向分析研究中，执行轨迹重放和动态污染跟踪技术成为关键的核心技术。

2. 基于补丁分析的漏洞发现技术主要依赖于软件开发方发布的补丁信息，进而定位和恢复被修补的漏洞信息。这类方法通常也被称为二进制比对技术。

二进制比对技术目的在于定位补丁修复点。2004年，H. Flake等人首先提出了针对可执行文件的结构化比对方法[81]，但该方法局限于控制流图的同构比较，在高级编译优化后这种结构很容易发生变化，效果并不好。2008年，H. Park等人提出了基于结构化和常量传播信息的二进制比对方法SCV[130]，一定程度上提高了定位的准确性。同年，Debin Gao等提出了BinHunt[87]，实现了更完备的基于语义的二进制比对方法，但分析性能较低，尚且无法应用于大型应用程序。

定位补丁修复点之后，为进一步确定漏洞信息，研究人员提出了不同的探索性方法。D. Brumley等人在符号执行技术基础上，根据补丁信息自动生成测试样本[42]，力图构造能够触发未修补的二进制程序中安全漏洞的样本。王嘉捷[3]提出了根据软件补丁应用切片和符号执行技术来分析相关漏洞。

总体而言，由于漏洞被动发现技术需要依赖于从第三方获得漏洞线索（例如捕获的攻击实例或补丁），制约了这类技术的应用范围。虽然补丁比对的研究相对成熟，已经有一些商业化产品，但如何在补丁比对基础上进一步分析漏洞的成因依然存在很多未解决的困难。二进制比对后的漏洞信息进一步恢复，等价于增加了已知约束和更多启发条件的漏洞挖掘问题，因此可以借鉴很多现有的漏洞挖掘技术。

1.2.2 漏洞主动挖掘技术

漏洞主动挖掘技术以软件为主体，不依赖于先验的漏洞线索知识，主动挖掘软件中的未知安全漏洞。近些年来漏洞主动挖掘技术不断发展，已经成为信息安全领域的核心能力之一。

对漏洞主动挖掘技术的分类维度很多[5]。从研究对象形态角度，可以分为面向源代码挖掘技术、面向二进制程序的挖掘以及不依赖于代码的黑盒测试技术[27]；从是否需要运行软件角度，可以分为静态漏洞挖掘和动态漏洞挖掘[78]；从程序分析理论角度，可以分为基于模型检验、抽象解释、符号执行等方法的挖掘技术[6]。从所分析程序语言角度，可以细分成面向各种语言规范的挖掘技术[150]：例如，面向C/C++、面向Java、面向Ada等程序语言的挖掘技术。此外，随着Web 2.0时代的到来，针对浏览器环境下脚本语言(例如PHP[101, 171]、

JavaScript语言[31, 32, 57])中跨站点脚本访问xss、SQL注入、信息泄漏等安全漏洞的研究也越来越多。

本文将漏洞主动挖掘技术分为三类：手工分析、动态挖掘技术和静态挖掘技术（见图1.1）。下文对每一大类技术做进一步探讨。

手工漏洞发现技术

在相当长的时间内，漏洞挖掘以手工分析为主。采用的主要方法是C. Weissman在1973年提出的缺陷假设法(Flaw Hypothesis Methodology, FHM)[166][167]，即首先根据系统的规范和文档估计系统中可能存在的缺陷或漏洞的列表，然后进行渗透攻击或代码分析以验证这些漏洞是否存在，如果验证了漏洞的确存在则进一步把缺陷模式普适化以发现系统中更多的缺陷或漏洞。手工分析主要使用的工具是软件逆向工程分析工具，包括商业化的反汇编器、调试器、切片分析等。手工分析极大的依赖于安全分析人员个人的能力和精力，而且在挖掘过程中产生的经验知识难以复用。由于针对软件漏洞的手工分析代价高昂，所以在学术界如何使用计算机辅助分析来进行漏洞挖掘一直是研究热点。

静态漏洞挖掘技术

静态漏洞挖掘是以漏洞模型为指导，运用程序静态分析技术发现程序中安全问题的过程。以程序静态分析为基础的静态漏洞挖掘方法的发展极为悠久。在词/语法分析、控制流分析、数据流分析、抽象解释(Abstract Interpretation)、类型理论、软件模型检验(Software Model Checking)等理论的支持下，静态漏洞挖掘取得了显著进展。

静态漏洞挖掘技术的分类维度也很多。例如传统编译领域[23]通常从数据分析精度角度，按流敏感(flow sensitive)、上下文敏感(context sensitive)及路径敏感(path sensitive)等维度对分析技术分类。本文将从安全漏洞刻画模型角度，对现有工作分类描述，并阐述现有工作的发展及不足。

1. 简单模式比对阶段。在缓冲区溢出漏洞泛滥初期，人们发现缓冲区溢出漏洞通常是由对字符串操作函数（例如strcpy，strcat）滥用引起的。为检测这类漏

洞，ITS4 [157]、FlawFinder [76]等词法分析工具遍历程序源代码，检测对字符串函数的不安全使用。然而，词法分析工具的误报率和漏报率都很高，一般只作为手工分析的起点，不再是研究的重点。

2. 数据流分析阶段。结合不同编程语言的特性，研究人员发现很多安全漏洞根源之一在于不良编程习惯引起。很多研究人员将编译过程中的数据流分析理论扩展应用于检测这一类别的安全问题。例如，在C/C++程序中，空指针解引用、内存多次释放等问题，是由于编程人员对指针变量不安全操作引起的，传统编译领域的数据流分析一定程度上可以描述和检测这类安全问题。

由美国斯坦福大学开发的MC[56, 52, 92, 25]系统，扩展了GCC编译器的数据流分析方法，可以根据用户指定的规则进行代码检查。在对Linux、FreeBSD等操作系统代码分析时，MC 从中发现大量指针操作的安全隐患。从MC孵化出的代码静态扫描软件Coverity[10]已经被工业界广泛应用。

对于格式化字符串漏洞类型，U. Shankar和D. Wagner等人在CQual类型推理系统[83]中应用结合污染分析思想来检测C语言中的格式化字符串漏洞[144]，但其作用受限于函数间数据流分析能力和指针别名分析能力。

3. 自动机可描述漏洞模式。为提高静态分析的可扩展性，很多研究人员采用自动机模型统一刻画不同类型的安全漏洞，进一步结合软件模型检测理论遍历代码空间，检测疑似安全漏洞，一时间涌现出很多典型的检测工具，例如UNO[93]、MOPS[53]、SLAM[30]。

时序安全漏洞（Temporal Vulnerability）是一种典型的可以用自动机刻画的漏洞类型。程序执行过程中很多操作（例如API调用）应该满足严格的时序关系；一旦执行过程中这种时序关系被破坏，就对应一个潜在安全问题。MOPS[54, 53]用下推自动机(Pushdown Automata)PDA表示程序控制流图，用有限状态自动机FSA刻画安全漏洞。下推自动机本质上是一个带有堆栈数据结构的有穷自动机。下推自动机的这种特性很容易用于表示函数间的调用关系。判断程序是否违反安全属性的问题转换为判断两个自动机求交是否为空的问题，MOPS系统能够做到检测结果无漏报（完备性），但是MOPS是数据流不敏感的分析，无法跟踪数据依赖关系，导致MOPS误报率很高。

与MOPS系统类似，SLAM[28, 30, 29] 是一种基于谓词抽象技术的软件模型检验器，主要用于检测Windows操作系统驱动程序是否满足用户定义的属性约束。SLAM的核心技术是谓词抽象，将每个变量都抽象为只有0/1两种取值。这种过近似(Over-Approximations)抽象方法导致SLAM误报较为严重。所以微软将SLAM集成到Windows驱动开发平台中，在驱动开发阶段生成编译警告，辅助开发人员发现安全隐患。

4. 复杂数据约束关系模式。很多安全漏洞背后存在复杂的数据约束关系，为了降低分析的误报率、排除程序中不可行路径、准确检测到这些复杂机理的安全漏洞, 研究人员意识到需要采用更复杂数据约束模型刻画漏洞，因此很多检测系统结合抽象解释理论以及符号执行和约束求解技术进行漏洞挖掘。

抽象解释理论[63, 66, 65]通过在抽象域上模拟程序的执行，获取程序真实执行的部分信息，在分析精确度和分析效率之间寻求平衡。例如，ASTR*É*E[36, 64]是基于抽象解释理论的程序静态分析器，抽象理论的创始人Patrick Cousot领导该项目。ASTR*É*E可以检测数组越界访问、除零异常、浮点运算溢出和整数运算溢出等问题。ASTR*É*E曾用于检验法国空中客车公司的空中巴士A340和A380系列飞机飞行控制软件[8]，受到工业界的认可。抽象表达提供了对变量边界值的估计，但无法有效追踪变量之间的约束关系，在路径可行性判定上依然存在不足。

符号执行技术[39, 102]采用抽象符号代替程序变量，根据程序的语义，遍历代码执行空间。本文将在第三章中详细讨论符号执行技术的发展和不足。总体而言，符号执行的主要优势在于能够发现变量之间运算关系，便于理解程序的内在逻辑；在漏洞挖掘时，有利于在复杂的数据依赖关系中发现数据之间本质的约束关系，而且符号执行精确记录了路径的约束条件，可以进一步用于判断路径可行性和路径约束的完备性。

早期静态符号执行典型系统包括Prefix[46]和ESP[71]。W. Bush等人在1998年提出了Prefix，并在美国申请了若干专利。微软公司于1999年收购了Prefix，在Prefix基础上实现了Prefast。Prefix/Prefast已经成为微软内部标准源代码静态检验工具之一。ESP[71]借鉴了MC由用户制定检查规则的思想，基于符号执行技术识别和保留路径上安全属性状态相关的约束条件，能够有效排除很多不可行

路径。

虽然符号执行技术如今已经广泛应用于漏洞挖掘领域，一个不可回避的问题是路径执行空间爆炸问题。程序的执行路径数目随着程序规模呈指数增长。穷举所有可能执行的路径是不可行的方式。如何缓解符号执行过程中的路径爆炸问题是当前的开放性难题。

动态漏洞挖掘技术

动态漏洞挖掘指利用程序运行时信息进行漏洞挖掘的过程。动态漏洞挖掘的本质是通过真实遍历程序状态空间，监测程序执行过程中是否会违背特定安全属性。

程序状态空间可以用二元组(c, M)描述，其中c代表处理器CPU所有可能的状态，M代表内存空间所有可能状态。从遍历程序状态空间的实现方式上，动态漏洞挖掘技术可以分为以下几类：事件空间遍历、输入空间遍历、程序执行路径遍历。

1. 事件空间遍历。对于事件驱动类型(event driven[122])的程序，通过枚举所有可能事件驱使程序执行到不同状态。这里的“事件”可以是收到网络报文、网络通信超时、内存分配失败等。CMC[122]就通过枚举这些不同事件，在Ad-hoc On-demand Distance Vector网络协议客户端中发现大量安全漏洞。

FiSC[173, 174]将CMC的思想应用到文件系统的安全性检测。FiSC从一个空的磁盘开始，在磁盘每一个状态下都枚举所有可能的操作（例如文件创建、删除，磁盘镜像挂载、卸载等），遍历所有可能的状态。同时FiSC保存已产生过的状态，避免重复遍历。FiSC在ext3、ext2、JFS等文件系统中发现了多个安全漏洞。

这类技术通常需要对目标程序有深刻理解，在应用过程中经常依赖目标程序的源代码，而且与目标程序的种类密切相关。

2. 输入空间遍历。通过构造海量输入数据，让目标程序处理海量数据，以此达到遍历程序状态空间的目的。这类技术也通常被称为模糊测试（Fuzz Testing）。本文在第四章详细阐述模糊测试技术的发展和不足。

模糊测试技术易于实现，能够快速部署实施，而且误报率低。所以模糊测试

技术在黑客社区有着广泛应用。特别是近些年，模糊测试技术已经被工业界广泛采纳，成为软件安全测试的重要手段。但是，模糊测试的核心问题在于，如何保证海量数据能够驱使程序遍历不同的状态空间。人们发现很多情况下，模糊测试技术并不奏效，无法对目标程序进行有效测试。究其原因，研究人员发现基于校验和（checksum）的数据完整性检验机制成为模糊测试的主要障碍。如何克服完整性校验机制，使畸形样本遍历不同的执行空间，是当前研究的难点。

3. 程序执行路径遍历。这类方法力图通过驱动程序执行不同路径以达到遍历程序状态空间的目的。这类技术也被称为测试例生成(Test Case Generation)技术。

在混合符号执行技术及细颗粒度污点分析的支持下，测试例生成技术近些年来取得了显著的进展。斯坦福大学和微软的研究人员，分别独立提出了混合符号执行（conclic symbolic execution）的概念，将符号执行从一种程序静态分析技术转变为动态分析技术，实现了一系列代表工具，如KLEE[49]、EXE[50]、Cute[143]、DART[90]、SAGE[91]等。在混合符号执行基础上，这些系统收集程序路径约束条件，通过约束求解生成驱动程序执行其他路径的测试例，达到遍历代码执行空间的目的。

事实上，很多研究并不严格区分测试例生成技术和模糊测试技术，而是在模糊测试的框架下，引入测试例生成方法提高模糊测试的挖掘效率。但是，一旦目标程序中路径约束过于复杂，包含大量复杂地非线性运算操作时，现有求解系统并不能满足约束求解的需求。更为严重的是，一旦路径上的约束条件涉及到单项散列运算（例如校验和计算），现有约束求解器根本无法求解约束条件。

国内在软件安全漏洞挖掘方向比较活跃的机构包括中国信息安全测评中心、中国人民解放军总参谋部、北京大学、中国科学院软件所、北京邮电大学、解放军信息工程大学、国防科学技术大学、北京理工大学、哈尔滨工业大学、武汉大学等机构和院校。这些单位在漏洞挖掘和管理等方面取得了一定的成绩，但在零日漏洞发现的数量、国际顶级会议论文数、漏洞库的有效利用、漏洞响应管理等方面与美国相比还有较大距离。

在工业界，神州绿盟、启明星辰等信息安全企业在漏洞挖掘方向积累了大量

经验，开发了很多已知漏洞扫描系统，为信息产品安全评估提供了强有力的技术支撑。但是在漏洞主动挖掘方向，仍然主要依赖于安全专家手工分析。

1.2.3 研究必要性

虽然软件漏洞挖掘技术经过几十年的发展，积累了丰富的理论知识和技术手段，但是总结上述相关研究，可以得到如下结论：

1. 软件安全漏洞挖掘的需求已经从面向源代码的分析转移至面向二进制程序的分析；但是，以往在源代码层积累的漏洞挖掘经验，很多时候并不能直接应用于二进制程序，二进制层的漏洞挖掘工作面临更多的挑战。

2. 基于程序分析理论的**静态漏洞挖掘技术**，需要与具体漏洞模型相结合，才能获得更好的检测效果。在处理漏洞挖掘问题时，特别是对于具有复杂数据依赖关系的漏洞类型时，静态分析虽然具有代码覆盖率高的优势，但同时存在高误报率的劣势。采用路径敏感的分析技术可以提高分析的准确性，但是路劲敏感分析技术同时会导致路径爆炸问题。如何利用漏洞特征提高静态漏洞挖掘的准确性是当前研究的重点和难点。

3. 以模糊测试为代表的**动态漏洞挖掘技术**已经被业界广泛应用，但是基于校验和（checksum）的数据完整性检验机制是现有模糊测试方法的主要障碍。一旦目标程序根据校验和判断数据的完整性，在协议未公开的情况下，现有模糊测试方法无法对目标程序有效测试。克服校验和问题是一个尚未解决的难题。进一步，一个畸形测试用例仅能测试一条程序执行轨迹，如何更有效的生成测试例遍历不同程序轨迹以及对特定执行轨迹深度安全分析面临诸多挑战。

1.3 本文工作及贡献

1.3.1 本文主要工作

本文主要工作如图1.2所示。本文以面向二进制程序的漏洞挖掘为总体目标，深入分析了动静态漏洞挖掘技术的优劣及面临的主要挑战，重点研究了动态漏洞

挖掘过程中如何加强畸形样本在程序执行中的纵深传递和静态漏洞挖掘过程中漏洞建模与空间遍历方法，以污点分析、符号执行、二进制反编译等技术为基础方法，提出了校验和感知的模糊测试、脆弱性包络提取、惰性检测策略等漏洞挖掘方法，设计实现了漏洞动态挖掘平台TaintScope和静态挖掘平台IntScope，运用这些系统在Microsoft、Google 、Adobe等公司的软件产品中发现大量安全漏洞，充分验证了方法和系统的有效性。

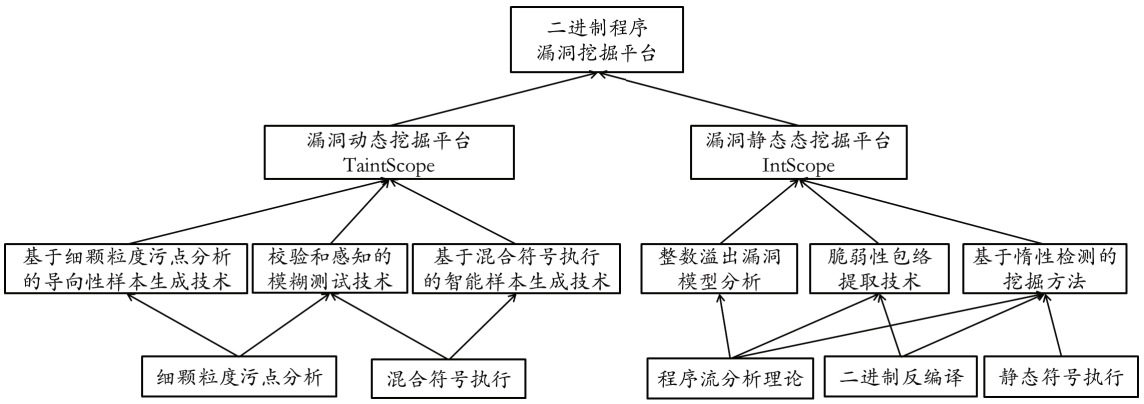


图 1.2: 本文主要工作

针对现有动态漏洞挖掘过程中无法处理基于校验和机制的完整性检测问题，本文综合运用细颗粒度污点分析、程序执行路径差异定位、混合符号执行等技术，首次提出一种绕过校验和防护机制的方法，加强了畸形数据在程序执行过程中的纵深传递，为运用模糊测试发现深藏的软件漏洞扫除了障碍。该方法不需要访问程序源代码，也不依赖于具体数据格式及校验和算法，自动化程度高。

进一步，为了使动态漏洞挖掘更高效地发现安全漏洞，本文设计实现了两种畸形样本生成技术。其中，基于细颗粒度污点分析的导向性样本生成技术以细颗粒度污点分析为基础，通过跟踪输入数据在程序执行过程中的传播，识别影响安全敏感操作的输入数据片段，进而对这些片段加以扰动生成畸形测试数据。由此生成的测试用例，在保留原始样本正常结构的同时，关键数据片段被修改为异常数值。与传统模糊测试技术相比，导向性样本生成技术避免了对目标程序整体输入空间的盲目枚举，生成的测试用例能够直接影响安全敏感操作，显著提高动态漏洞挖掘的效率。基于混合符号执行的智能样本生成技术首先记录目标程序处理

正常样本时的执行轨迹，进而采用符号执行技术，将输入数据视为符号值，根据每条指令的语义对执行轨迹进行重放；在重放过程中，一方面，结合整数溢出、栈溢出、堆溢出等漏洞类型的机理，检查路径约束是否完备，求解是否存在既满足路径约束又引发安全漏洞的输入实例；另一方面，对路径约束逐一取反并求解，生成新的测试用例遍历测试不同执行路径。作为导向性样本生成技术的互补方法，智能样本生成技术提供了对执行路径深度分析的能力。

针对静态符号执行面临的执行路径空间爆炸问题，本文根据整数溢出漏洞的特点，提出了一种整数溢出漏洞模型，并以该模型为指导设计了二进制程序中脆弱性构件自动识别方法；与传统静态挖掘工具力求遍历所有执行路径不同，本文方法重点分析脆弱性构件中的程序路径，结合基于惰性检测的挖掘方法，显著降低了路径总数，有效缓解了路径爆炸问题。

1.3.2 本文主要贡献

本文主要贡献总结如下：

1. 首次提出了一种校验和感知的模糊测试方法，为运用模糊测试发现深藏的软件漏洞扫除了障碍。该项研究发表于信息安全领域顶级会议IEEE Symposium on Security and Privacy 2010，并荣获最佳学生论文奖。

2. 提出了一种基于roBDD的离线细颗粒度污点分析技术，提高细颗粒度污点分析的性能的同时，降低了细颗粒污点分析的内存需求，为进一步扩大细颗粒度污点分析在漏洞挖掘领域的应用提供了途径。该项研究论文被北京大学学报录用。

3. 提出了一种基于细颗粒度污点分析的导向性样本生成方法，避免了对目标程序整体输入空间的盲目枚举，生成的测试用例能够直接影响安全敏感操作，提高动态漏洞挖掘的效率；提出了一种基于混合符号执行的智能样本生成方法，提供了对执行路径深度分析的能力。本文中提出的这两种样本生成方法的研究论文已被信息安全领域顶级期刊ACM Transactions on Information and System Security (TISSEC)录用。

4. 提出了整数溢出漏洞模型，并设计了整数溢出漏洞静态挖掘方法，通过对脆弱性包络的识别和遍历，有效缓解了符号执行面临的路径爆炸问题。该项研究论文发表于系统安全领域国际高水平会议Annual Network and Distributed System Security Symposium (NDSS) 2009。

5. 设计实现了软件安全漏洞动态挖掘系统TaintScope和整数溢出漏洞静态挖掘系统IntScope，并应用在Microsoft、Adobe、Google等著名IT公司的产品中发现数十个零日漏洞，大部分漏洞已被国家安全漏洞库、国际安全漏洞机构CVE（Common Vulnerabilities and Exposures）等国内外权威漏洞管理组织收录。

1.4 论文结构

本文后继章节安排如下：第二章介绍一种基于roBDD的离线细颗粒污点分析技术；第三章介绍离线混合符号执行系统的设计与实现；在这两章技术基础上，第四章详细介绍校验和感知的模糊测试技术；第五章将介绍基于反馈式的畸形测试例生成方法；第六章将围绕整数溢出漏洞，讨论如何提炼漏洞抽象模型，并在漏洞模型指导下进行快速漏洞检测；最后，第七章对本文进行总结，并对下一步工作进行展望。

第二章　基于roBDD的细颗粒度污点分析技术

2.1　引言

作为一种新兴的程序分析技术，动态污点分析（Dynamic Taint Analysis）已经广泛应用于信息安全很多领域，例如恶意代码分析[175, 146]，网络攻击检测与防护 [127]，软件安全漏洞挖掘 [158, 47]、协议格式逆向分析[48, 170, 106, 69, 62]等。动态污点分析的基本思想是在程序执行过程中，追踪特定数据片段（例如输入数据、密码）的传播过程，收集目标程序对这些数据的详细使用信息。例如，TaintCheck系统 [127]在程序执行过程中，跟踪网络报文数据的传播；一旦程序计数器PC依赖于网络报文数据，TaintCheck 生成一个攻击警报。

动态污点分析主要由两个部分构成：污点数据的标定和污点数据的追踪。污点数据的标定通常可以通过对特定系统API函数的劫持实现。但为了追踪污点数据，通常需要维护一个影子内存（shadow memory），用于记录内存地址和污点属性的映射关系。为便于表达，本章中将内存地址记为m，污点属性记为t。影子内存M 为m到t的映射，记为$\{m \mapsto t\}$。由于传统污点分析系统大多关注内存单元是否依赖于污点数据，布尔变量即可描述此类型的污点属性t。例如，$M(m) = 1$表示内存单元m依赖于污点数据，$M(m') = 0$表示内存单元m'不依赖于污点数据。换而言之，在传统污点分析中，所有的污点数据被赋予了同样的标签。

与传统污点分析不同，细颗粒度污点分析指为每个污点数据字节赋予唯一的标签，进而跟踪每个污点字节的传播过程。由于程序内部复杂的数据依赖关系，一个内存单元通常依赖于很多污点数据，因此需要用集合结构体来描述污点属性t。集合结构体的引入给细颗粒度污点分析带来严重影响。一方面，集合结构体的使用会显著增加内存消耗。另一方面，追踪污点数据传播的过程中，涉及到大量的集合合取（union）操作，这也将导致严重的性能损失。虽然很多污点分析系

统例如[48, 176]实现了细颗粒度污点分析的功能，现有研究中并未充分讨论如何降低分析过程中的内存消耗、提高细颗粒度污点分析的性能。

本章分析了细颗粒度污点分析面临的困难，提出一种基于有序二元决策图（Reduced Ordered Binary Decision Diagram，roBDD）的细颗粒度污点分析技术，实现了离线污点分析系统TaintReplayer。实验结果表明，该方法能够减低分析过程中的内存需求，显著提高细颗粒度污点分析的性能，为进一步扩大细颗粒度污点分析的应用提供了途径。

本章内容安排如下：第2.2节介绍了污点分析的研究背景，分析细颗粒度污点分析面临的困难；第2.3节介绍基于roBDD的集合表达的基本思想和理论背景；第2.4.2节给出本文中细颗粒污点分析系统的设计与实现；第2.5节给出系统性能的实验结果；最后，第2.6节对本章总结。

2.2　研究背景

污点分析本质上是一种信息流分析。D. Denning于1976年发表文献[72]，奠定了信息流分析的基础。信息流分析起初用于检查程序变量之间的信息传递是否违背安全约束，防止信息从高密级客体流向低密级客体。污点分析从信息流分析理论发展而来，其目的是跟踪特定数据片段在程序执行过程中的传播，为其他分析提供支撑。例如，程序对数据的访问过程蕴含了数据协议信息，这就为协议格式逆向分析提供了重要的帮助[48, 106]。

污点分析方法的工作方式可以分为静态方式和动态方式两类。静态污点分析以数据流分析或类型系统等理论为基础[98, 7]，通过分析程序代码而推衍污点数据的传播。动态污点分析在具体程序执行轨迹上结合运行时信息追踪污点数据的传播。进一步，动态污点分析系统在实现上可以分为两类 [4]：基于硬件支持的实现和基于源代码/二进制代码植入（code instrumentation）的实现。

污点分析主要由两部分构成：污点数据标定和污点数据传播追踪。此外，可以结合具体安全策略和分析需求，对污点数据的使用和传播加以限制。污点数据标定较为简单，通常可以对系统调用或特定API劫持完成。而污点数据传播追踪是污点分析的关键。为追踪污点数据的传播，通常要考虑数据依赖和控制依赖

关系[59]。由于数据依赖反应污点信息的直接传递，在污点分析过程中尤为重要。另外，最近一些研究[60, 99]也指出追踪控制依赖会导致大量误报，并提出缓解方法。

表 2.1: 污点分析示例

程序语句	传统污点分析	细颗粒度污点分析
read(fd, &x, sizeof(int));	x ↦ 1	x ↦ {0,1,2,3}
read(fd, &y, sizeof(int));	x ↦ 1 y ↦ 1	x ↦ {0,1,2,3} y ↦ {4,5,6,7}
z = x + y;	x ↦ 1 y ↦ 1 z ↦ 1	x ↦ {0,1,2,3} y ↦ {4,5,6,7} z ↦ {0,1,2,3,4,5,6,7}

表2.1给出了污点分析的一个示例。第一列表示相继执行的三行代码。假定文件句柄`fd`初始偏移为0，第一行代码负责从文件中读取4个字节至变量`x`，第二行代码继续读取4个字节至变量`y`，第三行将变量`x`和`y`求和后赋予变量`z`。第二列展示了传统污点分析过程。如果将`fd`作为污点数据源，变量`x`、`y`直接来源于污点数据源，所以被标示为污点数据；变量`z`数据依赖于变量`x`、`y`，也被标示为污点数据。第三列展示了细颗粒度污点分析过程。与传统污点分析不同，细颗粒度污点分析将每个污点数据（以byte为单位）赋予一个标签，所以变量`x`映射至集合{0,1,2,3}，类似地变量`y`映射至集合{4,5,6,7}。由于变量`z`同时数据依赖于变量`x`和`y`，执行第三句代码后，变量`z`依赖于集合{0,1,2,3,4,5,6,7}。

细颗粒污点分析需要标定和追踪每个污点数据单元。由于一个内存单元可能依赖于多个污点数据，因此需要用集合结构描述一个内存单元的污点属性t。假设污点数据长为m字节，那么以字节为单位，细颗粒度污点分析需要创建m个污点标签。进一步，假设分析过程采用整型变量表达污点标签，程序中有n个内存单元依赖于污点数据。最坏情况下，每个内存单元都依赖于m个污点标签，因此影子内存空间需求为：$n * m$*`sizeof(int)`。由此可见，细颗粒度污点分析的空间需求很大。

进一步，在细颗粒度污点分析过程中，涉及到大量集合合并操作。例如，`z = x+y`语句中，如果`x`和`y`都是污点数据时，需要合并两个变量对应的污点属性，

并赋予变量z。在x86机器指令层，存在大量的二元操作指令，包括算术运算指令和逻辑运算指令。但在细颗粒度污点分析过程中，一条简单的二元操作指令就可能引起一个复杂的集合合并操作，这也导致严重的性能损耗。

2.3　基于roBDD的集合表达

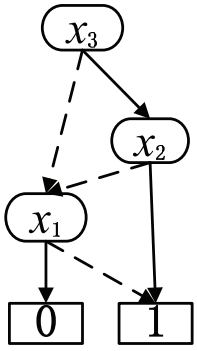


图 2.1: roBDD示例图

二元决策图（Binary Decision Diagram，简称BDD）是一种表达布尔函数的有向图结构 [44, 45]。任意一个布尔函数都可以建立相应的BDD结构，该BDD与布尔函数的真值表等价。具体而言，BDD是具有一个根节点的有向无环图(G, E)，并且:

- 含有两个出度为0的终止节点（terminal node），这两个节点上的标签分别为0和1；
- 除终止节点外，其他所有节点（相应于不同的布尔变量）的出度均为2；两条边叫做0-边和1-边，在BDD图中分别用虚线和实线代表。

例如，图2.1是逻辑表达式$(x_3 \bigwedge x_2) \bigvee \overline{x_1}$的BDD表达。1-边和0-边可以视为是将相应布尔变量赋值为True和False。从根节点到终止节点1的一条路径，就是使布尔函数成真时的一组赋值。

BDD的概念自1978年被提出后，就受到广泛关注。1986年，R. Bryant [44]提出了roBDD（Reduced Ordered Binary Decision Diagram）的概念，并给出了

从BDD到roBDD转换的高效算法。roBDD是对BDD结构的压缩表达，消除了BDD结构中同构子图和冗余节点。roBDD也是对布尔函数的一种规范表达，已经成为符号模型检验的重要基础 [45]。下文中为描述方便，BDD亦指为roBDD。

从更抽象的层次，BDD也可以看作是对集合关系的压缩描述。更为重要的是，基于BDD的集合表达直接支持各种集合操作。不失一般性，本文讨论整数集合。以4比特位可表达的整数为例，我们可以将4个比特位依次看作4个布尔变量：v_0 v_1 v_2 v_3，其中v_3对应域于最高比特位。整数即可抽象为对4个布尔变量的赋值。令整数i的二进制表达为$<b_3, b_2, b_1, b_0>$，一个整数集合S等价于一个布尔函数$f(v_3, v_2, v_1, v_0)$, 当且仅当：

1. $i \in S \Rightarrow f(b_3, b_2, b_1, b_0) = 1$

2. $i \notin S \Rightarrow f(b_3, b_2, b_1, b_0) = 0$

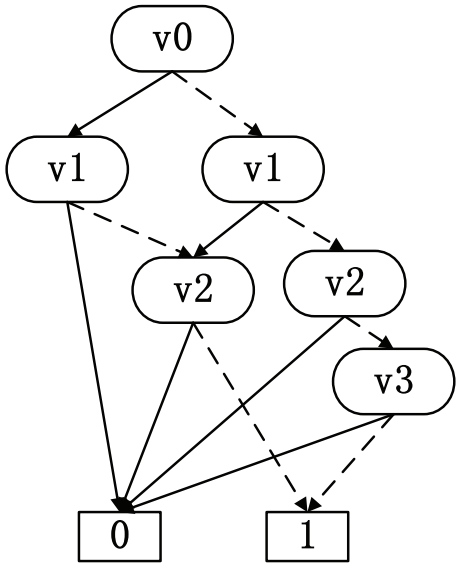


图 2.2: roBDD描述集合{0, 1, 2, 9, 10}

进一步，如果集合$S1$等价于布尔函数$f1$，集合$S2$等价于布尔函数$f2$，那么集合$S1 \cup S2$也等价于$f1 \bigvee f2$ [113]。类似地，集合$S1 \cap S2$也等价于$f1 \bigwedge f2$。由于BDD结构是对布尔函数的规范描述，我们可以采用BDD结构描述集合关系，BDD结构同样也支持各种集合操作。对于两个分别含有m和n个节点的BDD结构，

文献[113]中证明BDD的合并操作复杂度为$O(m*n)$。由于其消除了冗余信息,在存储空间上BDD一般要比布尔函数的真值表小很多。因此，BDD内部的节点数通常远远小于BDD所表达集合的势。图2.2给出了集合{0, 1, 2, 9, 10}的BDD表达。

2.4　基于roBDD的细颗粒度污点分析系统架构设计与实现

2.4.1　污点传播属性分析

本节分析细颗粒度污点分析过程中污点数据的传播特点，介绍如何利用这些特点，降低细颗粒度分析的内存需求，提高分析效率。

首先，污点数据的传播具有重复性。污点分析主要是追踪内存单元对污点数据的依赖关系，而很多内存单元/寄存器可能具有同样的污点属性。例如，mov是一个经常使用的x86指令，用于内存、寄存器之间的数据移动。这意味着执行mov指令后，其源操作数与目的操作数具有相同的污点属性。从高级语言层来看，很多高级语言特征例如函数参数的值传递、临时变量、对象克隆等都会引起污点数据的重复性。很多研究[180, 79, 107]都有类似的发现。显然，利用污点传播的重复性将有助于降低分析的内存需求。

其次，污点数据的传播具有重叠性。污点数据的交互操作引起了重叠性。对于二元操作，如果两个操作数都依赖于污点数据，那么操作结果将依赖于两个操作数污点属性的并集。例如，表2.1中，变量z依赖于集合{0,1,2,3,4,5,6,7}，而变量x污点属性为集合{0,1,2,3}，变量y污点属性为集合{4,5,6,7}。变量z与变量x、y的污点属性都有重叠。利用不同内存单元污点属性的重叠性也有助于降低污点分析的内存使用。

最后，污点数据的传播具有连续性。由于局部性原理，连续的污点数据通常被一起使用。例如，一个32比特的整数通常保存至连续4个字节。如果一个内存单元依赖该整数，意味着该内存单元的污点属性依赖于连续4个字节。其他原语类型例如字符串也具有类似的特性。

X. Zhang等人 [180]在对程序动态切片的研究中，发现基于roBDD的集合表达可以有效利用上述特性，降低集合的空间需求，能够提高切片效率。本文在细颗粒度污点分析中引入roBDD结构，采用roBDD描述污点属性起到了很好的效果。

2.4.2 系统设计与实现

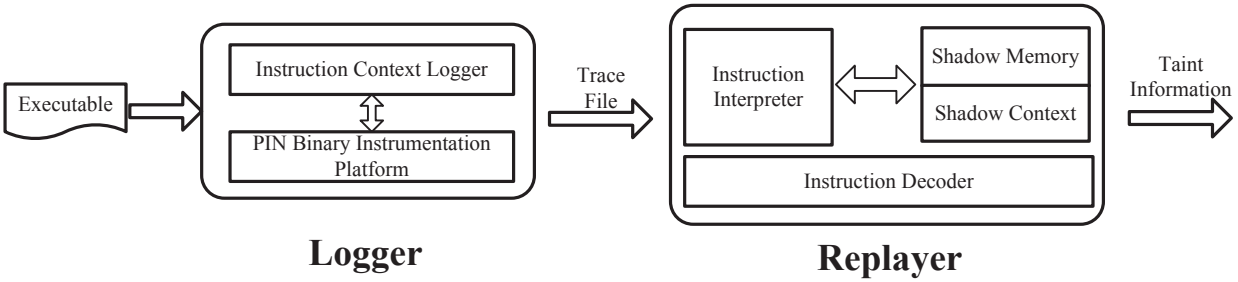


图 2.3: TaintReplayer系统架构图

本文实现了一个细颗粒度动态污点分析系统TaintReplayer，其系统结构如图2.3所示。TaintReplayer主要由两个模块构成：执行轨迹记录模块(Logger)和执行轨迹重放模块(Replayer)。TaintReplayer有以下特征：

1. TaintReplayer是一个离线动态污点分析系统。很多动态污点分析系统以在线模式运行，即在程序运行的同时进行污点追踪。然而，TaintReplayer首先记录目标程序的执行轨迹，进而对轨迹进行重放。在重放过程中，TaintReplayer进行细颗粒度污点分析。在线污点分析由于内存消耗大，对目标程序的性能影响巨大，经常导致目标程序无法正常运行，或过早退出。鉴于此，离线模式将目标程序的执行和污点分析过程分离。一方面，记录执行轨迹的工作对目标程序影响有限；另一方面，一旦执行轨迹被记录后，可以在高性能分析平台上，对该轨迹甚至轨迹片段进行灵活、重复分析，不再受操作系统环境或硬件平台的限制。

2. 在细颗粒度污点分析过程中，TaintReplayer采用roBDD表示污点属性，降低了分析过程的内粗消耗，提高了分析效率。

3. TaintReplayer不依赖与程序源代码，直接处理二进制程序，可运行在Windows、Linux等主流操作系统平台。

TaintReplayer的执行轨迹记录模块基于二进制植入平台Pin实现[110]。对于每条执行过的指令，TaintReplayer记录的信息包括指令静态信息（指令的

Algorithm 1: Simulate System Read

Input: buf is the buffer's starting address;

len is the number of bytes read from input file;

$offset$ is the file position;

Output: Memory map M;

for $i = 0$ *to* len **do**

    roBDD $b = offset + i$;

    M inserts $(buf + i) \mapsto b$;

return M

图 2.4: 文件读取系统调用模拟算法

地址和机器码）和指令运行时信息（访问的内存单元的地址/值，修改的寄存器等）。此外，对于引入污点数据的系统调用，TaintReplayer记录相应的参数。TaintReplayer使用PIN_AddSyscallEntryFunction 和PIN_AddSyscallExitFunction等Pin API劫持系统调用，获取系统调用参数和返回结果信息。例如，对于文件读取系统调用，TaintReplayer将记录操作发生时文件的偏移、接收缓冲区的起始地址，读入数据的长度等信息。

在重放过程中，执行轨迹重放模块负责对轨迹上每条指令进行语法和语义解析，实现细颗粒度污点分析。令影子内存为$M = \{m \mapsto r\}$，这里m代表内存地址或寄存器编号，r代表roBDD结构。进一步，令C为程序执行的上下文环境（context）。对于影子内存M和程序执行的上下文C，指令的污点传播函数$T(M, C)$描述指令的执行对污点数据传播的影响。

给定执行轨迹$< I, N >$，这里I表示指令序列，N表示轨迹上指令总数，令$T_i()$代表第i条指令的污点传播函数，M_0 和R_0分别为初始影子内存和上下文环境。污点分析即计算$(M_{i+1}, C_{i+1}) = T_i o T_{i-1} o \cdots o T_0(M_0, C_0)$的过程。

值得注意的是，对于最初引入污点数据的指令（例如系统调用），该指令的污点传播函数负责在影子内存M中初始化最初的污点映射关系。例如，图2.4描述了对文件读系统调用的污点传播函数。对于其他指令的污点传播函数，TaintReplayer根据指令的语义进行了处理。TaintReplayer当前主要追踪数据依赖关系，支持对绝大部分常用x86指令。

2.5 实验评估

我们实现了TaintReplayer的原型系统。其中，执行轨迹记录模块基于二进制植入平台Pin实现[110]。我们采用BuDDY [9]作为roBDD的实现。我们将BuDDY进行了移植，使之可以在Windows 操作系统上运行。我们在运行Win 7操作系统的平台上（硬件环境：Intel Core 2 Duo CPU 2.26GHz、3GB 内存）对TaintReplayer进行了测试，本节介绍测试结果。

2.5.1 实验设置

我们将TaintReplayer应用于Adobe Acrobat和Google Picasa，相应版本信息见表2.2。Adobe Acrobat是一款PDF格式阅读、编辑软件，Google Picasa是一款图片查看、修改软件，都具有巨大的用户量。测试中，我们使用Adobe Acrobat将一副PNG图片（大小为24.2 Kbytes）转换成PDF格式，并用TaintReplayer记录Adobe Acrobat的执行轨迹；我们使用Google Picasa浏览同一副PNG图片，也使用TaintReplayer记录Google Picasa的执行轨迹。表2.2中第三列给出了轨迹文件的大小，第四列给出了轨迹中x86指令总数。

表 2.2: TaintReplayer测试样本信息

程序名称	版本	轨迹大小(GB)	指令总数
Acrobat	9.4.1	2.69	190,617,272
Picasa	3.1.0	3.02	229,020,734

进一步，我们使用TaintReplayer对收集的执行轨迹进行细颗粒度污点分析。C++标准模板库STL（Standard Template Library）提供了`set`和`bitset`两种集合结构。我们分别实现了基于STL `set`和STL `bitset`的细颗粒度污点分析系统。实验中，我们发现基于STL `set`实现的污点分析系统运行太慢，无法在短时间内获得分析结果，因此我们并未收集STL `set`系统的数据。下文将从内存使用量和分析时间两个角度，对基于roBDD的污点分析系统与基于STL `bitset`的污点分析系统进行比较。

2.5.2　细颗粒污点分析测试结果

我们首先测量了基于roBDD的污点分析系统与基于STL `bitset`的污点分析系统在分析过程中的内存使用情况。图2.5显示了实验结果。其中，横轴描述输入文件中多少字节被标识为污点数据，纵轴表示污点分析过程中的内存使用量。随着被标识为污点数据的数目增多，基于STL `bitset`的污点分析系统消耗越来越多的内存；但是，基于roBDD的污点分析系统的内存使用并未随污点数据增长而增长。

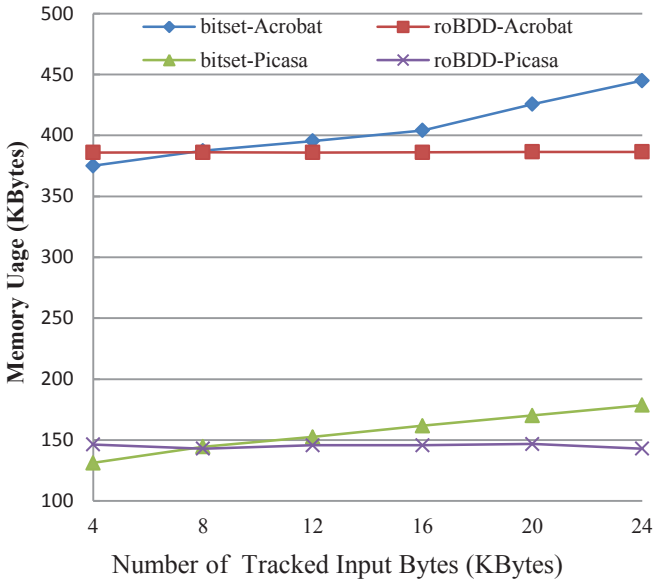


图 2.5: 污点分析系统内存消耗对比

为了比较两个系统的分析性能，我们测量了对千万条x86指令的污点分析的平均时间。实验结果如图2.6所示。其中，横轴描述输入文件中多少字节被标识为污点数据，纵轴表示千万条x86指令的平均重放时间。对执行轨迹简单重放（不进行污点分析）时，每千万条指令的平均重放时间是27秒。在对Acrobat和Picasa执行轨迹污点分析过程时中，基于roBDD的污点分析系统的千万条x86指令平均重放时间并不依赖于污点数据的多少；但是，基于STL `bitset`的污点分析系统的每千万条指令平均重放时间远远超过了基于roBDD的污点分析系统的分析时间，而且随污点数据的增多而迅速增加。

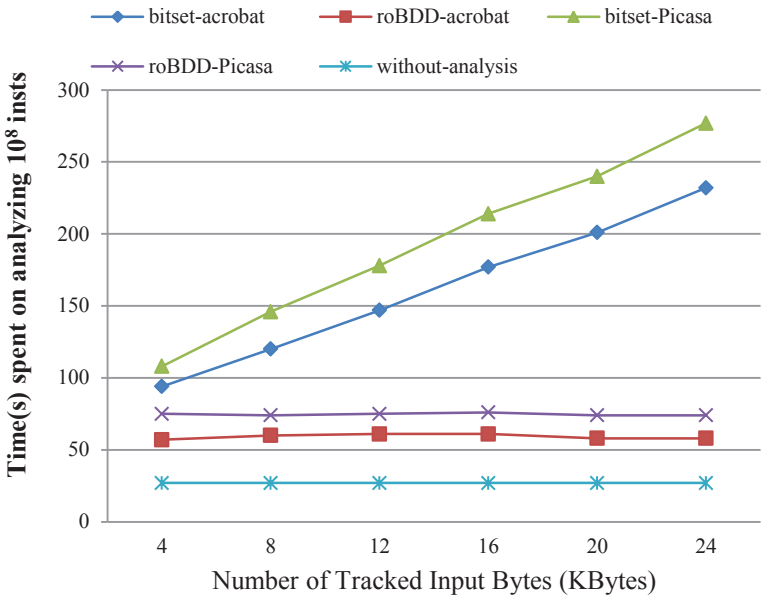


图 2.6: 污点分析系统性能对比

总体而言，实验结果表明基于roBDD的细颗粒度污点分析的内存需求低于基于STL `bitset`的污点分析系统，而且性能远远优于基于STL `bitset`的污点分析系统。此外，基于roBDD的细颗粒度污点分析的内存需求和分析性能，并不直接依赖于污点数据的多少。

2.6 本章总结

漏洞挖掘领域对细颗粒度污点分析有着越来越多的需求。本章介绍了一种基于roBDD的污点分析方法，并实现了原型系统TaintReplayer。TaintReplayer采用离线污点分析模式，在对程序执行轨迹的重放过程中，采用roBDD描述污点属性，进行细颗粒度污点分析。实验结果表明，roBDD结构降低了污点分析对内存的需求，显著提高了污点分析的性能，为进一步扩大细颗粒度污点分析的应用提供了支持。

第三章　面向二进制程序的混合符号执行技术

符号执行（symbolic execution）是一种代码执行空间遍历技术，在软件安全、恶意代码分析、程序调试等领域有着重要应用。本章介绍符号执行技术的原理、发展历史，分析符号执行技术的不足。结合二进制程序分析的需求，本章介绍一种面向二进制程序的混合符号执行技术的设计与实现。

3.1　符号执行原理及发展

符号执行用抽象符号代替程序变量，根据程序的语义，在每条路径上通过符号计算引擎对抽象符号做等语义操作，模拟程序执行。采用符号执行的好处在于能够发现变量之间代数运算关系，便于理解程序的内在逻辑，在漏洞挖掘时，有利于在复杂的数据依赖关系中发现数据之间本质的约束关系，而且符号执行精确记录了路径的约束条件，可以进一步用于判断路径可行性和路径约束的完备性。

表3.1展示了符号执行过程。其中，第一列表示一条程序路径，第二列表示对该路径污点分析的结果，第三列是对该路径的符号执行结果。本例中，在第一句代码执行后，符号执行引擎将变量x赋予符号值x_0，并根据后继各行代码的语义，更新程序变量与符号值的关系，收集路径上的约束条件。

将符号执行与污点分析过程对比，我们可以发现，污点分析跟踪了污点数据的传播，但并不能提供程序对污点数据处理的运算关系。而符号执行则记录了所有运算过程。例如，从指令层看，变量m与污点数据x存在数据依赖关系；但是从宏观角度讲，m在第5句代码处始终为常数值，实际上与污点变量x并不相关。污点分析并不能有效处理这种问题，仍然会将变量m视为污点数据。但通过符号计算，符号执行引擎可以计算出此时变量m为常数值2，与符号值x_0并不相关。

进一步，第二行代码中条件判断成立时，意味着谓词x_0>0成立；符号执行

表 3.1: 符号执行示例

程序路径	污点分析	符号执行
1. x = input();	x ↦ 1	x = x_0
		PC:
2. if(x>0){	x ↦ 1	x = x_0
		PC: x_0>0
3. y = x+1;	x ↦ 1 y ↦ 1	x = x_0 y = x_0+1
		PC: x_0>0
4. z = x-1;	... z ↦ 1	x = x_0, y = x_0+1 z = x_0-1
		PC: x_0>0
5. m = y-z;	... m ↦ 1	x = x_0, y = x_0+1, z = $x_0 - 1$ m = 2
		PC: x_0>0
6. if(y<1)	...	PC: x_0>0 && $x_0 + 1$ <1 路径不可行

将推演出第六行的代码实际是对谓词$x_0 + 1$ <1 的判断。而“x_0>0”成立时，谓词$x_0 + 1$ <1不可满足，这也意味着符号执行发现了一条不可行路径。

Boyer等[39]早在1975年就提出了符号执行的思想，并将符号执行应用于程序测试和调试领域。然而经历短暂的研究高潮后 [102, 123]，符号执行的研究就陷入低谷，在很长时间内并未取得显著进展，也未引起业界的关注。究其原因在于当时程序分析等领域追崇分析方法的完备性，而作为路径敏感分析的符号执行，无法对程序路径穷举遍历，自然不能保证分析的完备性。此外，符号执行也面临其他一些难题：

- 符号执行需要功能强大的符号计算系统支撑，对硬件性能要求较高，而符号计算系统在很长时间内一直没有取得突破性进步，制约了符号执行方法的发展；
- 符号执行很难处理循环或者递归调用；特别是边界条件中含有符号变量，符号执行无法判断循环、递归的次数；例如，在循环`while(x>0){...}` 中，如果`x`是符合变量，分析算法很难确定循环具体的执行次数；

- 实际程序实现过程中，代码会依赖大量系统API或第三方库函数，由于没有源代码，符号执行无法模拟执行这些调用，而这些系统API或第三方库函数很有可能对程序执行产生很大影响，改变程序原有数据约束关系；

- 符号执行可以对标量数据单元(如整型变量)精确建模，但是处理矢量数据单元(比如数组、字符串)面临巨大挑战。比如，在数组元素赋值时，如果以符号变量作为索引，符号执行无法判断具体对哪个数组元素的赋值; `strncpy(str2,str1,n)`将字符串`str1`前`n`个字符拷贝到字符串`str2`中，但是如果`n`是符号变量，分析算法不能确定具体拷贝多少字符，在引用`str2`字符串时无法确定内容。

随着约束求解器和符号执行引擎的计算能力不断提高，特别是在Prefix [46]在2000年前后成功将符号执行技术应用在漏洞挖掘领域后，符号执行重新引起了研究人员的关注。此时，研究人员不再追求挖掘技术的完备性或健全性，而是追求挖掘方法的效率和效果，提出了很多缓解符号执行固有缺陷的策略，实现了很多基于符号执行技术的漏洞检测工具 [172, 71]。

W. Bush等人在1998年提出了Prefix [46]，并在美国申请了若干专利。微软公司于1999年收购了Prefix，在Prefix基础上实现了Prefast。Prefix/Prefast已经成为微软内部标准源代码静态检验工具之一。Prefix/Prefast首先分析源代码，将其转换成抽象的语法树；然后对过程依照调用关系进行拓扑排序，再为每个过程生成相应的抽象模型，最后静态模拟执行路径并用约束求解的方法对约束集合进行检验。

Prefix/Prefast实现了过程间(inter-procedural)的符号执行；在模拟执行时，Prefix可以有效检测指针变量使用相关的漏洞，如未初始化指针引用、无效指针和空指针解引用(dereference)；路径结束时，Prefix检测路径上是否存在资源泄露等漏洞，如内存泄露，未释放的文件句柄等。为避免路径爆炸，Prefix在每个函数内部最多只检查50条路径；在所有路径结束后，Prefix合并所有路径的结果，建立函数的摘要，再次调用该函数时可直接应用已构建摘要，加速模拟。函数抽象模型的提取决定了Prefix/Prefast查错的能力和精度。

3.1.1 静态符号执行到混合符号执行

对程序执行环境的模拟是符号执行技术的基础。由于缺乏程序运行时信息，静态符号执行在模拟程序执行过程中遇到了很多困难。例如，对于全局变量，静态符号执行不能确定这些全局变量的取值，完全将这些全局变量视为符号值又会引起很多误报。

```
//rules for v = x op y
binary_assignment_rule(OP op, T v, T x, T y){
  if(neither x nor y holds a symbolic value)
    v = x op y;
  if(x holds a symbolic value && y holds a concrete value)
    s = x's symbolic value;
    v = sym_exp(op, s, y);
  else if(x holds holds a concrete value && y a symbolic value)
    s = y's symbolic value;
    v = sym_exp(op, x, s);
  else
    s1 = x's symbolic value;
    s2 = y's symbolic value;
    v = sym_exp(op, s1, s2);
}
```

图 3.1: EXE系统对赋值语句v = x op y的处理

斯坦福大学和微软的研究人员，分别独立提出了混合符号执行（conclic symbolic execution）的概念，将符号执行从一种程序静态分析技术转变为动态分析技术，实现了一系列代表工具，如EXE[50]、cute[143]、DART[90]。混合符号执行的核心思想是在程序真实运行过程中，运行时判断哪些代码需要经过符号执行，哪些代码可以直接运行。这样一来符号执行就充分利用了程序的运行时信息，提高了分析的精确性。

EXE[50]通过对程序源代码改写，将符号执行和约束求解能力整合到目标代码中。经过编译后，程序在执行时就可以把指定内存单元看做符号值，进行符号计算。图3.1是EXE系统对赋值语句v = x op y的处理规则。如果x和y都不是符号值，程序正常运行该语句；否则，程序进行符号计算。处理分支语句时，如

果分支语句的判断谓词中含有符号变量时，EXE 利用UNIX 系统fork()机制分别执行不同的分支，植入的约束求解器会判断哪个分支可行，摒弃不可行路径。2008年，Engler小组进一步提出了KLEE系统[49]，在LLVM编译器架构平台支持下，对EXE重新设计实现，并强化了对系统API的模拟，在多个经典程序中发现了漏洞，一举夺得OSDI2008的最佳论文。微软的研究人员则在函数摘要、非线性约束化简等方面做了大量研究，提升了混合符号执行对大型应用程序和复杂输入结构中漏洞的检测能力[89, 24, 88, 111, 156]。

3.1.2　面向二进制程序的混合符号执行技术

代码植入是混合符号执行技术的重要支撑。EXE[50]、DART[90]等工具是基于源代码植入平台CIL[124]实现。KLEE[49]是在编译器框架LLVM上实现。虽然混合符号执行技术取得重要进展，研究人员也意识到了基于源代码植入的混合符号执行存在诸多局限：

- 过于依赖程序源代码；以代码植入平台CIL、编译器框架LLVM为代表的源代码分析平台提供了丰富的程序分析技术支持。然而，在漏洞挖掘领域，目标软件的源代码通常不可获得。在只有可执行程序的情况下，基于源代码植入的混合符号执行无用武之地。

- 可扩展性受限；源代码分析平台大多面向于具体语言种类，导致目前基于源代码植入的混合符号执行工具也都面向特定的编程语言。而高级语言规则丰富（甚至存在不同的语言标准）、变化复杂，现有的源代码分析平台无法对各种语言标准完全支持，源代码分析平台的处理能力有限。以C语言为例，GNU GCC编译器和Microsoft VC编译器对C语言都有不同的扩展，C语言支持内嵌汇编指令，而C99、ANSI C等C语言标准也存在差异。这些都给C语言代码植入平台带来严重挑战。进一步，将现有混合符号执行工具扩展至支持其他编程语言的工作艰巨，受限于面向特定语言分析平台的实现，制约了混合符号执行技术的应用。

- 运行平台苛刻；很多大型程序的编译环境复杂，不仅对编译器的类型、版本有严格要求，对编译次序、环境变量等条件也非常敏感。但是，代码植入过

程很容易引入不兼容特征，导致植入后的程序因对编译环境敏感而无法正常编译、运行。

- 无法评估第三方模块、库函数的安全影响；软件开发过程中，对第三方模块、库函数的使用非常普遍，甚至是不可避免的。因此，即使是自主开发的软件，也不能保证掌握所有源代码。例如，C语言程序中经常使用C标准库函数，而C标准库都有不同的实现。除非分析过程中把这些C标准库的代码都纳入，否则混合符号执行工具就无法对二进制形式的C标准库植入，也就无法对C标准库函数实施符号执行。事实上，这些库函数可能对程序安全有着重要作用。缺乏对这些敏感库函数的支持，也影响了混合符号执行的准确性。

鉴于上述局限，很多研究人员提出直接在二进制程序层实现混合符号执行。直接在二进制层进行符号执行具有很多优势。首先，该技术不依赖程序源代码，适用范围更广，能够满足软件安全漏洞挖掘领域的需求。其次，面向二进制的混合符号执行并不针对特定编程语言，而是针对特定操作系统/指令系统。最后，面向二进制的混合符号执行忽略了目标程序高层实现上的区别，直接关注真实执行的指令序列，反应了很多安全本质信息。

但是，面向二进制的混合符号执行面临很多挑战。首先，与面向高级语言的程序分析相比，二进制分析缺乏大量信息。例如，从实现角度，程序变量是对内存的抽象。高级语言编程通过对一系列程序变量的操作来完成特定运算。然而，二进制程序中，没有程序变量的概念，处理的对象是内存单元和寄存器。没有程序变量的情况下，很多传统程序分析技术根本无法直接应用[27]。高级语言提供了明确的、丰富的变量类型信息，而这些类型信息在二进制程序层完全不能保留。其次，低级指令系统也十分复杂，给分析带来很多挑战。例如，x86指令系统具有数百条指令，而且很多指令的语义非常复杂。最后，程序的执行轨迹包含的指令数目庞大，这给符合执行的运行性能和空间消耗带来很多限制。

面向二进制的混合符号执行面临两个重要的技术选择：（1）在线符号执行vs离线符号执行和（2）面向底层指令系统的符号执行vs面向中间代码的符号执行。在线符号执行指在程序执行过程中，同时进行符号计算；离线符号执行指先

记录程序执行轨迹，在对轨迹重放过程中符号计算。面向底层指令系统的符号执行指根据底层指令的语义，符号化执行轨迹上每条指令；面向中间代码的符号执行指将轨迹上的底层指令，翻译到中间代码，对中间代码进行符号计算。

近些年涌现出的代表性系统有SAGE[91]，SmartFuzz[120]，BitScope[41]等。根据上述技术选择的差异，表3.2对这些系统进行了分类。其中，SAGE是在进程级虚拟机iDNA [34]基础上，实现了离线混合符号计算。SmartFuzz是在二进制植入Valgrind [125]上，实现了在线混合符号执行。BitScope是在基于系统虚拟机Qemu[33]的在线混合符号执行系统。下章将讨论各种选择的优劣，并介绍本文设计的离线混合符号执行系统SymReplayer。

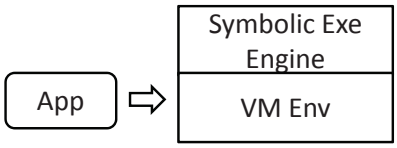
表 3.2: 面向二进制程序混合符号执行系统分类

系统名称	系统运行平台	二进制分析平台	混合符号执行模式	指令解释层
SAGE	Windows	iDNA	离线	二进制层（x86）
SmartFuzz	Linux	Valgrind	在线	中间代码层（VEX[125]）
BitScope	Linux	Qemu	在线	中间代码层（Vine[17]）
SymReplayer	Windows/Linux	PIN	离线	中间代码层（VEX[125]）

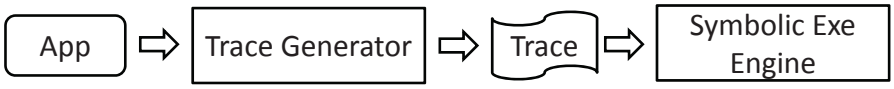
3.2　面向二进制的混合符号执行系统设计与实现

3.2.1　设计选择

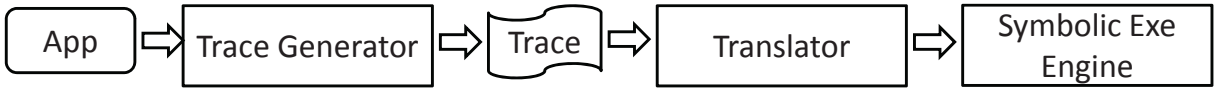
本文设计实现了面向二进制的混合符号执行系统SymReplayer。在符号执行模式的设计上，SymReplayer采用离线符号执行模式。图 3.2介绍了几种符号执行模式的区别。在线符号执行需要将符号执行引擎植入到目标程序的运行空间，在虚拟运行环境（VM Env）上，目标程序执行的同时对符号化数据进行符号演算。基于本文的研究，并结合文献[91, 120]的讨论，我们发现在线符号执行面临以下几个问题。首先，由于符号执行需要占用大量计算资源，在线符号执行会导致目标程序的性能严重下降。目标程序往往在运行过程中，会由于计算资源的不足而过早退出，限制了程序执行轨迹的深度，导致在线符号执行收集到的程序内部信息不足。



（1）在线混合符号执行



（2）离线混合符号执行



（3）离线+面向中间代码的混合符号执行

图 3.2: 混合符号执行模式

其次，在线符号执行大多基于二进制代码植入技术[110, 125]。符号执行的实现非常复杂，植入代码中发生的微小异常都可能导致目标程序运行失败。进一步，一旦目标程序采用了代码混淆 [61, 133, 108]等保护技术，都很容易导致混合符号执行无法正常运行。

最后，目标程序执行过程中的并发（concurrency）和不可确定性事件（nondeterminism）给在线混合符号执行带来重大挑战。在线混合符号执行很容易破坏多线程/多进程软件原有的同步关系，给目标程序带来资源冲突甚至错误。

采用离线符号执行模式可以有效缓解上述问题。离线符号执行指先记录程序执行轨迹，在对轨迹重放过程中符号计算。一方面，与在线符号执行相比，记录执行轨迹给目标程序执行带来的性能损失非常小；甚至在硬件虚拟化支持下，完全可以在硬件层实现执行轨迹的记录[73]。另一方面，记录执行轨迹的功能实现相对简单，给目标程序执行带来的影响有限，不会破坏多线程/多进程的同步关系。而且，执行轨迹重放的计算完全可以在高性能计算平台（可以与目标程序运行平台不一致）上实现，通过改善计算资源进而提高符号分析的性能。

在对指令解释层的选择上，SymReplayer采用面向中间代码的符号执行方式。

在对执行轨迹重放过程中，SymReplayer不是解释每条指令的语义，而是首先将指令翻译至一种中间代码，进而在中间代码层进行符号执行。面向中间代码的符号执行的主要优势是对不同的指令系统的兼容性。解释中间代码而非底层指令使得符号执行引擎具有很强的可扩展性。对于不同硬件平台（不同指令系统）上运行的程序，只需要重新实现执行轨迹记录，能够将新指令系统翻译至同一种中间代码，就可以对该平台上运行的程序分析。

3.2.2　系统设计与实现

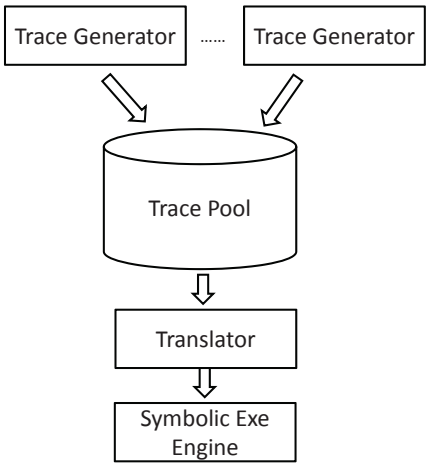


图 3.3: SymReplayer设计架构

基于上述考虑，SymReplayer采用离线且面向中间代码的混合符号执行模式。SymReplayer 的系统结构如图3.3所示。SymReplayer由三个模块构成，分别是执行轨迹记录模块（Trace Generator）、指令翻译模块（Translator）和混合符号执行模块（Symbolic Engine）。轨迹记录模块负责采集不同程序在不同环境下的执行轨迹，并将这些执行轨迹存储在数据库中（Trace Pool）；对于这些执行轨迹，指令翻译模块和混合符号执行模块协同工作，进行离线符号执行分析。具体而言，指令翻译模块负责将底层二进制指令翻译至中间代码，混合符号执行模块维护程序执行上下文环境，并进行符号计算。

作为前端，轨迹记录模块并不需要与后端（指令翻译模块和混合符号执行模块）运行在同一种硬件/软件环境下运行，约定执行轨迹格式后，轨迹记录模块在

不同的硬件/软件环境下也可以有灵活的实现方式。指令翻译模块负责将二进制指令转换到中间代码，因此需要设计一种灵活、易用的中间代码形式。下面详细介绍每个模块的设计。

执行轨迹记录模块

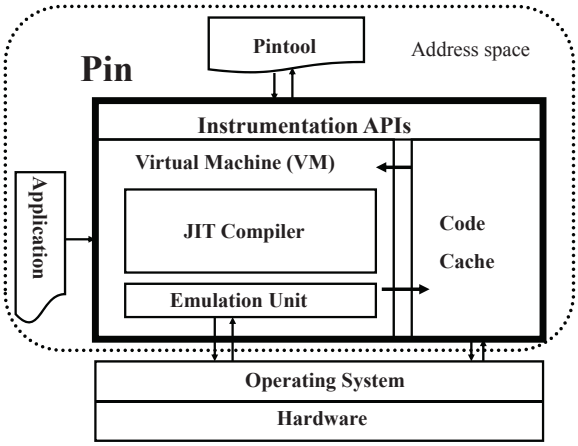


图 3.4: Pin体系结构图

执行轨迹记录模块（Trace Generator）主要负责记录目标程序执行过程的每条执行的运行时信息、系统调用参数和返回值等信息。考虑跨平台需求，执行轨迹记录模块基于二进制植入平台PIN[110]实现。

图3.4是二进制植入平台PIN的结构图[110]。PIN是Intel开发的二进制植入平台，支持对Windows、Linux等主流操作上应用程序分析。执行轨迹记录模块作为PIN的一个模块（图3.4中Pintool部分），结合指令层植入、系统调用劫持等技术，记录程序执行过程中每条指令的运行时信息。

为支持多线程程序，执行轨迹记录模块采用PIN 平台提供的`PIN_AddThreadStartFunction`和`PIN_AddThreadFiniFunction` 等API，监控线程的创建和结束，基于线程局部存储TLS（Thread Local Storage）机制为每个线程的执行单独记录执行轨迹，避免了植入代码的资源冲突。值得说明的是，在SymReplayer的设计结构中，完全可以兼容使用Valgrind[125]、Qemu[33]等二进制植入平台或虚拟机系统。

指令翻译模块

指令翻译模块（Translator）负责将执行轨迹上的二进制指令翻译至中间代码。在中间代码设计上，本文借鉴了精简指令集（RICS）的思想，基于Valgrind[125]中间代码翻译库实现了指令翻译模块，将执行轨迹上的二进制指令翻译到Valgrind中间代码VEX语言。指令翻译模块具有以下优点：

- VEX语言是一种体系独立的中间代码，结构简单清晰，便于各种程序分析。VEX代码的语法如图3.5所示。与x86指令系统（数百条不同语义的指令）相比，VEX中间代码中仅有十余种语句结构（IRStmt）和十余种表达式类型（IRExpr）。而且，VEX语言采用了平坦（flatten）模式，每条语句的表达式都是由原子表达式（atom）构成，即由立即数和临时寄存器构成。

- 中间代码翻译过程中实现了静态单一赋值SSA[70]形式，与原始二进制指令相比，VEX语言更贴近现有路径求解器，更容易生成被求解器接受的路径约束。VEX语言提供了丰富的类型信息。这些类型信息体现了一个变量的存储类型，对符号执行有巨大的帮助。

- 中间代码翻译过程中进行了一定的优化，避免了底层指令的负效应。例如，x86指令`xor eax, eax`会将寄存器`eax`赋值为0。经过翻译后，中间代码会直接将`eax`赋值为0。很多x86指令对状态寄存器`eflags` 都有影响，给指令模拟带来很多困难。但是在VEX语言中，已经将x86指令对`eflags`的影响，翻译为一系列布尔操作，有效解决了模拟`eflags`的难题。

- 除x86指令系统以外，指令翻译模块还支持AMD64、PowerPC等指令系统，能够将AMD64、PowerPC等指令系统翻译至同一种中间代码。Valgrind团队正在开发对ARM指令系统的支持。因此，只需要开发不同的执行轨迹记录器，SymReplayer就能够对ARM等平台上的应用程序分析。

图3.6给出了x86指令`0x8000 add esi, edi`对应的VEX语句序列。`IRSB`是VEX语言中的超级块（super block），`t0`、`t1`和`t2`是临时变量，`I32`指明相应变量为32bit整数类型。由于VEX实现了静态单一赋值，所有这些变量最多被赋值一

```
IRStmt :== NoOp
       | IMark of Addr64 * Int
       | AbiHint of IRExpr * Int
       | Put of Int * IRExpr
       | PutI of IRArray * IRExpr * Int * IRExpr
       | WrTmp of IRTemp * IRExpr
       | Store of IREndness * IRExpr * IRExpr
       | Exit of IRExpr * IRJumpKind * IRConst
       | Dirty of IRDirty
       | CAS of IRCAS
       | MBE
IRExpr :== Binder of Int
       | Get of Int * IRType
       | GetI of IRTemp * IRArray * IRExpr * Int
       | RdTmp of IRTemp
       | Qop of IROp * IRExpr * IRExpr * IRExpr * IRExpr
       | Triop of IROp * IRExpr * IRExpr * IRExpr * IRExpr
       | Binop of IROp * IRExpr * IRExpr
       | Unop of IROp * IRExpr
       | Load of IREndness * IRType * IRExpr
       | Const of IRConst
       | MuxOX of IRExpr * IRExpr * IRExpr
       | CCall of IRCallee * IRType * IRExprVec
IRExprVec  :== IRExpr | IRExprVec
IREndness  :== LittleEndian | BigEndian
IRArray    :== Int * IRType * Int
IRTemp     :== UInt
IRConst    :== Bool | UChar | UShort | UInt | ULong | Double
IRJumpKind :== Ijk_Boring | Ijk_Call | Ijk_Ret
       | Ijk_ClientReq | Ijk_Yield
       | Ijk_EmWarn | Ijk_NoDecode | Ijk_MapFail | Ijk_TInval
       | Ijk_NoRedir | Ijk_Trap | Ijk_Sys_syscall | Ijk_Sys_int32
       | Ijk_Sys_int128 | Ijk_Sys_sysenter
IRType :== Ity_INVALID | Ity_I1 | Ity_I8 | Ity_I16 | Ity_I32
       | Ity_I64 | Ity_I128 | Ity_F32 | Ity_F64 | Ity_V128
```

图 3.5: VEX中间代码BNF描述

次。VEX是一种体系独立的语言，具体硬件平台上的物理寄存器都对应于VEX语言中的一个编号。GET和PUT表达式分别读、写指定寄存器。例如，对于x86体系结构，GET:I32(24) 代表读取寄存器esi。虽然add指令会影响标志寄存器eflags，在VEX代码中已经将算术运算和标志寄存器修改的过程完全分离，使得后期符号执行过程中不必再专门模拟标志寄存器。

```
IRSB {
   t0:I32   t1:I32   t2:I32
   ------ IMark(0x8000, 2) ------
   t2 = GET:I32(24)
   t1 = GET:I32(28)
   t0 = Add32(t2,t1)
   PUT(32) = 0x3:I32
   PUT(36) = t2
   PUT(40) = t1
   PUT(44) = 0x0:I32
   PUT(24) = t0
   goto {Boring} 0x8002:I32
}
```

图 3.6: VEX IR for x86 instruction “0x8000 add esi, edi”

符号执行模块

符号执行模块（Symbolic Engine）在对执行轨迹重放过程中，一方面维护程序执行的真实上下文环境，另一方面维护程序执行的符号化上下文环境。图3.7给出了符号执行模块的算法。ctx代表真实执行上下文环境，记录真实地址/寄存器的真实值。$sctx$代表符号化上下文环境，记录真实地址/寄存器的符号值。本章中将内存地址/物理寄存器记为m，符号值记为s，$sctx$ 就是m到s的映射，记为$\{m \mapsto s\}$。

符号执行模块依次取出执行轨迹上的每条指令i，并根据轨迹文件中记录的i的上下文更新真实执行环境ctx。进一步调用指令翻译模块（对应于图3.7中translateBinaryInsttoIR 函数），符号执行模块将指令i翻译至VEX中间

Algorithm 2: Symbolic Execution on an Execution Trace

```
Input: Trace is a recorded execution trace
begin
    BinaryInst i;
    Context ctx = InitContext();
    ShadowContext sctx = InitShadowContext();
    foreach i in Trace do
        ctx = updateContext(i);
        IRSB block = translateBinaryInsttoIR(i);
        IRStmt stmt;
        foreach stmt in block do
            sctx = symbolicEvaluate(stmt, ctx);
        end
    end
```

图 3.7: 面向执行轨迹的混合符号执行算法

代码，在中间代码上进行符号计算。

在重放初期，由于执行环境中尚未引入符号变量，可以不用进行指令翻译和符号执行。在重放文件读取、网络报文接收等系统调用引入初始的符号变量后，在后继重放过程中，就需要检查每条指令是否与这些符号变量相关。

SymReplayer在比特级别模拟符号变量。具体而言，假设执行轨迹上从特定文件中读取了N字节数据到内存buf、...、buf+N-1，符号执行模块会创建N个8位比特矢量(bit-vector)结构用于描述N个符号值（即每个符号值是一个8位比特矢量），并在$sctx$中记录内存地址buf、buf+N-1与N个比特矢量的映射关系。很多约束求解器例如Z3[177]、STP[85]等都支持对比特矢量约束的求解。符号执行模块根据图3.8中描述的算法处理后继内存读取操作。对于N字节的内存读取操作，符号执行模块会在$sctx$和ctx中查询相应的内存地址，如果N个地址单元都未存储符号值，符号执行模块返回这N个字节对应的真实值；否则，符号执行模块从$sctx$和ctx中取出相应符号值/真实值，连接成一个长$8*N$的比特矢量并返回该值。

符号执行模块根据图3.9所示规则处理内存写操作。对于N字节的内存写操

```
//rules for memory load operation (little-endian)
//N represents the number of bytes to load
Expr memory_load_rule(UINT addr, UINT N){
  assert(N>=0);
  if(N==1){
    if(sctx[addr]==NULL)
      return ctx[addr];
    else
      return sctx[addr]
  }
  else{
    if(sctx[addr]==NULL)
      return memory_load_rule(addr+1, N-1)@ctx[addr];
    else
      return memory_load_rule(addr+1, N-1)@sctx[addr];
  }
}
```

图 3.8: 内存读取处理算法

```
//rules for memory write operation (little-endian)
//N represents the number of bytes to write
//e represents the value to write
void memory_write_rule(UINT addr, UINT N, Expr e){
  assert(N>=0);
  for(i=0; i<N; i++){
    //extract the value of the ith byte in e;
    Expr e' = e[i*8:(i+1)*8-1];
    if(e'.isConcrete())
      sctx.delete(addr+i);
    else
      sctx[addr+i]=e';
  }
}
```

图 3.9: 内存写处理算法

作，假设写入值的符号表达式为e，符号执行模块会依次取出e中各个字节对应的表达式，更新$sctx$中buf、...、buf+N-1等内存单元的映射关系。需要说明

的是，SymReplayer 并未处理符号地址推理，即内存读写地址表达式为符号值的情况。一旦内存读写操作的目的地址是符号表达式时，SymReplayer根据执行的真实上下文环境，采用真实地址代替该符号地址。上述策略与SAGE [91]、SmartFuzz[120]、LESE[140]等二进制混合符号执行工具类似。在后继章节中，本文会讨论符号地址推理所面临的问题，并根据具体问题的需求提出并实现了部分符号地址推理方法。

对于算术运算、逻辑运算等操作，符号执行模块依据操作语义生成新的符号表达式，更新符号化上下文环境$sctx$。当遇到条件跳转语句时，符号执行模块通过比较条件跳转指令的目标地址和执行轨迹中下一条指令的地址，判断当前条件跳转语句是否发生，进而生成相应谓词判断、收集路径约束条件。

3.3 讨论与小结

混合符号执行技术已经成为重要的代码执行空间遍历技术。与传统静态符号执行技术不同，混合符号执行技术充分利用了程序运行时信息，提高了符号执行的准确性，促进了符号执行技术的发展。另一方面，随着业界对二进制程序分析需求的不断增长，面向二进制程序的混合符号执行技术变得越来越重要。

本章介绍了符号执行技术的早期发展和不足，讨论了静态符号执行技术到混合符号执行的转变，分析了源代码级混合符号执行与二进制代码级混合符号执行所面临的问题，并对典型的面向二进制程序的混合符号执行系统进行了分类研究，深入研究了几种符号执行技术实现上的差异。

本文提出了一种面向二进制程序的混合符号执行模式。该模式综合利用了离线分析和面向中间代码的符号执行，模块化程度高，可扩展性强，支持对不同平台上运行的二进制程序进行分析。本文基于二进制植入平台PIN实现了执行轨迹记录功能，能够对Windows/Linux等主流操作系统上应用程序记录执行轨迹；进一步，本文实现了二进制指令翻译模块，能够将x86等二进制指令翻译至Valgrind中间语言VEX；最后，本文实现了面向VEX中间代码的混合符号执行模块，在对执行轨迹重放过程中，将二进制指令翻译到中间语言后，进行混合符号执行。本章实现的混合符号执行原型系统SymReplayer为后文的校验和感知的模糊测试技术及智能测试例生成技术奠定了基础。

第四章　校验和感知的模糊测试技术

4.1　引言

模糊测试（fuzzing/fuzz testing）技术是软件安全漏洞动态挖掘的典型方式。然而，基于校验和机制的数据完整性检查成为传统模糊测试系统的重要障碍。

本章围绕如何绕过校验和检验机制展开研究，首次提出了一种校验和感知的模糊测试方法，为运用传统模糊测试发现深藏的软件漏洞扫除了障碍，扩展了模糊测试技术的应用范围。

本章内容安排如下：首先介绍模糊测试技术的发展，接下来讨论校验和机制给现有模糊测试系统导致的困难；描述本章的研究目标后，概述校验和感知的模糊测试的核心思想；然后，详细阐述校验和感知的模糊测试技术的原理和实现，并给出实验结果；最后讨论校验和感知的模糊测试技术的局限和其他应用场景，并对本章进行总结。

4.2　模糊测试技术的发展

Wisconsin大学B. P. Miller教授是模糊测试技术的开创者，为模糊测试技术的发展做出了重要贡献。1988-1990年期间，B. P. Miller[115]提出了采用随机、无结构化数据测试应用程序的思想，称之为模糊测试（Fuzz）。虽然该思想非常简单，但对当时7种Unix系统的测试过程中，该方法使25-33%的Unix应用程序崩溃。在1995、2000、2006年，B. P. Miller分别对X-window[116]、Windows[82]、MacOS[114]等系统进行模糊测试，都发现了大量安全漏洞。

由于缺乏形式化模型和理论基础，模糊测试提出的初期饱受来自软件工程、软件测试等领域研究人员的抨击。在B. P. Miller 为《Open Source Fuzzing Tools》[136]一书所做的序言中提及，早期模糊测试被认为是“stone axes and bear skins”式的方法。

然而，模糊测试的思想深受黑客社区的青睐，短时间内涌现出大量针对特定通信协议（例如针对FTP[84]、VoIP[20]）或特定目标（例如针对浏览器[112]、ActiveX[74]、COM对象[181]）的模糊测试工具。模糊测试在黑客社区得到了成功的应用。在模糊测试工具的帮助下，黑客团队很容易对目标软件进行大规模测试，挖掘潜在的安全漏洞。例如，在黑客社区的推动下，“Month of Browser Bugs”[118]和“Month of Kernel Bugs”[119]等活动在连续一个月的时间内，每天都公布浏览器和操作系统内核的零日安全漏洞，给软件安全分析领域带来巨大冲击。

模糊测试在黑客社区的不断推广给软件厂商带来很大压力。为增强自身软件产品的安全性，微软、Adobe、思科等IT企业先后在软件开发周期中引入模糊测试环节。例如，微软推出的软件安全开发生命周期SDL（Security Development Lifecycle）理念中，模糊测试是安全测试不可或缺的步骤[117]。

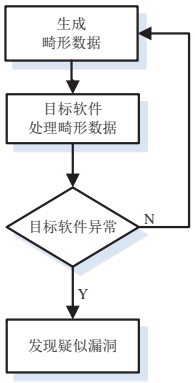


图 4.1: 传统模糊测试流程

图4.1给出了传统模糊测试的流程。模糊测试产生大量随机、无结构的畸形数据后，观测目标软件接收这些畸形数据后的表现。一旦目标程序发生崩溃等行为，意味着模糊测试发现了一个潜在的安全漏洞。

很多研究将模糊测试技术的思想推广，不再局限于仅在用户接口处输入畸形数据，而是在目标程序与外界所有可能接口处引入畸形数据。对于应用程序而言，大多存在与用户、系统内核、其他库函数等的接口（如图4.2所示）。扩展后的模糊测试可以在这些接口处注入随机、无结构数据，观测目标软件接收这些畸形数

据后的表现。在软件可靠性评估领域，模糊测试技术也有着广泛的应用。例如，可以在目标程序调用内存分配函数时，强制返回内存失败，观察目标程序是否正确处理了内存分配失败的情况，以考察目标程序在极端运行环境下的可靠性。

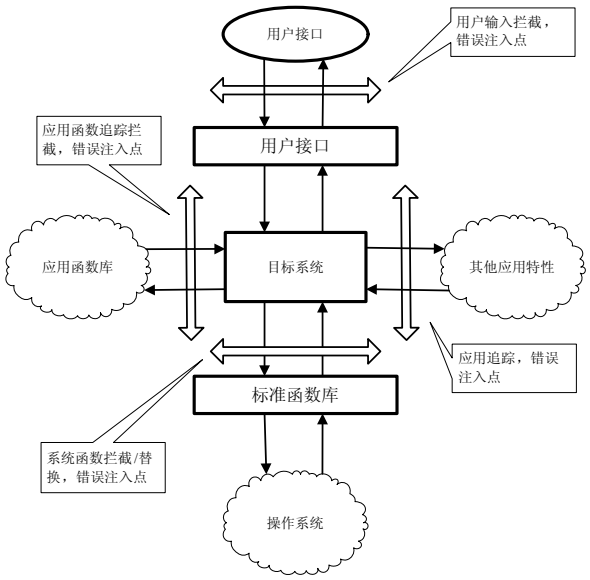


图 4.2: 扩展模糊测试模型

模糊测试技术的有效性完全依赖于畸形数据的生成方式。传统模糊测试系统的畸形数据生成方式可以分成两类[128]: 基于变异的生成方法和基于规范的生成方法。基于变异的生成方法（Mutation-based）的基本思想是通过对正常数据的修改而生产畸形数据；基于规范的生成方法（Specification-based）是在数据生成规则基础上，自动构造畸形数据。

基于规范的畸形数据生成方法能够充分利用测试人员对文件/协议格式掌握的先验知识，生成的畸形数据在满足格式约束的同时，能够对目标程序深入测试。典型的基于规范生成系统有SPIKE[147]，PEACH[132]，Protos[138]等。测试人员可以利用这些工具提供的API，描述一个文件/报文的组成结构，指定结构中哪些部分需要填充正常数据、哪些部分可以填充畸形数据，还可以描述结构中各部分的依赖关系。由此生成的畸形数据，可以通过目标程序对数据结构的有效性检查，测试目标程序处理畸形数据时是否产生异常。

然而，如何获取畸形数据生成规范成为这类系统的瓶颈。对于复杂的文件/协议格式，人工描述格式的工作量巨大，而且很容易出错。测试例生成规范中的任何错误，都可能导致所有生产的畸形数据都无法通过数据格式的有效性检查。另一方面，对于未公开协议格式，测试人员也不了解数据格式的详细信息，根本无法制定畸形数据生成规则，也就不能使用基于规范生成畸形数据的模糊测试系统。

近些年来，协议格式逆向工程技术（protocol reverse engineering）取得了显著的发展，实现了大量原型系统 [48, 170, 106, 69, 62]，甚至已有研究尝试将协议逆向技术和模糊测试技术相结合[62]。然而，对于复杂的协议格式，通过协议逆向技术生成畸形数据构造规范依然是一个难题。

基于变异的生成方法通过对正常数据的随机修改/破坏而生成畸形数据，其主要优势在于不依赖于具体格式而且易于实现。典型的系统有FileFuzz[80]、Zzuf[182]、MiniFuzz[117]等。为了进一步改进畸形数据的测试效果，Liu[109]等人提出基于遗传算法的模糊测试技术。近些年来，随着混合符号执行技术的发展，出现了大量基于符号执行技术生成测试数据的系统，例如SAGE[91]、SmartFuzz[120]等。这些系统把正常测试数据当做符号值，收集目标程序处理正常测试数据时的路径约束，通过约束求解生成新的测试数据，驱动程序执行不同路径，达到遍历程序执行空间的目的。本文在第五章中对这类方法进行讨论并展开研究。

4.3 校验和机制引起的问题

随着模糊测试在黑客社区和产业界的成功应用，传统模糊测试技术的不足也慢慢凸显出来。在应用过程中，人们发现很多情况下，模糊测试技术并不奏效，无法对目标程序进行有效测试。究其原因，研究人员发现基于校验和（checksum）的数据完整性检验机制成为模糊测试的主要障碍。

校验和是一种常见的数据完整性检验机制，已经广泛应用于各种文件格式和网络协议。下面简单介绍一下校验和机制的基本思想。数据内容D的校验和C本质上是D的一种散列值$H(D)$，这里$H()$代表散列函数。程序重新计算数据

内容D的校验和$H(D)$后，与输入数据中附带的校验和C进行比较，以此检验数据的完整性。一旦新计算出的散列值$H(D)$与输入的校验和C不一致，就意味着数据已经被破坏；通常程序不会再继续处理这些数据。

学术界对散列函数的研究已经非常成熟。常见的用于校验和计算的散列算法有MD5，SHA1，CRC系列和Alder等。一个好的校验和算法$H()$一般要具有单向性、抗弱碰撞性和抗强碰撞性[149]，即：

- 单向性：对于给定校验和值C，找到满足$H(D) = C$的数据内容D在计算上是不可行的；
- 抗弱碰撞性：对于给定数据内容x，找到满足$H(x) = H(y)$的数据内容y在计算上是不可行的；
- 抗强碰撞性：找到满足$H(x) = H(y)$的任意数据内容x和y在计算上是不可行的。

对于复杂的未公开格式，模糊测试技术大多采用基于变异的测试数据生成方法。然而，一旦这些格式中采用了校验和机制检测数据的完整性，那么对原始正常数据的任何微小修改都会破坏完整性，导致生成的畸形数据无法通过程序的校验和检查。现有协议逆向工程技术 [48, 170, 106, 69, 62] 也无法自动识别程序使用的校验和算法。如何自动生成满足完整性约束的畸形数据是一个公认的难题。

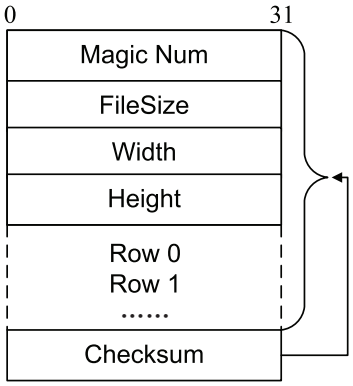


图 4.3: 含有校验和的文件格式示例

下面结合一个实例进一步解释校验和机制给模糊测试带来的挑战。图4.3展示一种示例图像文件格式。在该格式中，MagicNum域用于标记文件格式信息，FileSize域用于指明文件的有效长度，Width和Height域分别用于表示图像的宽度和高度。随后，按照行优先（row-major order）的次序保存图像的像素数据。值得注意的是，格式的最后四个字节，即Checksum域，用于保存该文件的校验和。

图4.4给出解析上述文件格式的示例代码。第3行代码读取图像文件的长度后，第4行通过调用checksum()函数重新计算了图像文件的校验和recomputed_chksum。接下来，第5行代码读取了保存在图像文件中的校验和chksum_in_file。然后，该函数比较重新计算的校验和与文件中已保存的校验和。如果二者不一致，该函数生成错误信息后退出；反之，该函数继续处理该图像文件。读取图像文件的高度和宽度后，函数在第12行计算需要为图像文件分配多少内存空间。成功分配内存空间后，按行读取像素数据至内存中。

```
1  void decode_image(FILE* fd){
2   ...
3   int length            = get_length(fd);
4   int recomputed_chksum = checksum(fd, length);
5   int chksum_in_file    = get_checksum(fd);
6   if(chksum_in_file != recomputed_chksum)
7     error();
8   int Width    = get_width(fd);
9   int Height   = get_height(fd);
10  if(Width==0 || Height==0)
11    error();
12  int size = Width*Height*sizeof(int); //integer overflow
13  int* p   = malloc(size);
14  ...
15  for(i=0; i<Height;i++){// read ith row to p
16    read_row(p + Width*i, i, fd); //heap overflow
```

图 4.4: 含有漏洞的代码示例

值得说明的是，这段代码存在一个整数溢出漏洞（在32位系统上）。由于没有严格检测图像的高度和宽度，在第12行计算内存空间需求时，过大的宽度和高度会导致Width*Height*sizeof(int)发生溢出。例如，Width=0x80000001、

`Height=1`时，`Width*Height*sizeof(int)`的计算结果为4。因此，第13行只会分配很少的内存，读取像素数据至内存的过程中就会发生堆溢出。

然而，在图像格式未知或具体校验和算法不公开时，传统模糊测试工具几乎无法生成通过第6行检查的畸形图像文件。黑客社区[75][129]极力改进现有模糊测试工具，但是依然无法有效对抗校验和检查机制。

上文已经介绍了一个好的校验和算法具有抗弱碰撞性和抗强碰撞性。因此，针对正常图片随机修改而生成的畸形图片就很难通过完整性检查。如果被随机修改的部分是数据部分，修改后的图片通过完整性检查的概率就相当于找到一个满足$H(D') = H(D)$ 的数据D'的概率，这里D代表原始正常数据部分；此外，只有畸形数据内容D'引起程序崩溃才意味着发现了一个潜在的安全漏洞。虽然近些年来国际上对散列函数碰撞的研究取得了显著进展，特别是我国学者王小云等[160]在该方向上做出突出贡献，但现有散列函数碰撞的研究并不能直接帮助模糊测试克服校验和检验问题。

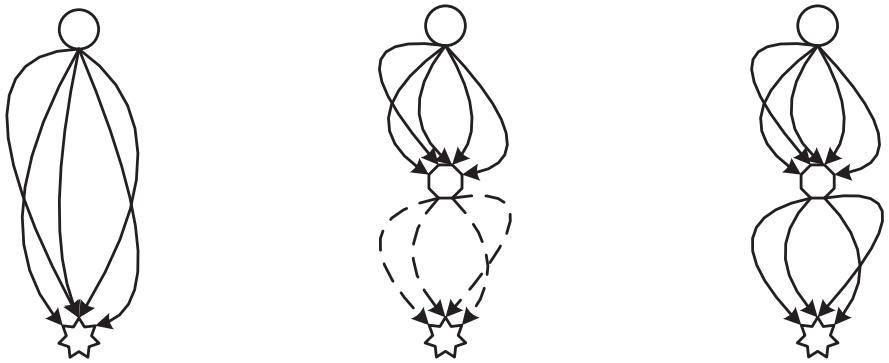
如果被随机修改的部分是输入校验和值，由于$H(D)$返回的是唯一的结果，生成的畸形图片就根本不可能通过完整性检查。即便是通过符号执行技术，收集到执行路径上对输入数据的约束条件，在D和C都是未知变量的情况下，也不可能完全求解$H(D) = C$这一类型的约束条件[145, 42]。

4.4 研究目标

基于校验和机制的数据完整性检查严重阻碍了现有模糊测试技术的应用。图5.2(a) 给出无校验和机制时模糊测试的效果图。大量的畸形数据能够充分遍历程序执行空间，进而发现潜在的漏洞。图中实线箭头表示畸形数据能够有效测试的程序执行空间。

图5.2(b)形象化展示了校验和机制给模糊测试技术造成障碍。图中八边形图案代表校验和检查点。图中虚线箭头表示畸形数据无法测试的程序执行空间。大量的畸形数据能驱使程序运行至校验和检查点，但由于这些畸形数据很难通过校验和检测，这些数据仅能对一小部分代码执行空间测试，无法发现校验和检测点以后的安全漏洞。

图5.2(c)形象化显示了本文的研究目标：克服基于校验和机制的完整性检测，使畸形测试数据能够穿透校验和检查，对目标程序深度测试。具体而言，该研究目标涉及两个主要问题。第一，本文的研究应适用于未公开协议。即在协议格式未知的情况下，依然能够有效地感知并克服校验和检查。第二，由于校验和算法多种多样，本文的研究不应仅针对某个具体的校验和算法，而应该具有普适性，能够不依赖于校验和算法的实现，拓宽模糊测试的应用范围。



a. 无校验和机制 b. 存在校验和检查 c. 本文的研究目标

图 4.5: 校验和感知的模糊测试技术示意

4.5 校验和感知的模糊测试方法概述

4.5.1 基本思想

校验和机制导致传统模糊测试失效主要原因在于（1）畸形数据无法通过校验和检查；（2）把所有输入都当做符号执时，符号执行和约束求解技术不能有效求解复杂路径约束。下面围绕这两个问题，介绍校验和感知的模糊测试技术基本思想。

针对畸形数据无法通过校验和检查的问题，本文提出的解决办法的核心思想是禁用程序中对输入数据进行完整性检测，强制目标程序接收这些畸形数据。详细地说，如果我们能够定位程序中的校验和检查点，我们就可以在校验和检查点修改程序，使程序接收所有输入。这种强制程序执行的方法在软件破解和恶意代码分析领域有着广泛应用[75, 121, 168]。

```
if(H(D)!= T(c))
   exit();
continue();
```

a. 原始目标代码

```
if(H(D)!= T(c))
   goto L1; // directed jump
   exit();
L1:
continue();
```

b. 修改后代码

图 4.6: 绕过校验和检查的基本思想

例如，图4.6（a）展示了程序中进行校验和检查的实现。如果H(D)和T(C)不一致，程序就直接退出，拒绝接受该数据；否则，程序继续处理输入数据。这里$H(D)$代表重新计算的校验和值，而$T(C)$代表程序对输入数据中校验和域保存的原始数据内容的处理结果。假设我们已经定位图4.6（a）中条件判断语句是用于检查输入数据完整性，我们就可以通过代码改写机制，植入直接跳转语句，类似于图4.6（b）的做法，使程序中校验和检查机制失效。因此，修改后的程序会“认为”所有数据都通过了校验和检查，继续处理这些数据。我们进一步采用传统模糊测试技术，产生海量畸形数据，对修改后的程序进行测试，挖掘隐藏在校验和检查之后的安全漏洞。

然而，上述方法存在一个重要问题：即使传统模糊测试工具生成了一个导致修改后程序崩溃的畸形数据，由于校验和检查失败，这个畸形数据也无法导致原始程序崩溃。为了使原始程序接收这个畸形数据，我们需要修复畸形数据的校验和，使得这个畸形数据满足校验和约束。

本文基于混合符号执行技术修复畸形数据中的校验和。上文已经强调，所有输入都当做符号值时，符号执行和约束求解技术不能有效求解$H(D) = T(C)$类型约束。我们进一步分析这个约束。这个约束条件之所以难于求解，是因为$H(D)$部分过于复杂。在传统模糊测试过程中，我们已经生成了导致修改后程序崩溃的畸形数据。为了使这个数据通过原始程序的校验和检查，我们只需要调整这个畸形数据中校验和部分即可。换而言之，在符号执行过程中，我们只把C部分当做符号值，把D部分用畸形数据中的真实值代替，因此$H(D) = T(C)$会大大简化，完

全可以被现有求解器处理。

4.5.2 核心问题

上节介绍了校验和感知的模糊测试方法的基本思想。为了实现该方法，我们必须解决以下问题。

1. 如何自动定位程序中校验和检测点。只有准确定位程序中的校验和检测点，才能修改程序以禁止程序检测校验和，从而接受各种畸形数据。如何自动定位程序中校验和检测点是方法能否实施的关键。但是，对于未公开复杂协议，特别是在目标程序源代码不可访问的情况下，校验和检测点的定位异常困难。现有研究中尚未见到能够有效、精确定位校验和检测点的方法。

2. 如何自动修复畸形数据中校验和域。如果一个畸形数据能够触发一个隐藏在校验和检查之后的安全漏洞，导致修改后的程序崩溃，除非能够修复畸形数据的校验和域，否则这个畸形数据还是无法导致原始程序崩溃。在协议格式不明确、校验和算法未知、程序源代码不公开的情况下，如何为畸形样本自动生成正确的校验和域，是本文需要攻克的难题。对于简单校验和算法（例如逐字节异或/累加），在校验和修复问题上，有些研究人员做了初步尝试。例如文献[126]提出了基于符号执行的修复方案。然而，即使这类简单校验和算法已经给文献[126]的约束求解带来巨大的挑战；复杂的校验和算法则成为一个遗留的难题。

此外，如何提高畸形数据触发安全漏洞的概率也是本文的研究重点。本文在第五章集中讨论畸形数据的生成方法。

4.5.3 系统概览

本文实现了校验和感知的模糊测试原型系统TaintScope，其工作流程如图4.7所示。TaintScope工作流程主要由三部分组成：校验和检测点定位、模糊测试和校验和修复。

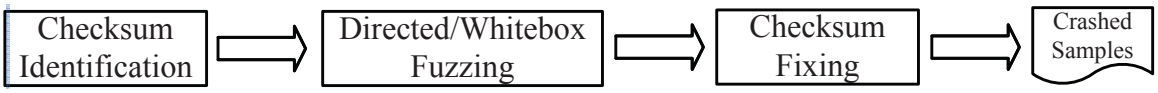


图 4.7: 校验和感知的模糊测试流程图

校验和检测点定位负责识别二进制程序中用于检测校验和的分支判断语句，修改原始程序以禁用校验和验证。此外，校验和检测点定位过程中，TaintScope也进行部分协议格式逆向分析，识别正常样本中校验和域的位置。

模糊测试阶段将结合第五章测试例生成方法，对修改后的程序进行测试。对于能够使修改后程序崩溃的畸形测试例，TaintScope在自动修复这些测试例的校验和域，使之通过原始程序的校验和检查。最终，TaintScope输出能够使原始程序崩溃的畸形测试用例。

4.6 TaintScope系统设计与实现

4.6.1 校验和检测特征分析

不失一般性，假设校验和检测形式为$H(D) == T(C)$，其中:

- D为数据内容；
- C为格式中校验和域的原始数据；
- $H()$代表校验和算法；
- $T()$代表程序执行过程中对校验和域的处理过程。

$H(D)$即重新计算的数据校验和，而$T(C)$就是输入的校验和值。二者不一致表明数据的完整性遭到破坏。

这里进一步解释一下$T()$的含义。例如，如果校验和域中按照big-endian 模式存储校验和值，而程序执行中按照little-endian模式处理校验和值时，程序在读入校验和域中数据后，需要做big-endian到little-endian的转换。$T()$就代表这种转换。

大量真实程序对已有协议的校验和检查方式都符合上述模式。从程序语言角度讲，这种检测模式通常由条件判断结构实现。进一步，这类条件判断语句具有以下特征：

- 高污点依赖度。**污点依赖度**指的是，细颗粒度污点分析过程中一个内存单元/寄存器被赋予的污点标签的个数。进一步，污点依赖度如果超过了预定的阈值，就被称为高污点依赖度；反之称为低污点依赖度。如果把所有输入数据看做污点数据，由于$H(D)$是对数据D散列运算的结果，那么$H(D)$通常依赖于数据内容D中的所有字节；类似地，$T(C)$的污点依赖度决定于校验和域的长度。因为校验和检测语句需要比较$H(D)$和$T(C)$，意味着该条件判断语句的操作数通常是一个高污点依赖度的数值。另一方面，校验和一般用于保护一长段数据，而校验和域本身仅需要几个字节（取决于具体校验和算法）。例如，CRC32 算法生成的校验和4字节，MD5算法生成的校验和16字节。因此，在检查校验和的过程中，通常会存在低污点依赖度值和高污点依赖度值相比较或运算的现象。

- 行为差异性。由于基于随机修改生成的畸形数据很难通过校验和检测，而正常数据都可以通过校验和检测，那么程序中负责检查校验和的分支语句（简称为校验和检查点）的行为就像一个分类器（如图4.8所示）：在处理正常数据样本和畸形数据样本时的行为完全不同。换言之，校验和检查点处理正常样本时，条件判断一直为True/False; 校验和检查点处理正常样本时，条件判断一直为False/True。

- 无负效应性。基于校验和机制的完整性检查仅用于检测数据完整性，并不会计算生成程序后继执行过程中不可或缺的数据。换个角度讲，校验和域保存的数据校验和，一旦数据通过了完整性检测，也就不会再继续影响程序的控制流。

4.6.2 校验和检测点自动定位

基于第4.6.1节中对校验和检查点行为特征的分析，本文提出一种面向二进制程序的校验和检测点自动定位方法。该方法综合利用细颗粒度污点分析技术

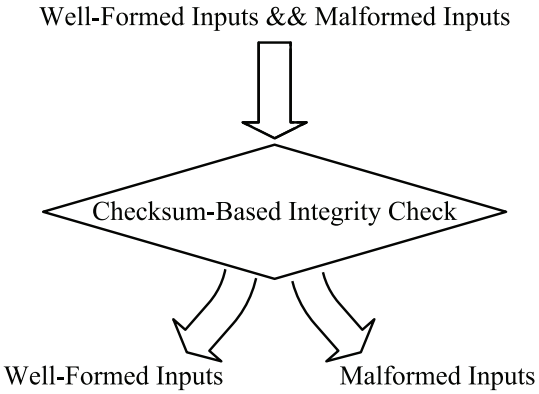


图 4.8: 校验和检测点行为示意图

和执行差异比对技术[100, 97]，分析目标程序在处理正常样本和畸形样本时的区别，定位程序执行的差异点，并进一步标识出校验和检测点。下面详细介绍在TaintScope系统中该方法如何自动定位程序中校验和检查点。

1. 识别处理正常样本时高污点依赖度分支

基于第二章中描述的细颗粒度污点分析技术，TaintScope首先识别目标程序处理正常样本过程中的高污点依赖度分支。正常样本的每个字节都赋予一个唯一的污点标签，TaintScope在程序执行过程跟踪这些污点标签的传播。

需要说明的是，污点传播过程中需要考虑标志寄存器eflags。例如，假设eax的污点标签是$\{6,7\}$，ebx的污点标签是$\{8,9\}$，那么执行指令"ADD eax, ebx"后，eax的污点标签更新为$\{6,7,8,9\}$；同时，由于ADD指令影响标志寄存器，eflags的污点标签也更新为$\{6,7,8,9\}$。

此外，污点传播过程考虑了间接数据依赖关系。例如，假设ecx的污点标签是$\{100\}$，同时0x8000000为起始地址的数据内容不依赖于污点数据，指令MOV eax, [0x8000000+ecx*4]读取地址"0x8000000+ecx*4"中数据并赋值给eax。虽然读出的数据并不依赖于污点数据，但作为访问该数据的索引，寄存器ecx依赖于污点数据，那么指令执行后eax也赋予污点标签$\{100\}$。

在程序执行条件跳转指令之前，TaintScope检查当时eflags的污点标签的个数，如果超过预定的阈值，该条件跳转指令就被当做一个高污点依赖度分支。TaintScope记录这些高污点依赖度分支地址、这些分支所依赖的污点数据标签及

分支跳转是否发生（taken或not-taken）。

由于这个过程中，我们选择多个正常样本，需要运行目标程序多次。TaintScope进一步计算两个集合$\mathcal{P}_1$和$\mathcal{P}_0$，其中$\mathcal{P}_1$保存程序执行过程中一直发生跳转的高污点依赖度分支语句，而$\mathcal{P}_0$保存程序执行过程中一直发生未跳转的高污点依赖度分支语句。

2. 识别处理畸形样本时高污点依赖度分支

TaintScope修改上述过程中使用的正常样本而产生畸形样本。目标程序处理正常样本s过程中，高污点依赖度分支p_i（$p_i \in \mathcal{P}_1 \cup \mathcal{P}_0$）所依赖的污点标签记做$|p_i|^s$。正常样本$s$中能影响到一直跳转/不跳转的高污点依赖度分支的字节集记做$\bigcup |p_i|^s$。对每个正常样本，TaintScope随机修改$\bigcup |p_i|^s$中的一个字节生成畸形数据。

类似地，TaintScope识别目标程序处理畸形样本过程中的高污点依赖度分支，这里不再赘述。值得说明的是，TaintScope 特别跟踪被修改的字节。只有一个高污点依赖度分支受被修改字节影响时，TaintScope才会记录该分支及其跳转行为。换言之，假设被修改字节的污点标签是i，高污点依赖度分支处`eflags`的污点标签集合为S，只有$i \in S$成立时，TaintScope才输出这个高污点依赖度分支并记录是否发生跳转。

对于目标程序处理畸形样本过程中遇到的高污点依赖度分支，TaintScope同样计算两个集合$\mathcal{P}_1'$和$\mathcal{P}_0'$，其中$\mathcal{P}_1'$保存程序处理畸形样本过程中一直发生跳转的高污点依赖度分支点，而$\mathcal{P}_0'$保存程序处理畸形样本过程中一直未跳转的高污点依赖度分支点。

3. 定位程序执行差异点

基于上述两步，TaintScope计算$(\mathcal{P}_1 \cap \mathcal{P}_0') \cup (\mathcal{P}_0 \cap \mathcal{P}_1')$，即识别出目标程序处理正常样本和畸形样本过程中，跳转行为完全不同的高污点依赖度分支。$(\mathcal{P}_1 \cap \mathcal{P}_0') \cup (\mathcal{P}_0 \cap \mathcal{P}_1')$包含的分支点，被认为是校验和检测点。

4. 识别校验和域

校验和域的识别在协议格式逆向分析工作中已经有比较好的处理方法（例如[69, 48, 126]）。本文实现类似于Tupni[69]系统的校验和域识别方法。

为识别正常样本中的校验和域，TaintScope在定位校验和检测点基础上，在程序处理正常样本的执行轨迹上，实现了后向切片（backward slice）[165, 22, 179]。第4.6.1节已经介绍了校验和通常用于检查一段数据的完整性，而为降低空间需求，校验和域本身只需几个字节。因此重新计算出来的校验和$H(D)$通常依赖于很多污点标签，是一个高污点依赖度值；与此同时，程序输入中原有校验和域的处理$T(C)$同样仅依赖于有限几个污点标签，是一个低污点依赖度值。

TaintScope以校验和检测点的标志寄存器`eflags`为起始点，根据数据依赖关系，生成后向切片。在计算切片的过程中，如果发现一个高污点依赖度值和一个低污点依赖度值相互操作，停止切片。低污点依赖度值所附属的污点标签，就当做校验和域。

由于校验和域仅用于完整性检测，一旦样本通过了完整性校验，校验和域一般不会再影响程序的控制流。因此，为进一步确保能准确识别校验和域，TaintScope可以进一步检查在校验和检测点之后，校验和域是否参与其他路径约束。

特例说明

下面讨论两种特殊情况。一个格式中可能存在多个校验和域，每个校验和域分别保存不同数据段的校验和。例如，图4.9展示了一个多校验和域的示例。在这个格式中，每个数据部分`D`都有一个相应的校验和域`C`。目标程序通常逐数据块验证完整性。假设随机修改只破坏了第i个数据段，那么前$i-1$个数据段仍然可以通过校验和检查。这就意味着目标程序处理这个畸形样本过程中，校验和分支的跳转行为并不唯一。

TaintScope在设计上已经考虑到了这种情况。在识别目标程序处理畸形样本时的高污点依赖度分支阶段，TaintScope特别跟踪被修改的字节。只有一个分支语句既满足高污点依赖度条件，又受到修改字节的影响时，TaintScope才记录该分支语句的行为。因此，虽然部分数据块能够通过校验和检测，TaintScope只会关注受修改后字节影响的行为；也就是说，校验和检测分支点仍然保存在$\mathcal{P}_0'$或而$\mathcal{P}_1'$中。

另外，当一个校验和值超过一个原语类型（例如整数类型`int`）所能表达范

图 4.9: 多校验和域示例

围时，一个校验和检测可能需要多个分支点实现。例如，MD5校验和值长16字节，因此在32位体系上，一个MD5校验和检查通常由4个整数比较来实现。从程序控制流图拓扑结构上，这类多个连续的校验和检测点会形成“复合分支”结构[58, 163]。源代码中由逻辑运算符（例如逻辑与&&、逻辑或||）连接的谓词判断语句，经编译成二进制后，对应的控制流就会形成复合分支结构。图4.10展示了一个复合分支结构。图中菱形代表谓词判断，标记T和F的边代表谓词为True和为False时的控制流转移。

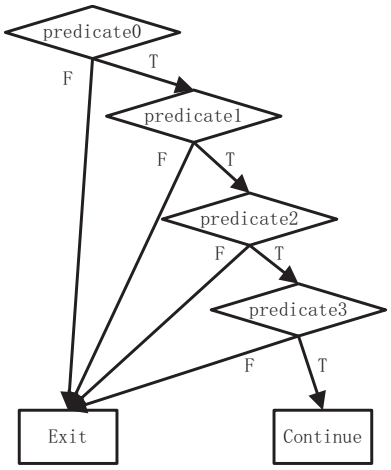


图 4.10: 多校验和检测点示例

北京大学韦韬博士在二进制程序逆向分析过程中控制流结构识别上开展了深入研究，开发了反编译平台Bestar [2, 159, 163]，能够准确识别二进制程序中的复合分支、嵌套循环等控制流结构。为了处理这种多校验和检测点问题，TaintScope利用了Bestar的反编译过程中控制结构识别信息。如果复合分支结构中一个分支点被识别为校验和检测点，TaintScope会进一步检查该复合分支结构中其他的谓词。如果这些谓词的操作数都是连续的、高污点依赖度内存单元，TaintScope会将整个复合分支结构当做一个校验和检测点。

检查这种较长校验和值的另外一种途径就是利用体系底层指令提供的内存比

较操作。例如，x86提供rep cmpxx系列指令，支持对不同内存区域逐一比较。高级编译器会充分利用底层指令的优势对目标程序进行优化。rep指令本身具有循环功能，但仅对应一条指令。TaintScope能有效定位和绕过这类校验和检查实现方式。

4.6.3 二进制修改和模糊测试

在这个阶段，TaintScope修改原始程序，在校验和检测点引入直接跳转，强制程序接受任何数据，从而绕过数据完整性检测。

如果目标程序处理正常样本时，校验和检查点一直发生跳转，意味着跳转目标处继续处理样本，TaintScope将该指令替换为直接跳转指令。例如，校验和判断后的条件跳转指令为“jcc NormalExecution”类型，TaintScope把该指令修改为“jmp NormalExecution”类型。这里jcc代表x86指令集中的条件跳转指令，jmp是x86指令集中的直接跳转指令。

如果目标程序处理正常样本时，校验和检查点一直不发生跳转，意味跳转指令的下一条指令将会继续处理样本，TaintScope将该条件跳转指令替换为等长的空操作指令。例如，假设校验和判断后的条件跳转指令为“jcc checksumErr”类型，TaintScope直接用等长的nop指令替换这个跳转指令。

修改后的二进制程序不会检测数据的完整性，因此，所有输入都会“通过”校验和检查，能够对校验和检测点之后的代码进行测试。本文将在第五章讨论畸形数据生成方式。

4.6.4 校验和域自动修复技术

如果模糊测试过程中，某个畸形数据导致修改后的程序发生崩溃，但却无法导致原始程序崩溃，通常是由于这个畸形数据无法通过原始程序中校验和检测。本文基于混合符号执行技术（见第三章），修复畸形数据的校验和域。

给定一个导致修改后程序崩溃的畸形数据，TaintScope首先记录原始程序处理该数据的执行轨迹，然后对执行轨迹重放，把畸形数据中校验和域部分作为符号值，收集执行轨迹上对这些符号值的约束；待重放至校验和检测点时（畸形数

据未能通过校验和检查），TaintScope已经收集了执行轨迹上对校验和域的约束，TaintScope在已收集的约束中，进一步添加校验和约束，通过约束求解器判断是否存在可行解。如果约束求解器能够求解路径约束，TaintScope根据求解结果，修改畸形样本中的校验和域。如果修复后的畸形数据能够导致原始目标程序崩溃，意味着TaintScope确实发现了一个安全漏洞。

与现有混合符号执行系统相比，本文提出的校验和域修复技术的核心特征在于仅将部分输入数据（校验和域）视为符号值。校验和约束条件“$H(D) == T(C)$”难于求解的根本原因在于散列函数$H()$过于复杂。然而，本文并未数据部分D视为符号值，$H(D)$部分是一个运行时可确定的常数。制约校验和约束条件能否求解的因素只有$T()$部分。

不同文件/协议格式对校验和域有不同的处理。例如，Tar文件格式中[154]的校验和值以八进制数字形式保存在文件中；Intel HEX文件格式[96]的校验和值以十六进制数字形式保存在文件中。目标程序会根据文件/协议格式的约定，在校验和检测之前对读入的校验和域做不同的处理。

```
//assume both x and y are symbolic values
//and are in the range of [0,3];
int A[4] = {0, 1, 2, 3};
A[x]     = 4;
b        = A[y]; //what is b's value?
```

图 4.11: 符号地址推理示例

为了使TaintScope能够应对复杂的校验和域处理过程（即求解复杂$T()$部分），我们在第三章混合符号执行框架基础上，扩展实现了部分符号化地址推理。

符号化地址指的是内存读写操作中，所访问的内存地址是符号值。我们用图4.11所示源代码阐述一下符号化地址推理的困难。在这段代码中，`A[4]`是一个含有4个元素的整型数组；假设`x`和`y`都是符号变量并且都在[0,3]范围内，那么`A[x]`可能覆盖数组中任意一个元素，而`A[y]`可能读取数组中任意一个元素。在符号执行模式下，变量`b`的取值就变得异常复杂，可以用下面约束描述`b`的取值情况：

```
(b==4 && x==y) ||
(b==0 && x!=y && y==0) ||
(b==1 && x!=y && y==1) ||
(b==2 && x!=y && y==2) ||
(b==3 && x!=y && y==3)
```

符号化地址精确推理是混合符号执行的一个难题。在符号执行过程中，一旦索引值是符号变量，那么对数组的访问就涉及到符号地址的推理。由于数组是高级编程语言的基本数据类型，符号化地址是面向源代码的符号执行工具无法回避的问题。在约束求解器社区和程序分析领域的推动下，数组推理有了一定发展，很多面向源代码的符号执行工具（例如EXE[50]、Klee[49]）逐渐支持符号索引。很多约束求解器提供了数组读/写谓词，为符号执行系统收集数组约束条件提供了方便。例如，在求解器具有数组推理能力时，上述对图4.11中变量b的取值就可以描述为：

`A = write(A, x, 4) && b = read(A, y)`。

这里，`write`原语描述向数组（`A`）的索引位置（`x`）写下指定的值（4）；`read`原语描述从数组（`A`）读取索引位置（`y`）的值。求解器会推理`x`和`y`之间不同关系下变量`b`的不同取值。

在二进制程序层进行符号化地址精确推理尤为困难。数组推理的重要前提是知道数组的起始地址和长度。高级语言中程序变量是对内存地址的抽象描述，为程序分析提供了很多便利。例如，面向高级语言的程序分析可以在数组变量的声明或创建过程中，获悉数组的起始地址和长度。然而，在二进制程序层，程序直接对内存地址访问，已经没有高级语言中所谓的“变量”概念[27]。高级语言中，不同的变量就代表不同的内存区域，实现了对内存区域的抽象划分；但是，二进制程序中，整个内存空间是一个“超大”数组，变量之间没有明确的边界信息。

文献 [77]尝试在二进制层实现精确指针推理，但是只能通过劫持内存分配函数，获取堆对象起始地址和大小，依然无法处理局部变量；BitBlaze[41]是最早在二进制程序层实现符号地址推理的系统。其核心思想在每次符号化内存读写时，

穷举所有可能的读写范围。这种方法必然导致执行效率严重下降。在BitBlaze后继的开发中，已经放弃了对符号地址的推理[140, 47]。

在校验和感知的模糊测试场景下，我们只需要将校验和域当做符号值，因此混合符号执行过程中符号地址出现次数有限。综合考虑到混合符号执行效率和具体需要，TaintScope实现了部分符号地址推理。具体而言，TaintScope首先在反汇编器IDA Pro [95]基础上，对目标程序进行反汇编。IDA Pro会结合一系列启发式策略，识别二进制程序中的全局数组和数组边界。某些情况下IDA Pro的识别结果还比较粗糙。如何改进全局数组及数组边界的识别是本文的未来工作之一。TaintScope实现了一个IDA Pro的插件，用于导出IDA Pro的全局数组识别结果。

在执行轨迹重放初始时期，TaintScope会利用约束求解器提供的API 为全局数组创建数组表达式，并根据全局数组的内容初始化这些数组表达式。在执行轨迹重放过程中，当TaintScope发现一个内存地址是符号地址时，TaintScope将检查该内存表达式的真实地址（已经保存在执行轨迹文件中）；如果这个真实地址在某个全局数组范围内，TaintScope将创建数组表达式。例如，假设TaintScope发现`0x00400000`是一个全局数组的起始地址，TaintScope将为这个全局数组创建一个数组表达式`Arr`；进一步，假设TaintScope发现一个内存读取操作的地址表达式`0x00400000+x*4`是符号值，而符号地址对应的真实地址`0x00400010`在全局数组范围内，TaintScope创建数组读表达式`read(Arr, x*4)`作为内存读取操作的返回值。

综上所述，校验和感知的模糊测试技术主要由三部分构成。首先，TaintScope在细颗粒度污点分析的基础上，识别程序执行过程中的高污点依赖度分支点，并进一步定位目标程序处理正常样本和急性样本时的路径执行差异点，准确识别校验和检测点；接下来，TaintScope在校验和检测点修改目标程序，禁止目标程序检测输入数据的完整性。TaintScope生成大量畸形数据，对修改后的目标程序进行测试。最后，针对导致修改后程序崩溃的畸形样本，TaintScope在混合符号执行技术的基础上，增加了对符号地址的推理，实现了校验和自动修复技术，能够修复畸形样本，使之通过原始程序的校验和检查。

本文已经在第二章和第三章技术和工具基础上，实现了TaintScope模糊测试

原型系统。在约束求解器的选择上，TaintScope最初选用STP[85, 158]求解器。但是使用过程中发现，STP求解器只能运行在Linux平台下，而且自身存在内存泄露问题。在综合比较其他约束求解器特性后，TaintScope目前采用了微软开发的Z3求解器[177]。与STP相比，Z3的求解效率更高，而且具有跨平台特性。值得说明的是，求解器是TaintScope系统的后端模块。在设计上，TaintScope并不局限于某种特定的求解器。

4.7 实验结果

4.7.1 实验设置

表 4.1: 测试程序信息

Category	Application	Version	OS	File Format	Checksum Algorithm
Image Viewer	Google Picasa	3.1.0	Windows	PNG	CRC
	Adobe Acrobat	9.1.3	Windows	PNG	CRC
Network Tool	Snort	2.8.4.1	Linux	PCAP	TCP/IP
	Tcpdump	4.0.0	Linux	PCAP	TCP/IP
Misc	ClamAV	0.95.2	Linux	CVD	MD5
	GNU Tar	1.22	Linux	Tar Archive	Tar
	objcopy	2.17	Linux	Intel Hex	Hex
	Open-vcdiff	0.6	Linux	VCDIFF	Adler32

在实验部分，为了测试TaintScope系统的有效性，本文选取了Windows、Linux等平台上多款流行应用程序。表4.1列出了测试程序的具体信息。其中，第二列给出了程序的名称，第三列给出了程序的版本信息，第四列给出了程序所运行的操作系统信息。下面简单介绍一下这几款程序。

- `Picasa`是由Google开发，广泛使用的图像查看、编辑软件；
- `Acrobat`是Adobe公司开发的PDF格式编辑、阅览、制作软件；
- `ClamAV`是Linux环境下最流行的反病毒软件；

- Snort和Tcpdump分别是广泛应用的网络入侵检测系统和网络数据采集分析工具；
- objcopy是GNU工具集中用于转换二进制目标代码格式的工具；
- GNU Tar是GNU版本的、用于创建和操作归档文件（archive file)的程序；
- Open-vcdiff是VCDIFF文件格式[103]编解码工具。

进一步，表4.1中第五列给出测试程序处理的文件/协议格式类型，第六列介绍这些格式采用的校验和算法。本文选择了MD5、CRC、Adler32等常见校验和算法。TaintScope会定位测试程序处理相应文件过程中的校验和检测点，并进行校验和域修复。下面介绍一下这些格式及校验和算法。

- PNG[38]是一种常见的图像文件格式。一个PNG图像内部由多个数据块（chunk）组成，每个数据块最后4字节保存该数据块之前内容的CRC校验和。Picasa和Acrobat都接受PNG格式的图片，前者可以显示图片，后者将图片转换成PDF格式。在实验中，本文以Picasa和Acrobat为目标，使用TaintScope 定位这两个程序中CRC校验和检测点，并修复畸形PNG图片中的校验和。实验中使用的PNG图片均从互联网上下载。
- CVD是反病毒软件ClamAV的病毒特征库的文件格式。为防止病毒特征库文件被破坏，CVD文件中含有一个MD5校验和。ClamAV在打开病毒特征库文件时，首先会验证特征库的完整性。
- PCAP [131]是一种通用的用于保存网络报文的格式。TCP/IP网络协议中都含有校验和信息。实验中，我们向Snort和Tcpdump输入PCAP格式文件，强制这两个工具检验TCP/IP报文的完整性。
- Intel Hex[96]一种以ASCII文本方式保存单片机或其他处理器的目标程序代码的文件格式。Intel Hex文件中，每一行的最后两个字节保存该行的校验和。Intel Hex校验和算法的基本思想是对数据部分累加，进而计算累加值的补码，最后保留计算结果的最低有效字节（least significant byte）作为校

验和。objcopy程序可以对各种目标程序代码进行转换，也支持Intel Hex格式。

- 归档文件（archive file)是UNIX平台上常见的文件格式，通常也被称作Tar包。一个Tar包是对一个或多个文件的联结。在Tar包里，每个文件前都有一个512字节的文件头。文件头中保存了相应文件的基本信息，如文件名、文件大小、最后修改时间等。同时，文件头中也保存着整个文件头的校验和，以便解包过程中验证文件头的完整性。Tar包中校验和算法的基本思想是对文件头中字节累加后，以八进制数字形式保存。TaintScope将处理GNU Tar程序解包过程中校验和检查问题。

- VCDIFF（Generic Differencing and Compression Data Format）是一种为差分编解码（delta encoding/decoding）需求而设计的文件格式[103]。值得提出的是，VCDIFF本身并未指定校验和信息。Open-vcdiff对VCDIFF进行了扩展，增加了Adler32校验和域。实验中，TaintScope将定位和绕过Open-vcdiff处理扩展VCDIFF格式文件过程中的校验和验证问题。

此外，对于Windows操作系统上的应用程序，TaintScope运行在Windows XP SP3 操作系统的平台上（硬件环境：Intel Core 2 Duo CPU (3.0 GHz)、3GB 内存)；对于Linux操作系统上的应用程序，TaintScope运行在Fedora Core 10 操作系统的平台上（硬件环境：Intel Core 2 Duo CPU 2.4 GHz、4GB 内存)。

4.7.2 校验和检测点定位

TaintScope系统首先需要识别目标程序处理正常样本过程中的高污点依赖度分支。判断高污点依赖度分支时，需要预先指定一个阈值。表4.2给出了不同阈值下（8，16，24和32）高污点依赖度分支的个数。

表4.2的前三列指明了程序名称、输入文件格式以及校验和算法，这里不再赘述。第四列|W|表明TaintScope共运行了多少正常样本。HTDB8、HTDB16、HTDB24和HTDB32分别指在不同阈值下，TaintScope识别的高污点依赖度分支语句的数目。

表 4.2: High-taint-degree Branches(HTDB) 识别结果

Executable	Format	Checksum	|W|	HTDB8	HTDB16	HTDB24	HTDB32
Picasa	PNG	CRC	3	1,079	881	544	544
Acrobat			3	16,472	10,537	10,537	10,537
Snort	PCAP	TCP/IP	3	2	2	1	1
Tcpdump			3	8	2	1	1
ClamAV	CVD	MD5	1	39	39	32	2
open-vcdiff	VCDIFF	Adler32	3	19	1	1	0
GNU Tar	Tar Archive	Tar	3	34	34	28	5
objcopy	Intel HEX	Intel HEX	3	38	23	23	23

由表4.2中数据可见，一般情况下阈值越大，高污点依赖度的分支点越少。其中，`Picasa`和`Acrobat`运行过程中存在大量的高污点依赖度分支。原因在于PNG图片中像素信息以压缩形式保存。由于解压缩过程中污点依赖关系会广泛扩散，导致解压缩后的一个字节可能依赖于解压缩前所有内容。对解压后数据的任何判断都会成为一个高污点依赖度分支。所以，仅靠识别高污点依赖度分支并不能定位校验和检测点。

接下来，TaintScope会对正常样本进行修改，生成畸形样本。具体修改策略见第4.6.2节。TaintScope进一步识别目标程序处理畸形样本过程中的高污点依赖度分支，并定位程序执行差异点（即校验和检测点）。程序执行差异点指的是程序处理正常样本中和畸形样本中跳转行为完全不一致的高污点依赖度分支。

表4.3给出了最终定位结果。第二列|M|代表TaintScope生成的畸形样本的个数。CCP8、CCP16、CCP24和CCP32表示在不同阈值下TaintScope定位到程序执行差异点的个数。需要特别说明的是，表4.3中的*标志说明在相应阈值下TaintScope并未定位到校验和检测点。

手工分析确认了TaintScope能够定位到校验和检测点。虽然在第一步中TaintScope识别了大量的高污点依赖度分支，但是经过第二步后，TaintScope能准确定位到程序校验和检测点。例如，`Acrobat`对PNG图像的校验和检测点位于名为`ImageConversion.api`的动态库中。TCP报文中有两个校验和域，一个

是IP首部的校验和，另一个是TCP首部和数据的校验和。TaintScope在Snort和Tcpdump 程序中准确定位到了这两处校验和检测点。

表 4.3: 程序执行差异点定位结果

Executable	|M|	CCP8	CCP16	CCP24	CCP32
Picasa	9	1	1	1	1
Acrobat	9	1	1	1	1
Snort	9	2	2	1*	1*
Tcpdump	9	2	2	1*	1*
ClamAV	3	1	1	1	1
open-vcdiff	9	1	1	1	0*
GNU Tar	9	1	1	1	1
objcopy	9	1	1	1	1
Detected?		√	√	Miss 2	Miss 3

但是，过高的阈值会导致TaintScope丢失部分校验和检测点。例如，在IPv4报文中，IP首部（包含校验和域）通常只有20字节[135]，除非含有选项字段。而IP首部的校验和仅覆盖IP首部，并不覆盖IP数据报中的任何数据。因此，对IP首部的校验和检查点的污点依赖度通常只有20。在Snort和Tcpdump处理IP报文的过程中，如果选择24或32作为污点阈值，TaintScope就不会定位到对IP首部的校验和检测。类似的，当污点阈值为32时，TaintScope未能定位到open-vcdiff中的Adler32 校验和检测点。基于上述实验结果以及对广泛文件/协议格式的调研，TaintScope缺省采用16作为污点阈值。

上述实验结果中，并没有直接给出TaintScope定位校验和检测点的时间代价。整个校验和检测点定位过程的时间开销依赖于TaintScope使用了多少正常样本和多少畸形样本，以及对目标程序进行细颗粒度污点分析的代价。根据我们的经验，一般情况下，3-5个正常样本和十几个畸形样本就足够用于定位校验和检查点。本文会在第五章介绍对每个样本细颗粒度污点分析的时间代价。总体而言，针对每个样本的细颗粒度污点分析通常需要1-5分钟，因此整个校验和检测点定位过程的总时间花销在数十分钟。

4.7.3 校验和修复

表 4.4: 校验和域识别及修复结果

Executable	File Format	# Checksums	Checksum Length	Repaired?	Time (s)
Acrobat	PNG	4	4	√	41
Picasa					106
tcpdump	PCAP	8	2	√	58
open-vcdiff	VCDIFF	1	5	√	9
ClamAV	CVD	1	32	√	34
tar	Tar Archive	3	8	√	226
objcopy	Intel HEX	4	2	√	121

本节介绍TaintScope对校验和域的识别以及修复能力。表4.4展示了实验结果。第三列表示TaintScope在正常样本中发现多少校验和域，第四列表示每个校验和域的长度（以字节为单位）。在上述校验和测试点精确定位的基础上，TaintScope也精确地定位了正常样本中的校验和域。例如，对于PNG图片，二进制文件编辑器010editor[19]提供了解析模板，能够根据PNG格式对PNG图片解析。我们对比了TaintScope的识别结果与010editor的解析结果，发现正常的PNG样本中，存在IHDR、PLTE、IDAT和IEND等四个块，共包含4个校验和域，每个校验和都占用4字节。我们也手工分析了其他格式的样本，确认了TaintScope识别的准确性。

进一步，为测试TaintScope能够修复校验和域，我们特意修改了正常样本的校验和域，生成了畸形样本。由于数据部分并未发生改变，这意味这些畸形样本的正确校验和域就是原始的校验和域。我们使用TaintScope修复这些畸形样本的校验和域。如果TaintScope生成的校验和域原始样本一致，就意味着TaintScope生成了正确的校验和；否则，表明TaintScope没有生成正确的校验和。

表4.4的第五列指出TaintScope为这些畸形样本生成了正确的校验和域，而第六列给出了TaintScope修复校验和域进行混合符号执行的时间代价。由于TaintScope仅将校验和域的数据当做符号值，混合符号执行过程中收集到的路径约束很少，现有求解器很容易求解正确的校验和域。例如，`ClamAV`的恶意代码

特征库CVD文件中采用MD5校验和。但是，ClamAV计算出数据校验和后，直接通过简单的字符串比较语句（strcmp）比较特征库中保存的校验和和新计算出来的校验和。因此路径约束只有字符串比较类型的约束，现有求解器可以迅速求解。

总体而言，TaintScope需要几分钟修复一个畸形样本的校验和域。由于并不是每一个畸形样本都会导致程序崩溃，TaintScope系统中将校验和修复作为最后一步。只有一个畸形样本导致修改后的程序崩溃，TaintScope才会修复该样本中的校验和域。因此，校验和域修复的时间代价是可接受的。

4.7.4 漏洞挖掘

基于校验和感知模糊测试系统，本文在Adobe Flash Player、CamlImage、wxWidgets、Dillo等软件中发现了多个零日安全漏洞，均已得到软件开发方的确认，也已被国际安全漏洞库CVE收录。本文会在第五章中集中报告模糊测试过程中已发现的零日安全漏洞。

```
509 wxPNGHandler::LoadFile(wxImage *image,
...//ignore the CRC checks
575 for (i = 0; i < height; i++){
577   if ((lines[i]=(unsigned char*)malloc(width*4))
          == NULL){
579     for ( unsigned int n = 0; n < i; n++ )
580       free( lines[n] ); //first time free()
581     goto error;
...
621 error:
...
630  if ( lines )
631  {
632    for ( unsigned int n = 0; n < height; n++ )
633      free( lines[n] ); //second time free()
```

图 4.12: wxWidgets指针多次释放漏洞(CVE-2009-2369)

下面介绍一下TaintScope系统在著名的跨平台图形界面开发库wxWidgets中发现的内存多次释放安全漏洞。该漏洞存在于wxPNGHandler::LoadFile函数中，如图4.12所示。其中，wxPNGHandler::LoadFile用于加载PNG图片。其中，

第510-575行被省略的代码里，wxPNGHandler::LoadFile检查了输入PNG图片的校验和。wxPNGHandler::LoadFile拒绝加载不能通过校验和检查的图片。

第577行中变量width来源于输入PNG图片，用于保存图片的宽度。第577行代码负责为图片的每一行分配像素空间。过大的width会导致内存空间迅速耗竭，并导致内存分配函数malloc失败返回NULL。一旦malloc内存分配失败，为防止内存泄露，第579-581行代码会回收已经分配的内存。然而，在错误处理部分（第630-633行），wxPNGHandler::LoadFile 会进一步释放内存，产生指针两次释放安全漏洞。

由于wxPNGHandler::LoadFile函数开始阶段会检查输入PNG图片的校验和，导致传统模糊测试工具失效。实验过程中，我们采用FileFuzz工具[80]对wxWidgets做了长时间（8小时）的测试，并未发现任何安全漏洞。

TaintScope精确定位到wxWidgets中PNG校验和检查点后，修改了wxWidgets，禁止了校验和检查功能。对于修改后的wxWidgets，FileFuzz工具在30分钟内就触发了图4.12中的安全漏洞。进一步，TaintScope成功修复了畸形样本中的校验和域，并在原始wxWidgets中重现了该漏洞。该漏洞已经活动软件开发方确认和致谢，并被国际漏洞库CVE收录（漏洞编号CVE-2009-2369）。

由此例可见，校验和问题导致传统模糊测试工具无法对目标程序做到有效测试。TaintScope能够为有效解决校验和检验问题，拓展了传统模糊测试工具的应用场景。

4.8 讨论

信息安全经典CIA模型[35]强调信息的机密性（Confidentiality）、完整性（Integrity）和可用性（Availability）。其中，完整性和模糊测试存在密切关系。很多应用程序处理输入数据（本地文件或网络报文）时并未验证数据的完整性。因此，传统模糊测试工具通过修改正常而生成的畸形数据可以对这些程序安全测试。但是一旦应用程序采用了完整性验证机制，畸形数据无法通过完整性检测，就意味着传统模糊测试工具失效。

完整性检测机制实现机制大抵分为三类，散列函数（即校验和机制）、消息认证码MAC（Message Authentication Codes）和数字签名（Digital Signatures）。

详细而言，散列函数主要用于检测数据传输过程中发生的偶然破坏，而无法对抗人为的恶意破坏。因此，本文提出了修改目标程序，绕过校验和检测的方案；进而结合混合符号执行技术，修复畸形数据的校验和域。

本文提出的方案并不适用于基于消息认证码或数字签名的完整性检测。消息认证码和数字签名在保护消息完整性的同时也提供了对消息的认证，能够发现数据传播过程中的任何破坏（包括偶然和人为破坏）。消息认证码或数字签名都需要数据的产生方和数据的使用方共享密钥信息。而第三方（即模糊测试方）不知道密钥信息时，即使TaintScope能够精确定位到程序中消息认证和数字签名检测点，也无法为畸形数据生成正确的消息验证码或数字签名。

另一方面，正因为消息认证码或数字签名都需要共享密钥信息，这两类完整性验证机制在通用应用程序中并不流行。通用应用程序通常用户量很大，而每个用户生成的数据需要和其他用户共享。这些应用程序不关心数据的认证性，仅关注数据的完整性以检测传输过程中差错。因此，本文的工作的支撑下，传统模糊的是工具也能够对这些应用程序进行安全测试。如何克服基于消息认证码或数字签名的完整性检测以及其他消息认证机制将作为本文的未来工作。

同时，数据加密和压缩等技术也为模糊测试制造了诸多障碍。本文的方法为绕过数据加密和压缩提供了新的思路。本文工作在IEEE Security & Privacy 2010发表后，立刻被UC Berkeley大学J. Caballero等安全研究人员引用[47]。J. Caballero等人对本文方法扩展后，有效缓解了数据加密和压缩引发的问题，并成功应用于恶意代码分析领域。

4.9 本章小结

模糊测试技术是软件安全漏洞动态挖掘的主要方法，已经工业界和黑客社区得到广泛应用。然而，基于校验和机制的数据完整性检查成为传统模糊测试系统的重要障碍。

本文首次提出了一种校验和感知的模糊测试技术。该技术综合运用了混合符号执行与细颗粒度动态污点跟踪技术，通过自动定位程序中校验和检测点、修复畸形样本的校验和域，成功绕过程序中的校验和检查机制。该技术为运用传统模

糊测试发现深藏的软件漏洞扫除了障碍，扩展了模糊测试技术的应用范围，同时也提高了模糊测试的漏洞挖掘效率。

该项研究的学术论文“TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection”经过五轮严格评审，最终被IEEE Security & Privacy 2010录用，并被大会评选为最佳学生论文。

第五章　反馈式畸形样本生成技术研究

5.1　引言

畸形样本生成是模糊测试技术的关键。本章在第四章校验和感知模糊测试基础上，进一步讨论畸形样本生成技术。

对于复杂未公开数据格式，现有模糊测试工具大多采用对正常数据随机修改破坏的方式生成畸形数据。然而，这种畸形数据生成策略过于盲目，严重制约了模糊测试系统的效率和能力。例如，图5.1展示了一段简单的代码。当x为0xdeadbeaf时，代码会发生除0异常。但是，在32bit空间上随机生成x的方式仅有$1/2^{32}$的概率触发除0异常。

```
1. if(x==0xdeadbeef){
2.    return 100/(x-0xdeadbeaf);
3. }
4. else
5.    return 100;
```

图 5.1: 含有除0异常的代码片段

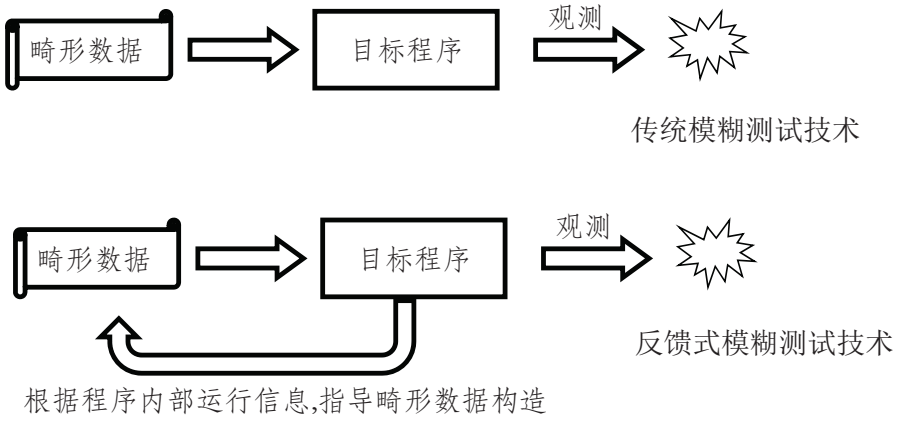


图 5.2: 传统模糊测试与反馈式模糊测试对比

究其原因，传统模糊测试采用黑盒测试的方式，在生成畸形样本时，并没有利用到程序内部状态信息。本文将利用程序执行过程中的运行时信息，指导畸形样本的生成，实现反馈式畸形样本生成技术，提高模糊测试的效率。图5.2展示了传统模糊测试于反馈式模糊测试的区别。

具体而言，本文实现了两类反馈式模糊测试技术，分别是基于细颗粒度污点分析的导向性样本生成技术和基于混合符号执行的样本生成技术。这两种畸形样本生成技术都已经集成到模糊测试平台TaintScope中。接下来，本文详细介绍这两类技术的设计与实现。

5.2 基于细颗粒度污点分析的导向性样本生成技术

5.2.1 方法原理

前文已经阐述过，在协议不公开或过于复杂时，传统模糊测试工具通常对正常样本进行修改、破坏，进而生成畸形样本。但由于缺乏先验知识，模糊测试工具无法判定修改正常样本的哪些部分，所以传统模糊测试工具经常采用两类修改策略：（1）随机位置修改为随机/边界值，或（2）每个位置依次修改为随机/边界值。

但是，正常样本是由安全相关数据和安全无关数据组成的。安全相关数据指的是能够影响程序执行过程中安全敏感操作的输入数据；而安全无关数据指的是对安全敏感操作不产生影响的输入数据。进一步，正常样本中只有很少一部分是安全相关数据，其余绝大部分数据是安全无关数据。值得说明的是，所谓的安全敏感操作（例如某些API调用、内存读写操作）依赖于具体的执行上下文。

例如，在BMP格式图片中，像素数据通常不会对安全敏感操作产生影响。对像素数据的修改，只会改变图像的显示效果，但并不会导致程序执行异常。但是，BMP图片文件头的控制信息（例如图像宽度、高度）通常会影响内存分配。根据[159, 178]的研究，这些数据内容通常与整数溢出漏洞相关，针对性的修改这类数据会起到事半功倍的效果。

导向性样本生成技术的核心思想是识别正常样本中的安全敏感数据后，针对性地修改这些字节生成畸形样本，进而使用这些畸形样本对目标程序进行测试。

由此生成的畸形样本，在保留原始样本正常结构的同时，关键数据片段被修改为异常数值。与传统模糊测试技术相比，导向性模糊测试避免了对目标程序整体输入空间的盲目枚举，生成的测试用例能够直接影响安全敏感操作，更可能触发安全问题。

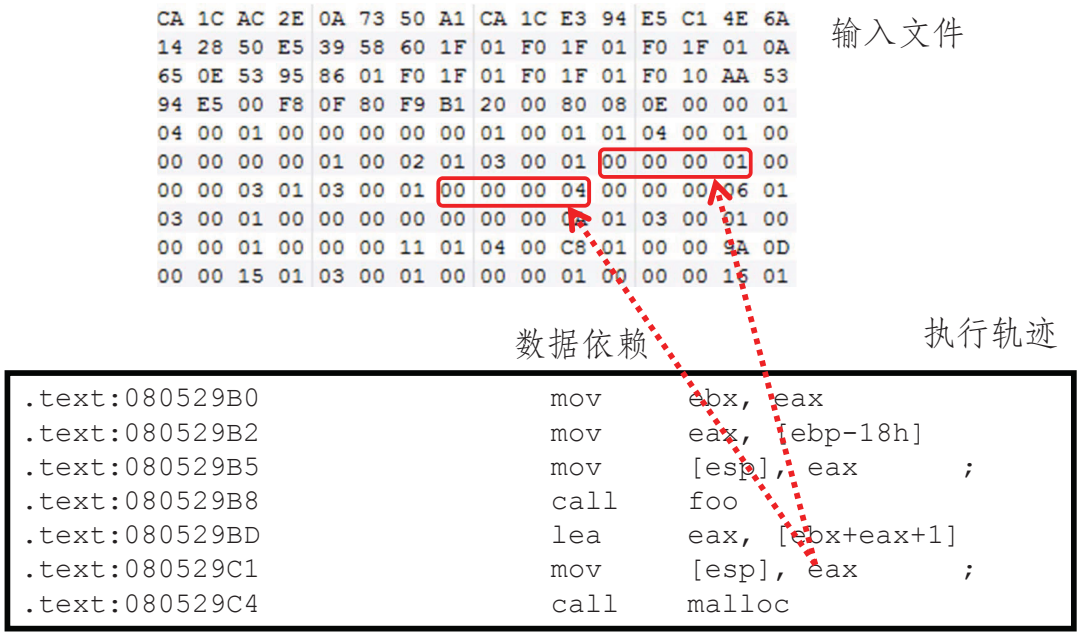


图 5.3: 导向性模糊测试示例

图5.3展示了导向性样本生成技术的示例。图中上部分代表一个正常的二进制输入文件，下部分代表目标程序的执行轨迹。图中显示的轨迹中最后一条指令调用内存分配函数malloc，而传入的参数来源于寄存器eax。图中红色虚线箭头代表执行0x080529C1处指令时，寄存器eax依赖于二进制文件中被红色框图标识的字节。以内存分配函数当做安全敏感操作时，导向性模糊测试将针对性的修改红色框图标识的字节。

5.2.2　设计与实现

导向性模糊测试的关键在于如何识别正常样本中的安全相关数据，即哪些字节会影响到安全敏感操作。细颗粒度污点分析技术为导向性模糊测试提供了支撑。

导向性模糊测试首先跟踪目标程序处理正常样本过程中输入数据的传播过程。本文第二章中，已经介绍了细颗粒度污点分析的原理和实现，这里不再赘述。具体而言，导向性模糊测试实现了字节精度的数据依赖关系跟踪，能够识别输入数据在程序执行过程的传播和使用。典型的安全敏感操作包括内存分配函数系列（例如`malloc`，`realloc`，`calloc`以及操作系统相关的函数`HeapAlloc`等）、字符串处理函数（例如`strcpy`、`strcat`）等。当然，用户可以根据先验安全知识或具体需求，指定特定的API或者指令地址作为安全敏感操作，导向性模糊测试将识别哪些输入字节会影响到这些操作。

识别安全相关数据后，导向性模糊测试将针对性的修改这些字节生成畸形样本。目前，本文整合了SPIKE[147] 、Peach[132]等传统模糊测试平台的攻击启发式策略，依次将安全相关数据替换为各种边界值/畸形值。通常安全相关数据只占正常样本的很小的比重，这显著降低了变异空间范围。导向性模糊测试可以结合各种修改策略对安全相关数据深度变换。

导向性模糊测试与校验和感知的模糊测试是无缝衔接的。如果TaintScope定位到校验和检测点并修改了目标程序，那么在模糊测试阶段就可以采用导向性样本生成技术；反之，如果目标程序未启用校验和检测，TaintScope直接采用导向性模糊测试技术对目标程序安全测试。

导向性模糊测试在实际应用中，取得了出色的成果，在多种流行应用软件中发现了零日安全漏洞。本章第5.4节将详细介绍实验结果。

5.2.3 讨论

与本文中导向性模糊测试最接近的工作是MIT研究人员设计的Buzzfuzz系统[86]。Buzzfuzz与本文研究同时期独立设计开发，其核心思想是通过导向性模糊测试技术协助软件开发方对软件产品测试。但是，Buzzfuzz与本文研究的关键区别在于，它基于源代码级代码植入技术，而本文的导向性模糊测试直接面向二进制程序。

Buzzfuzz在C语言植入平台CIL[124]的基础上，通过改写软件源代码，植入细颗粒度污点跟踪代码。植入后的程序经编译后，在运行过程中自动跟踪污点数据

的传播。Buzzfuzz的测试模块根据污点传播信息，生成新的测试数据。Buzzfuzz系统强烈依赖于目标程序的源代码。但是现代软件开发过程，由于模块化程度高，普遍会依赖于第三方库。例如，C语言开发的软件通常会依赖于libc库。源代码植入技术无法跟踪污点数据在第三方库执行过程中的传播，识别安全相关数据时存在漏报。MIT研究人员也把该思想应用于web安全分析，实现了面向PHP语言的安全检测[101]。

Google安全研究人员D. Will和O. Tavis在二进制植入平台Valgrind[125]基础上，实现了基于污点分析的模糊测试工具Flayer[75]。但是Flayer仅跟踪数据是否来源于输入，即仅跟踪数据的0/1污点属性，未实现细颗粒度污点分析，因此并不能定位安全相关数据的具体位置。

5.3 基于混合符号执行的样本生成技术

5.3.1 方法原理

基于细颗粒度污点分析的导向性模糊测试并没有解决测试过程中的代码覆盖率问题。例如，即使图5.1 中x被识别为安全相关数据，如何自动生成导致程序异常的x依然困难。因此，本文进一步实现了基于混合符号执行的样本生成技术。作为导向性模糊测试的互补，基于混合符号执行的样本生成技术能够对执行轨迹深度安全分析，也能够以代码覆盖率为指导，生成高覆盖率样本。

```
1. int x, y;
2. x = read_16bits();
3. y = read_16bits();
4. if(x>0 && y>0 && x*y*4>x && x*y*4>y*4){
5.   p = malloc(x*y*4);
6.   if(p==NULL) return 0;
7. ......
```

图 5.4: Adobe Flash Player 整数溢出漏洞（CVE-2010-2170）伪码

下面结合图5.4中代码，介绍基于混合符号执行的样本生成技术的原理。图5.4展示的是本文在Adobe Flash Player中发现的零日漏洞（CVE漏洞编号CVE-2010-2170）的原理代码。变量x和y都源于输入数据，其中read_16bits函数用于

从输入文件读入2字节。为防止整数溢出问题，代码中第4行对x和y 做了严格检查。对于通过检查的输入，代码第5行分配大小为x*y*4的内存空间。虽然第4行代码已经对输入数据做了严格过滤，恶意用户仍然可以构造使x*y*4溢出的输入。

但是，在第四行代码严格的过滤下，整个输入空间中只有很少一部分输入样本才能触发该漏洞。传统模糊测试或导向性测试都很难生成触发整数溢出的输入。混合符号执行技术为解决这类问题提供了支持。

符号执行技术把输入数据当做符号值，进而收集程序路径上对输入数据的约束条件。通过约束求解，就可以生产高覆盖率的测试样本。同样，在安全敏感操作前，如果操作数包含符号变量，可以进一步检查路径约束是否能确保该操作的安全。

5.3.2 设计与实现

本文第三章介绍了离线混合符号执行的框架，进一步，第四章在该框架基础上，实现了部分符号地址的推理。在上述工作基础上，本章对混合符号执行技术进一步改进，应用于测试例生成。

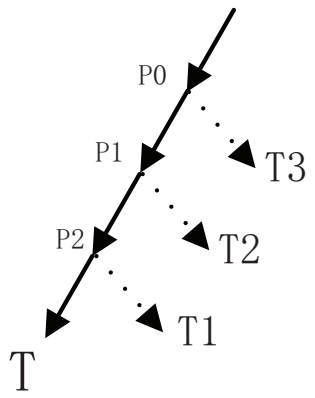


图 5.5: 路径约束求解生成新测试例示意图

混合符号执行的主要优势在于能够收集程序执行轨迹上对输入数据的约束条件。每个约束条件，都对应于一个分支语句。假设图5.5中的执行轨迹T上收集到的谓词条件分别是P0、P1、P2。这里黑色箭头代表执行过的控制流，黑色箭头衔接处代表存在分支语句；虚线箭头代表未执行的控制流。换言之，当P0 &&

P1 && P2成立时，程序执行轨迹T。那么，我们对谓词条件P2取反后，满足P0 && P1 && ¬P2的输入数据，就会驱动程序执行轨迹$T1$。进而，满足P0 && ¬P1的输入数据就会驱动程序执行轨迹$T2$。反之，如果没有满足¬P0的输入数据，那么意味着$T3$是不可执行的路径。

对于图5.1中代码，x=1时的执行轨迹为<1,4,5>。在此执行轨迹上，收集对x的约束条件，即x!=0xdeadbeef。通过对执行轨迹谓词逐一取反并求解，就能构造满足x=0xdeadbeef的输入，从而触发代码中的除0异常。

很多符号执行系统都基于上述路径谓词求解的方式生成新的测试用例，对程序执行空间进行遍历。典型的系统包括SAGE[91]、BitFuzz[47]、Klee[49]等。

混合符号执行的另外一个重要优势在于能够对执行轨迹进行深度安全分析。虽然很多测试用例驱动程序执行了含有漏洞的代码，但是并不能触发这些漏洞。例如，对于图5.4中代码，大量测试用例可以驱动程序执行malloc语句，但是并不能触发整数溢出漏洞。混合符号执行收集路径约束后，可以进一步检查路径约束是否确保安全相关操作的安全性。

例如，对于图5.4中代码，在真实输入为x=100、y=100的驱动下，程序的执行轨迹为<1,2,3,4,5>。在该轨迹上，把两次read_16bits的返回值视为16bit符号值，那么收集到的路径约束就是：
x∈[0,2^16-1] && y∈[0,2^16-1] && x>0 && y>0 && x*y*4>x && x*y*4>y*4。这里关系运算符>均为有符号比较。在调用malloc之前，可以通过约束求解判断传入malloc的参数（即符号表达式x*y*4）会不会溢出。如果求解器找到既满足执行轨迹约束条件又能导致x*y*4溢出的输入，就根据求解结果，生成畸形样本后测试目标程序。

然而，混合符号执行一个不可回避的问题就是执行效率。大量符号计算和约束求解会使混合符号执行效率非常低，无法应对大型应用程序。这也导致很多经典的符号执行系统主要应用于小型程序[49][47]或单元函数测试[143][90]。

为了提高混合符号执行效率，TaintScope 进一步借鉴了导向性模糊测试的思想（见第5.2节）：在混合符号执行过程中，仅把安全相关数据视为符号值，而对其他输入数据使用真实值。换言之，TaintScope在对路径离线重放过程中，仅把

一小部分输入数据当做符号值。虽然执行轨迹上记录了大量指令，只有一小部分执行需要被符号执行，从而避免了无谓的符号计算，提高了符号执行的效率。TaintScope将收集执行轨迹上对安全相关数据的约束条件，依次取反后求解生成新的畸形测试样例。

为了对执行轨迹进行深度安全分析，TaintScope实现了对内存访问越界、整数溢出、除0异常的检测。对于整数溢出漏洞，TaintScope会检查传入内存分配函数的size参数是否为符号表达式。如果是，TaintScope进一步检查该符号表达式在当前路径约束下能否发生溢出。如果当前的路径约束条件不能完全阻止整数溢出，TaintScope求解约束后生成一个新的输入，验证该整数溢出问题。

二进制程序中检测内存访问越界非常困难。二进制程序中缺乏变量的概念，无法直接获取变量边界信息，给内存访问越界的判定带来了不便。TaintScope采用两种策略获取矢量类型（例如数组类型）变量的边界。

对于堆上的元素，都是通过内存分配函数动态生成的；因此，TaintScope在内存分配函数的入口，获取分配的空间大小S；内存分配函数的返回时，TaintScope获取分配空间的起始位置D。所以，TaintScope可以用二元组（D，S）标识一个堆空间元素。在重放过程中，如果TaintScope遇到一个内存访问的地址是符号表达式X，而该内存访问的真实地址位于（D_0，S_0）堆区间内，意味着该内存访问在该堆区间的偏移为$X - D_0$。TaintScope将检查是否会超过S_0或小于0。如果求解器能找到满足解，意味着该执行轨迹上存在堆溢出漏洞。

为了识别栈空间变量（即局部变量）的边界，TaintScope在重放执行轨迹过程中，重构了函数调用栈。虽然TaintScope无法精确计算栈变量之间的边界，TaintScope以函数栈帧的边界作为局部变量的边界的估计。局部变量不会跨越栈帧结构，而且与所在的栈帧结构具有相同的生命周期。当TaintScope发现一个内存写操作所访问的地址位于栈帧($Base$,Top)内，这里$Base$代表栈帧的基址，Top代表当前栈顶，同时如果该内存写操作所访问的地址又是一个符号表达式X，TaintScope 将检查X能否超($Base$,Top)范围。

5.4 实验结果

TaintScope系统以校验和感知的模糊测试技术为基础，集成实现了基于细颗粒度污点分析的导向性样本生成技术和基于混合符号执行的样本生成技术。第四章在实验部分已经介绍了TaintScope系统识别和处理校验和问题的能力，本节介绍TaintScope系统样本生成技术的实验结果以及应用TaintScope系统过程中发现的真实安全漏洞。本节实验中的硬件平台信息与第四章在实验部分一致。

5.4.1 安全相关数据识别能力

TaintScope能够基于细颗粒度污点分析，识别正常样本中安全相关的数据部分，进而对这些数据部分修改生成畸形样本。因此，识别安全相关数据的能力直接影响TaintScope生成畸形数据。

这部分实验中，我们选择了三款软件，分别是Acrobat、Picasa以及ImageMagic软件包中Display程序。我们已经在第四章中介绍了前两款软件，这里再简单介绍一下ImageMagic。ImageMagic被称作图像魔术师，是Linux系统下功能强大、广为应用的图像处理软件包。Display程序是ImageMagic 软件包中显示图片的命令行工具。图5.1第一列给出了这三款软件的版本信息。

我们选择了各种常见格式的图片，包括PNG、BMP、JPEG等，见图5.1第二列。TaintScope在目标软件处理这些图片的过程中，检测能够影响内存分配函数size参数的输入字节。

图5.1第三列给出了输入图片的大小，第四列给出里能够影响内存分配的输入字节数。我们选择的输入图片都在几千字节，但是只有几十个字节会影响到内存分配。主要原因在于，图片中影响内存分配的一般只有图像的高度、宽度、色深等信息，其余像素数据不会影响内存分配。

图5.1第五列统计了目标程序处理相应输入文件时执行轨迹上x86指令的数目，最后一列给出了目标程序运行的时间。需要特别说明的时，第二章，我们介绍了离线的污点分析技术，而在本节实验中，我们采用的在线的污点分析技术，即在二进制程序植入技术的基础上，在目标程序执行过程中同时进行污点分析。由于在线污点分析和离线污点分析在实现上并没有本质区别，这里不再赘述。

图5.1最后一列给出植入后程序处理输入图片的时间消耗。由于植入了细颗粒度污点分析功能，程序性能下降严重。总体而言，目标程序大概需要几分钟处理一个输入。但是这个性能损失是值得的。一个几千字节的图片中，仅有数十个字节是安全相关数据。导向性测试阶段仅需要对这些字节扰动，大大降低了变异空间范围。

表 5.1: 安全相关数据识别结果

Executable	Input Format	Input Size (Bytes)	# Hot Bytes	# x86 Instrs	Run Time
Display V6.5.2-7	TIFF	5,778	18	191,759,211	2m53s
		2,020	18	82,640,260	1m30s
	PNG	5,149	9	19,051,746	1m54s
		1,250	29	47,246,043	1m8s
	JPEG	6,617	11	48,983,897	1m13s
		6,934	9	48,823,905	1m11s
Picasa V3.1.0	GIF	3,190	14	304,993,501	1m25s
		6,529	43	536,938,567	2m57s
	PNG	2,730	18	712,021,776	5m16s
		1,362	16	660,183,239	4m8s
	BMP	3,174	8	310,909,256	1m21s
		7,462	19	468,273,580	2m35s
Acrobat V9.1.3	BMP	1,440	6	658,370,048	4m25s
		3,678	6	663,923,080	5m2s
	PNG	770	21	297,492,758	3m8s
		1,250	12	354,685,431	4m31s
	JPEG	1,012	13	328,365,912	4m14s
		2,356	4	356,136,453	4m36s

5.4.2 混合符号执行能力

为了测试TaintScope的混合符号执行能力，我们选择了Adobe Flash Player (V10.0.45)。根Adobe公司的报告[21]，Flash Player是世界上最流行的软件，99%的

接入互联网的主机都安装了Adobe Flash Player。正是由于Adobe Flash Player拥有巨大安机量，很多黑客都致力于挖掘Adobe Flash Player中的安全漏洞。同时，为了提高自身软件的安全性，Adobe公司也在软件开发过程中引入了模糊测试机制。换而言之，Adobe Flash Player是一款被黑客和开发方充分"测试"过的软件产品。

Adobe Flash Player的输入文件格式SWF[152]非常复杂。图5.6显示了SWF文件的基本文件结构。在文件头和文件属性标签(Tag)后，各种其他种类的标签依次排列，直至最后的结束标签。图5.6中结构很清晰，但是各种标签却可以包含复杂的数据格式，支持内嵌各种图像、文本、音视频数据等。复杂的SWF文件格式也给基于规则生成测试用例的模糊测试工具带来很多困难。

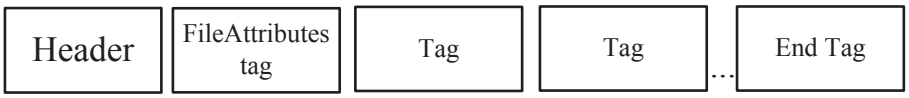


图 5.6: SWF文件格式结构

传统模糊测试工具很难在Adobe Flash Player中挖掘出安全漏洞。但是，TaintScope在基于混合符号执行生成测试例技术的支持下，成功地在Adobe Flash Player发现了3个零日漏洞。

实验设置

为了获得原始的正常SWF文件，我们使用了SWFTools工具[153]，将随机选择的4副PNG图片和4副JPEG图片转换成8个SWF文件。接下来，TaintScope分别记录Adobe Flash Player处理8个SWF文件的执行轨迹。后继的混合符号执行都在这8条执行轨迹上开展。

值得说明的是，为了防止SWF文件中含有校验和检测，TaintScope分别对由PNG图片转换而来的SWF文件和由JPEG图片转换而来的SWF文件进行校验和分析。结果是，TaintScope发现Adobe Flash Player处理由PNG图片转换而来的SWF文件时，存在一个校验和检测点；在处理由JPEG图片转换而来的SWF文件时，TaintScope未发现校验和检测点。因此在模糊测试阶段TaintScope修改

了Adobe Flash Player的校验和检测功能。

在实际应用过程中，我们发现混合符号执行性能很慢。因此，我们并没有把收集到的路径约束逐一求反的方式生产新的测试例，而是仅检查路径上现有约束条件能否保证安全敏感操作的安全。我们使用微软Z3[177]求解器。我们设置一次求解的最长时间为5分钟。如果超过5分钟求解器仍然未能返回结果，TaintScope放弃这次检查，继续重放。

实验结果

以8个正常的SWF文件为初始样本，我们最初使用微软开发的MiniFuzz工具[117]对Adobe Flash Player进行了24小时的安全测试，但是没有发现任何安全漏洞。

接下来，TaintScope将8个正常SWF文件中的安全相关数据视为符号值，在对执行轨迹进行重放，进行混合符号执行推理。基于第5.3节中描述的测试例生成技术，TaintScope生成不同的畸形样本对Adobe Flash Player进行测试。

表 5.2: 针对Adobe Flash Player混合符号执行统计信息

Input Size	x86 Instrs	Symbolic Instrs	VEX Stmts	Constraints	Time(s)
51,485 (16)	95,771,481	4,318,044	29,739,150	357,414	9,678

表5.2是对8条执行轨迹的平均统计信息。第一列给出了正常SWF文件的平均大小，括号里数字是指有多少输入数据被替换为符号值；第二列给出了执行轨迹平均x86指令数，而第三列给出了需要符号执行的x86指令数目；由于TaintScope符号执行时，需要把x86指令翻译至VEX中间代码，第四列给出了相应的VEX语句数量。第五列给出了TaintScope在执行柜设计上收集到的路径约束个数。最后一列是TaintScope对每条轨迹重放所需时间。

对一条执行轨迹完成符号分析的时间大概要3小时；对8条轨迹依次重放的整体时间花销在24小时左右。但是基于混合符号执行生成的畸形样本，成功触发Adobe Flash Player解析`DefineBits`等标签结构代码中的3个零日安全漏洞。这些漏洞已经被Adobe确认和公开致谢。基于我们的漏洞报告，Adobe也迅速发布

了安全补丁[1]。鉴于Adobe Flash Player安全漏洞的影响力，我们不详细讨论漏洞的细节。

结果分析

- 越多的输入数据被视为符号值，符号执行越慢。在实验中，我们仅将能够影响内存分配的字节视为符号值。但是如果我们将所有输入数据都当做符号值时，TaintScope的重放速度越来越慢，内存消耗也越来越大，在分析百万条指令后就会因内存耗尽而退出。

- 符号执行缓慢的另一个原因在于SWF文件采用了编码文件格式。为了减少SWF文件的存储空间，SWF格式要求所有的整数类型都被压缩存储。一个4字节的整数会被编码为1-5字节（依赖于该整数的具体数值）。在SWF文件播放过程中，会使用大量的按位操作(例如`and`、`xor`)进行解码，这会引起符号表达式迅速扩大，进而影响符号执行的性能。

- 执行轨迹上很多约束都与符号值相关。平均每条执行轨迹上收集的路径约束条件有数万个。有两点原因引起这个现象。第一，这些符号值被用于循环边界条件。通常图像数据的宽度、高度等数据被替换为符号值，在逐行解析像素数据的循环中，这些符号值也经常被用于循环条件中。程序实际执行过程中循环会被执行很多次，因此重放过程中会收集到很多针对符号值的约束条件。第二，这些收集的路径约束条件中，存在非符号值的条件。在TaintScope当前的实现中，只要一个表达式含有一个符号因子，就会被当做符号表达式。但是，很多符号表达式经化简会，可能就是一个常数值。例如，`x+1-x`的简化结果就是常数1。Z3求解器提供了一个API `Z3_simplify`用于简化符号表达式。TaintScope之所以没有化简符号表达式，是由于化简过程时间代价非常大。TaintScope 仅简单的通过一个表达式是否含有符号因子而判读是否为符号表达式。因此，收集的路径约束中可能包含很多非符号值的约束。

[1] http://www.adobe.com/support/security/bulletins/apsb10-14.html

5.4.3 TaintScope系统漏洞发现能力

作为一个模糊测试平台，TaintScope已经Microsoft、Adobe、Google等著名IT企业的软件产品中发现了30个严重安全漏洞。表5.3总结了TaintScope的挖掘结果。

表 5.3: TaintScope平台挖掘的安全漏洞统计信息

Package	Vuln-Type	# Vulns	Checksum-aware?	Advisory	Rating
Microsoft Paint	Int-Overflow	1	N	CVE-2010-0028	Moderate
Google Picasa	Infinite loop	1	N	N/A	N/A
	Int-Overflow	1		SA38435	Moderate
Adobe Acrobat	Infinite loop	1	N	CVE-2009-2995	Extremely
	Int-Overflow	1	N	CVE-2009-2989	Extremely
Adobe Flash Player	Int-Overflow	3	N	CVE-2010-2170	Extremely
			Y	CVE-2010-2171	Extremely
ImageMagick	Int-Overflow	1	N	CVE-2009-1882	Moderate
CamlImage	Int-Overflow	3	**Y**	CVE-2009-2660	Moderate
LibTIFF	Int-Overflow	2	N	CVE-2009-2347	Moderate
wxWidgets	Buf-Overflow	2	N	CVE-2009-2369	Moderate
	Double Free	1	**Y**		
IrfanView	Int-Overflow	1	N	CVE-2009-2118	High
GStreamer	Int-Overflow	1	**Y**	CVE-2009-1932	Moderate
Dillo	Int-Overflow	1	**Y**	CVE-2009-2294	High
XEmacs	Int-Overflow	3	**Y**	CVE-2009-2688	Moderate
	Null-Deref	1	N	N/A	N/A
MPlayer	Null-Deref	2	N	N/A	N/A
PDFlib-lite	Int-Overflow	1	**Y**	SA35180	Moderate
Amaya	Int-Overflow	2	**Y**	SA34531	High
Winamp	Buf-Overflow	1	N	SA35126	High
Total		30			

表5.3第一列给出了测试程序的名称。测试程序涵盖了主要的应用程序，包括浏览器软件（Dillo、Amaya）、图形图像软件（Google Picasa、Microsoft Paint等）、音视频软件（MPlayer、Winamp等）、文本编辑软件（XEmacs）以及常用开发库（LibTIFF、wxWidgets等）。

第二列和第三列分别给出TaintScope在相应软件中发现的漏洞类型和数量。TaintScope发现的漏洞包括整数溢出（Int-Overflow）、缓冲区溢出（Buf-Overflow）、指针两次释放（Double Free）等。

第四列“Checksum-aware?”指明发现相应漏洞过程中，是否需要禁止目标程序的校验和检测功能以及对畸形样本修复校验和域。“N”表示目标程序没有对样本进行校验和检查，TaintScope直接根据导向性模糊测试或混合符号执行技术生成畸形样本。大约42%安全漏洞需要TaintScope处理校验和问题。

TaintScope在这些软件中共计发现了30个安全漏洞，均已经被开发方确认，或已经赋予国际漏洞库CVE、权威漏洞发布机构Secunia[142]的编号。第五列展示了漏洞收录信息，CVE-xx代表该漏洞的CVE编号，SAxx代表该漏洞的Secunia编号，N/A表示漏洞直接被开发方确认修复，并没有收录至公开漏洞库。第六列给出了Secunia组织对漏洞的安全程度的评级。Secunia把安全漏洞的严重程度分为Extremely、High、Moderate、Less和Not五个级别。TaintScope发现的安全漏洞中，基本都在Moderate及以上严重级。在Acorbat、FlashPlayer 等软件中发现的安全漏洞被评估为最高严重级（Extremely）。

在商业软件中发现了大量安全漏洞说明TaintScope具有很强的实用性。在Secunia的漏洞库中，收录了Google Picasa的3个安全漏洞，其中两个安全漏洞都是由TaintScope系统发现。我们发现Google Picasa安全漏洞后，第一时间联系了Google的安全团队。在提交了触发样本后，Google确认了安全漏洞。然而，处于未知原因，Google将软件升级修复了该漏洞后，并未将漏洞修复信息公布。因此，我们发现Google Picasa其他安全漏洞后，与国际权威漏洞发布机构Secunia合作，公布了漏洞信息，Google才公开承认Picasa产品存在问题。从此案例中也可以看出，出于商业推广和利益等原因，目前很多软件开发方并不愿意承认自身产品中存在问题，也不会与第三方安全人员合作提供自身产品的安全漏洞信息。

Microsoft Paint画图软件随Windows操作系统所有版本发布，已经有二十多年的历史。Microsoft Paint 的安全性非常出色。在我们发现Microsoft Paint漏洞之前，CVE和Secunia漏洞库中都没有Microsoft Paint的漏洞记录。目前为止，TaintScope是唯一在Microsoft Paint软件中发现安全漏洞的系统。微软已经为该漏洞发布了安全补丁并公开致谢[2]。

[2]http://www.microsoft.com/technet/security/bulletin/ms10-005.mspx

5.5 本章小结

本章介绍了TaintScope系统的畸形测试样本生成技术。与传统模糊测试技术盲目修改正常样本不同，TaintScope系统采用了反馈式畸形样本生成技术。特别地，TaintScope实现了基于细颗粒度污点分析的导向性测试例生成和基于混合符号执行的测试例生成技术。

基于细颗粒度污点分析的导向性测试例生成有效地降低了变异空间范围，提高了模糊测试的效率；作为导向性模糊测试的重要互补，基于混合符号执行的测试例生成技术能够对单条执行轨迹进行深度安全分析并生成高代码覆盖率的样本。此外，这两种样本生成技术都可以与第四章校验和感知技术相结合，提高TaintScope系统漏洞挖掘能力。

TaintScope系统对Microsoft、Google、Adobe等著名IT企业的知名软件产品进行了安全测试，发现了30个未公开安全漏洞。这些安全漏洞大部分已经被国际上权威漏洞收录机构或发布机构确认。根据我们的安全报告，软件厂商迅速发布了安全补丁。

第六章　二进制程序整数溢出漏洞检测技术研究

6.1　引言

前文主要介绍软件安全漏洞动态挖掘方法。与动态挖掘技术相比，静态挖掘方法具有不依赖于具体执行环境、能够对任意代码片段分析的优势，因此在软件安全性快速测评领域有着重要应用。

程序静态分析技术经过几十年的发展，形成了丰富的理论方法和技术手段。但是在处理漏洞挖掘问题时，特别是对于具有复杂数据依赖关系的漏洞类型时，静态分析虽然具有代码覆盖率高的优势，同时存在高误报率的劣势。采用路径敏感的分析技术可以提高分析的准确性，但是路劲敏感分析技术同时会导致路径爆炸问题。

现有静态漏洞挖掘技术高误报的重要原因之一在于这些技术并没有充分利用漏洞的特征。实际上，不同类型的安全漏洞具有不同的漏洞特征。如何利用漏洞特征提高静态漏洞挖掘的准确性是本章的研究重点。

针对整数溢出漏洞，本文在广泛调研真实漏洞的基础上，深入分析整数溢出的本质原因，提炼总结整数溢出漏洞模型；进而提出一种面向脆弱性包络的整数溢出漏洞检测技术，直接在二进制反编译的基础上，遍历代码执行空间，结合惰性检测策略，挖掘潜在的整数溢出漏洞。

本文实现了二进制程序整数溢出漏洞检测原型系统IntScope。IntScope对部分Windows系统动态链接库及多种应用程序的检测结果表明，IntScope不仅准确识别了已知的整数溢出漏洞，而且在多款流行软件(包括QEMU [33], Xine[18], Faad2 [12])中发现了20多个零日安全漏洞. 国际漏洞公布组织(CVE)、法国安全事件应急小组(FrSIRT)等机构为此先后发布了多项安全公告(如CVE-2008-4201, VUPEN/ADV-2008-2919),相关软件商也迅速发布了安全补丁。

本章内容安排如下：第二节介绍研究背景和主要相关工作；第三节介绍本文对整数溢出漏洞的建模分析；第四节讨论二进制程序中检测整数溢出漏洞所面临的挑战；第五节详细报告本文提出的整数溢出漏洞检测方法，讨论原型系统的设计与实现；最后，第六节报告相关实验结果，第七节对本章总结。

6.2 研究背景

6.2.1 整数溢出：被低估的安全威胁

高级语言(例如C/C++)的原语类型(例如`int`, `char`)在通常都有确定表达能力。例如C语言中32位`int`类型变量只能表达$[-2^{31}, 2^{31}-1]$ 范围内的整数。一旦程序中算术运算、移位运算的结果超出了相应类型的表达能力，就会引起“整数溢出”。文献[55]总结了C/C++语言中能够引起整数溢出的操作，如表6.1所示。虽然整数溢出并不会直接破坏系统安全属性，但是由整数溢出引发的其他安全漏洞则会给系统带来严重威胁。

表 6.1: 标准C/C++操作与整数溢出关系

Overflow Possible	Overflow Impossible
+ >>=	% unary +
- <<	= <
* >>	%= >
/ unary -	&= >=
++	|= <=
--	^= ==
+=	& !=
-=	| &&
*=	^ ||
/=	~ ?:
<<=	!

图6.1显示了2000-2010年间，美国国家漏洞数据库(NVD[14])收录的整数溢出安全漏洞的数量情况。由图6.1中可见，截止到2007年，整数溢出漏洞逐年迅速增

长；2007-2010年间，NVD每年收录的整数溢出漏洞趋于稳定。

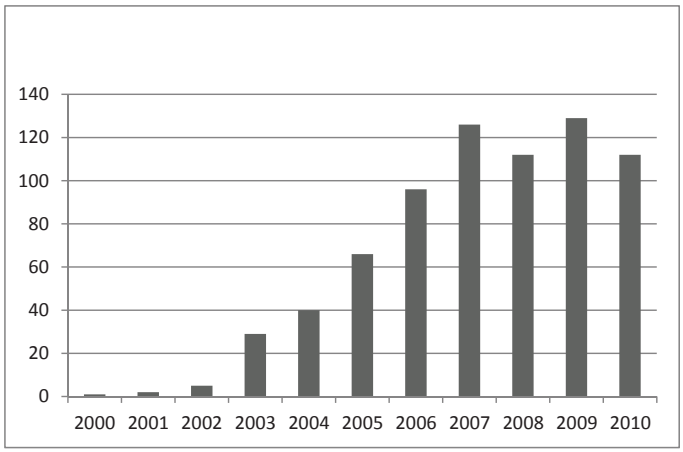


图 6.1: 整数溢出漏洞增长趋势

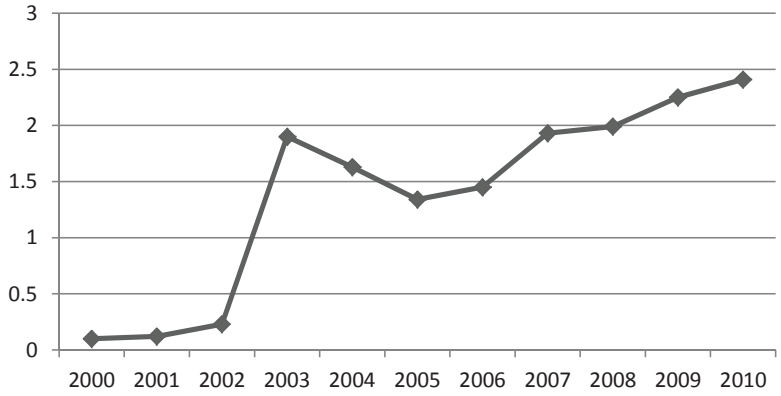


图 6.2: 整数溢出漏洞比重趋势

但是整数溢出漏洞占当年所有漏洞总数的比重在不断增长，如图6.2所示。2010年，整数溢出漏洞占NVD收录所有漏洞的2.41%[1]。虽然这个比重依然比较低，但是据CVE组织2007年对操作系统厂商发布的安全公告的漏洞类型的统计报告[11]，整数溢出漏洞已经在操作系统厂商安全公告中高居第二名。此外，根据

[1]数据来源：http://web.nvd.nist.gov/view/vuln/statistics

文献[178]的研究，我们发现很多整数溢出被统计为缓冲区溢出，一定程度上影响了整数溢出的总数。

6.2.2 主要相关工作

黑客社区[94, 169, 37]很早就开始关注整数溢出漏洞。近些年来，针对整数溢出漏洞的攻击越来越多，给信息系统安全带来了很多隐患。如何检测并排除整数溢出漏洞是研究的热点。

整数溢出主动防护的研究很多。传统编译器也提供了相应的功能。例如，GCC编译器在打开`ftrapv`选项下，会采用图6.3中代码替换所有的有符号整数类型加法操作。由于采用函数调用替换简单的算术运算，势必会造成程序性能严重下降。为降低这种性能损失，RICH[40]充分利用x86指令中检测标志寄存器的指令，例如`jo`、`jc`等，通过简单的指令组合检测整数溢出。类似地，IntPatch[178]在LLVM编译器基础上，对最容易发生整数溢出的操作进行防护，可以防止很多整数溢出问题。为防止整数溢出问题，一些研究人员还提出采用安全整数库（例如SafeInt）或任意精度整数类型（例如GMP、CLN）等替换原始的整数类型。但是，源代码级别的防护不可避免导致程序性能下降；更为严重的是，并非所有整数溢出都会导致安全问题。一旦程序中存在良性的整数溢出，上述方法就会产生误报。

```
96 __addvsi3 (SItype a, SItype b){
97
98   const SItype w = a + b;
100  if (b >= 0 ? w < a : w > a)
101   abort ();
102
103  return w;
104 }
```

图 6.3: GCC4.2.0中`__addvsi3`函数实现代码

在整数溢出漏洞检测的研究上，一些基于软件模型检验、数据流分析或类型系统推导理论的工具（例如Prefix[139], RangeChecker[26]等）通过分析程序

源代码、匹配代码中不安全的整数使用模式，可以检测部分整数溢出漏洞。E. Ceesay等[51]在类型系统Cqual基础上，实现了对不可信整数类型的追踪，检测对不可信整数的滥用。D. Sarkar等[139]实现了流不敏感（flow-insensitive ）的检测方法，在净化约束图上追踪未被净化的整数变量的使用方式。

然而，当没有源代码时，整数溢出漏洞的检测主要依靠模糊测试。虽然一些模糊测试系统(如SAGE[91], SmartFuzz[120], IntHunter[1])能够利用程序内部信息，根据程序的一次执行生成更多测试例，但是这些系统仅能对程序部分执行空间分析，代码覆盖率有限，而且受限于初始样本的选择。

与本文工作比较类似是UQBTng[169]。该工具整合了反编译工具UQBT[16]和模型检验器CBMC[104]。在UQBT把目标二进制文件反编译至C代码后，UQBTng进一步在内存分配函数前插入适当的`assert`类型语句，并利用CBMC检测`assert`语句是否会被触发。然而，把二进制程序反编译至C代码是一个巨大的挑战，UQBT仅仅在该领域上作出了一些尝试。UQBTng的检测能力不仅受限于反编译前端，还受限于CBMC的分析能力。

6.3　整数溢出漏洞建模

基于对200多个真实整数溢出漏洞实例分析，我们发现绝大部分整数溢出漏洞都是由于程序没有对外界可控数据(下文称作污点数据)进行正确的范围检查。精心设计的外界可控数据在传播过程中可以进一步导致整数溢出。虽然整数溢出不会对程序执行逻辑造成直接破坏，一旦溢出结果被用于内存分配、内存访问、谓词语句，就会造成堆溢出、栈溢出等严重安全漏洞。

含有整数溢出漏洞的程序路径具有以下特征:

- **调用污点数据引入函数**：发生整数溢出操作中，部分操作数和污点数据存在数据依赖关系，而这些污点数据来源于外界可控输入源，例如文件、网络报文、命令行参数等，并且由专门的API函数引入。典型的污点数据引入函数包括`read`、`fread` 、`recv`。

- **对污点数据不恰当检查**：程序路径上没有检查污点数的范围或者范围检查不完备。例如，图6.4是CUPS软件中整数溢出漏洞(CVE-2008-1722)的部分代

码. 虽然程序对污点数据width和height进行了范围检查，但是该范围检查不正确，使得污点数据传播后，image->xsize * image->ysize仍然可能溢出。

```
1. if (width == 0 || width > CUPS_IMAGE_MAX_WIDTH ||
2.     height == 0 || height > CUPS_IMAGE_MAX_HEIGHT)
3.  return (1);
4. img->xsize = width;
5. img->ysize = height;
6. ……
7. if (color_type == PNG_COLOR_TYPE_GRAY ||color_type ==
8.     PNG_COLOR_TYPE_GRAY_ALPHA)
9.  in = malloc(img->xsize * img->ysize);
```

图 6.4: CVE-2008-1722整数溢出漏洞代码片段.

- **将溢出结果用于敏感操作**：并非所有的整数溢出都会导致安全问题。只有当溢出结果影响了程序的控制流、内存分配、内存访问等操作时，才会进一步引起其他安全问题。在已知的整数溢出漏洞中，溢出结果对程序的影响分以下几类：

  1. 内存分配：溢出值被用于内存分配函数(例如malloc函数)的参数，这通常会导致分配小于预期配额的内存，对该段内存的访问导致缓冲区溢出；
  2. 内存访问：溢出值被用于内存访问(例如作为数组访问的index、指针偏移),精心构造的溢出结果可能会导致任意内存单元的读写；
  3. 分支语句：一旦溢出值被用于分支语句，而该分支语句不是用于捕获整数溢出错误，那么溢出值会破坏程序固有逻辑，造成程序的非预期执行；
  4. 其他程序相关的敏感点：溢出值可能引起某些程序相关的错误，例如，NetBSD 整数溢出漏洞(CVE-2002-1490)的原因是溢出值被用于共享内存的引用计数，发生溢出时，导致共享内存被提前释放；

本文将整数溢出漏洞视为一种特殊的source-sink问题[141]，如图6.5所示。存在整数溢出漏洞的路径上，首先应该调用污点数据引入函数（即图6.5中的Source节点）。常见的污点源包括`fread`、`recv`等。其次，这些路径上存在对溢出结果敏感的操作（即图6.5中的Sink节点），例如内存分配函数`malloc`等。最后，这条路径上的约束条件是自洽、可满足的，即该路径是一条可行路径。此外，路径约束并没有对污点数据做到充分检查，路径上仍然存在导致整数溢出的运算。

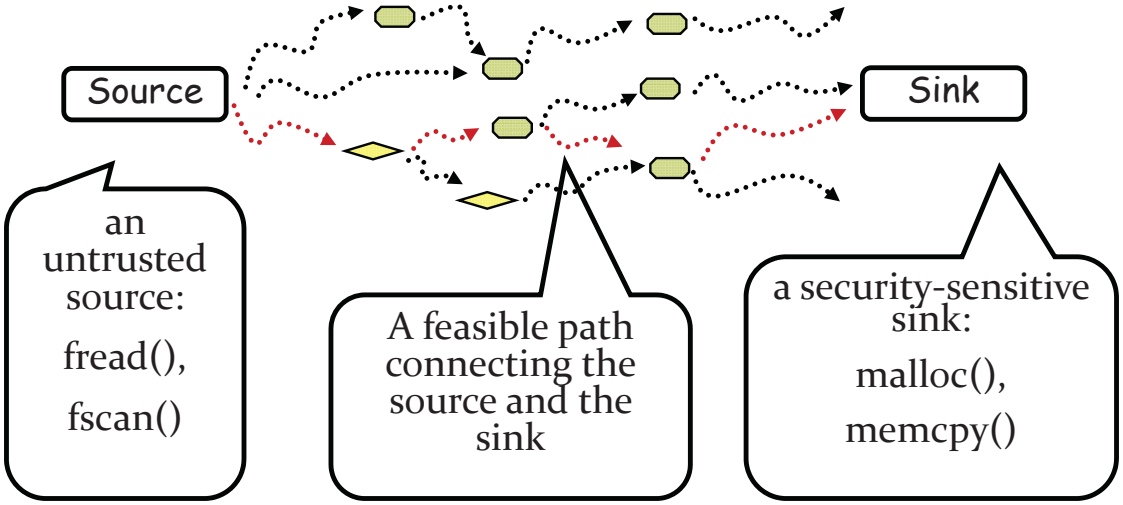


图 6.5: 整数溢出漏洞模型

6.4　二进制程序中整数溢出漏洞检测面对的困难

与在源代码中检测整数漏洞相比，检测二进制程序中整数溢出漏洞面临更多困难。

- 二进制代码中控制结构非常复杂，非结构化特征明显，给二进制代码分析带来很多不便。X86指令系统复杂，指令副作用多，直接分析x86指令事倍功半[43]。
- 二进制程序中损失了很多源程序中变量的类型信息，而整数溢出检测与整数类型密切相关。同样32-bit的内存单元，在有符号整型(`int`)和无符号整

型(unsigned int) 的解释下可能会有完全不同的取值。二进制指令操作对象是寄存器和内存单元，无法直接获得类型信息。例如，这段代码中，

```
mov eax, 0xffffffff;//eax = 0xffffffff
add eax, 2;         //eax = 0xffffffff+0x2
```

我们不能判断这段代码中是否存在加法溢出：如果0xffffffff解释为有符号数-1，这属于正常的加法运算；如果0xffffffff解释为无符号数，这段代码发生加法溢出。

- 更为严重的是，二进制程序无害的整数溢出非常普遍。由于整数溢出本身不构成对程序的破坏，程序员甚至编译器会故意利用整数溢出溢出。例如，GCC编译器在O2优化选项下，会把源代码if(x >= -2 && x<= 0x7ffffffd)编译生成:

```
mov eax, x;  // eax = x
add eax, 2;  // eax = eax+2
js target
```

这段二进制代码中，将源程序中两个谓词判断简化为一个谓词判断，虽然极大的x会使add指令会发生溢出，GCC正是利用了溢出的性质，简化了谓词判断。检测过程中，不能将这种对程序不造成破坏的良性溢出识别为整数溢出漏洞。

- 二进制程序中函数调用关系不明确。函数指针、虚函数等高级语言特性，经编译以后，在二进制程序层就表现为间接函数调用。例如，call eax语句的调用地址是运行时决定的，静态分析过程中获取这些函数调用关系也非常困难。

- 为降低误报率，路径敏感分析技术不可或缺。因为大部分整数溢出漏洞是由于程序对污点数据检查的不完备，为了准确识别整数溢出漏洞，需要路径敏感的程序分析方法以判断路径上的约束关系能否避免整数溢出。在真实软件中，程序路径数目随着程序分支语句数目指数增长，细颗粒度的路径敏感分

析不可实现。如何降低遍历空间规模是路径敏感程序分析方法的关键问题。

```
int foo(int x) {
    if (x>=0 && x+x>=0){
      use(x+x);
    }
}
```

图 6.6: 源代码层进行了整数溢出检测

直接在二进制层进行漏洞挖掘也有一个重要优势：能够发现在源代码成无法发现的一些安全漏洞。源代码层的代码安全并不意味着二进制程序的绝对安全，由于编译器优化错误或差异，经常会导致源代码和二进制代码并不一致。例如，图6.6中展示的代码里，函数`foo`只有在“`x>=0 && x+x>=0`”成立时才会使用`x+x`。这里的条件判断语句主要目的就是为了防止`x+x`溢出。

然而，由于C语言标准中将有符号整数类型的溢出定义为未定义行为（undefined behavior），GCC编译器在优化选项下会将`x+x>=0`的判断优化掉，因而在二进制程序中只有对应于`x>=0`的判断。程序员群体和GCC编译器开发方就此特性展开了激烈的辩论，目前为止GCC仍然保留该特性[2]。Python语言实现库就是一个真实的案例：源代码层是没有整数溢出问题的代码，由于GCC的优化，二进制代码层反而存在整数溢出漏洞[3]。

6.5　面向脆弱性包络的整合溢出漏洞挖掘技术

6.5.1　系统概览

IntScope系统架构如图6.7所示。IntScope以BESTAR[162][2]反编译器为前端，将二进制程序反编译到PANDA中间表达后，在中间代码层进行分析。BESTAR的反编译结果中，已经包含了程序函数调用关系[164]和过程间控制流关系。

为了避免空间爆炸问题，IntScope的Pre-pruning模块将通过脆弱性包络提取方法，自动识别执行空间中最容易发生整数溢出漏洞代码部分。在符号执行阶

[2]http://gcc.gnu.org/bugzilla/show bug.cgi?id=30475#c2

[3]http://bugs.python.org/issue1608

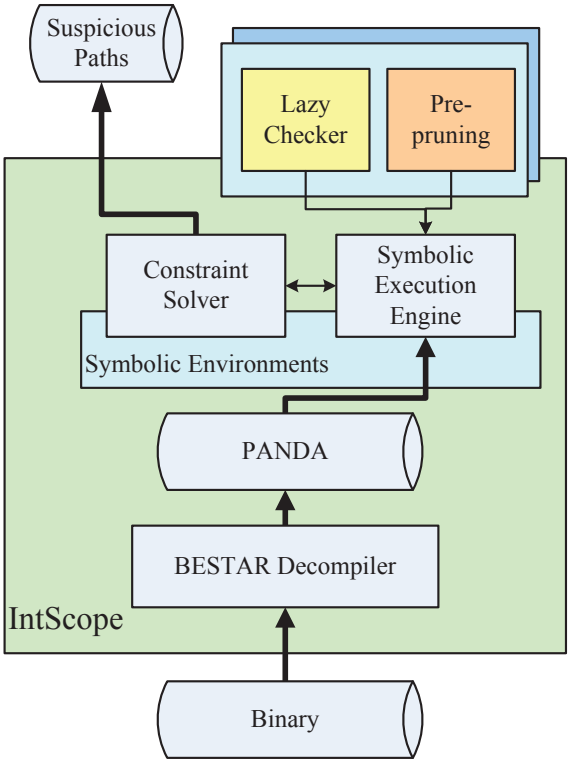


图 6.7: IntScope架构

段，IntScope维护符号化内存空间，精确模拟脆弱性包络中每条程序路径的执行，并通过约束求解器判断路径的可行性。为了克服二进制程序中整数溢出检测的困难，IntScope采用惰性检测策略（lazy check）检测可能的整数溢出漏洞。一旦IntScope发现污点数据可能引起计算溢出，而且溢出结果又被用于敏感操作，IntScope报告一个潜在的整数溢出漏洞。

6.5.2 脆弱性包络提取

为了缓解路径爆炸问题，本文根据第6.3节所总结的整数溢出特征，自动识别二进制程序中脆弱性包络(Vulnerable Component)，即最容易发生整数溢出漏洞的代码。本文中，脆弱性包络是一种程序函数间控制流图的子图结构$G = (N, E, h, t)$，这里N代表控制流图上节点集合，E代表控制流图上边的集合，h和$t \in N$分别代表子图入口节点和出口节点；对于给定的污点数据引入函数集$Source$和溢出值敏感函数集合$Sinks$，图G上任意一条从h到t的路径都会调

用$Source$和$Sinks$中函数。具体而言，识别整数溢出漏洞的脆弱性包络由以下两步完成：

1. 整数溢出漏洞的基本特征之一是，在程序的路径上调用污点数据引入函数(如`fread`)后又调用了某些溢出值敏感函数(如`malloc`). 我们定义集合$Source$为污点数据引入函数集，集合$Sinks$为溢出值敏感函数集合。我们根据程序中函数调用关系建立函数调用图: $C = (N, E)$，其中，N代表节点集合，任意$n_i \in N$表示程序函数；E是边集合, 每条边$(n_i, n_j) \in E$表示函数n_i直接调用了函数n_j。通过遍历函数调用图，我们选取既可以直节或间接调用$Source$中函数也直接或间接调用$Sinks$中函数的节点作为脆弱性包络的入口，即寻找$Source$中函数和$Sinks$中函数的公共前驱节点。

   例如，图6.8中，`read_and_malloc`既可以间接调用`fread`，也可以间接调用`malloc`，通过遍历该函数调用图，我们选取`read_and_malloc`作为脆弱性包络入口函数。

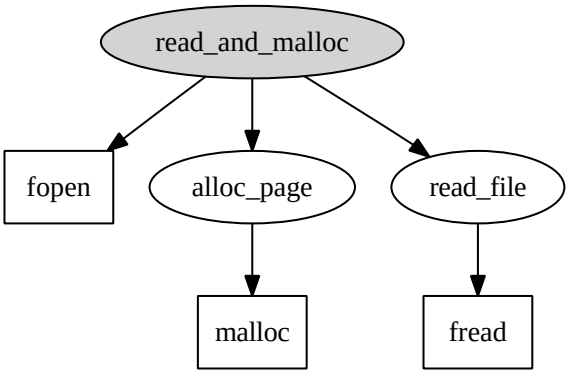


图 6.8: 函数调用图示例

2. 对每一个脆弱性包络入口函数，我们进一步裁剪函数的过程间控制流图，使裁剪后的程序控制流图上，仅包含既调用了$Source$中函数也调用了$Sinks$中函数的程序路径。

   令n_{entry}代表入口函数的入口节点，n_{exit}代表入口函数的退出节点, Sr代表所有直接调用$Source$中函数的节点，程序路径片段集合$\mathcal{P}(n_i, n_j)$代表所有

从n_i到n_j的路径上节点的集合，Sk代表所有直接调用$Sinks$中函数的节点，具体切片算法如下:

(a) 遍历函数过程间控制流图，计算集合$Esr = \{n_i \mid \forall n_j \in Sr, n_i \in \mathcal{P}(n_{entry}, n_j)\}$;

(b) 遍历函数过程间控制流图，计算集合$Esk = \{n_i \mid \forall n_j \in Sk, n_i \in \mathcal{P}(n_{entry}, n_j)\}$;

(c) 遍历函数过程间控制流图，计算集合$Se = \{n_i \mid \forall n_j \in Sr, n_i \in \mathcal{P}(n_j, n_{exit})\}$;

(d) 如果$Se \bigcap Esk \neq \phi$, 裁剪函数间控制流图，使新控制流图只包含$Esr \bigcup (Se \bigcap Esk)$中节点；否则，新控制流图中不含任何节点。

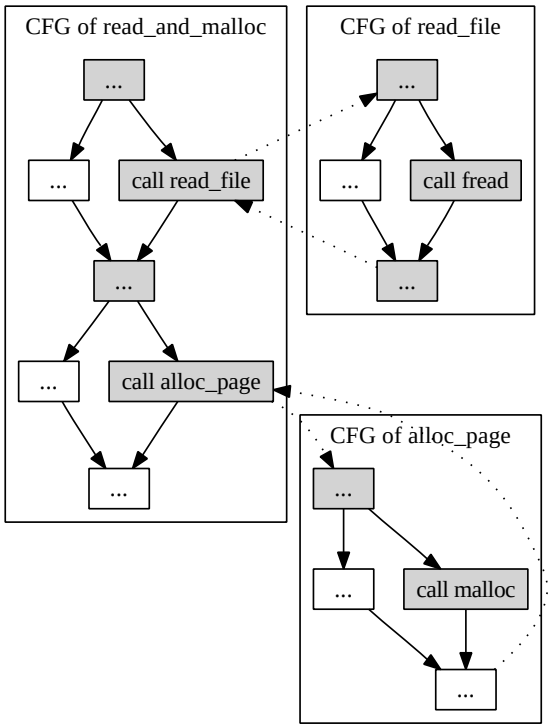


图 6.9: 控制流图示例

如图6.9中所示三个函数控制流图构成了以`read_and_malloc` 为入口函数的函数间控制流图，阴影部分为裁剪后的脆弱性包络。

经过上述步骤后，每个脆弱性包络中，仅包含即调用污点引入函数有调用溢出值敏感函数的路径，显著地降低了需要遍历的代码空间范围。IntScope会进一步对这些脆弱性包络进行在分析。

6.5.3　符号执行

在第6.5.2节工作的基础上，IntScope通过符号执行技术进一步模拟脆弱性包络中每条程序路径的执行，判断路径的可行性以及是否存在整数溢出问题。

图6.10展示了PANDA中间代码的部分语法规则[159][2]。IntScope在PANDA中间代码层进行静态符号执行分析。具体而言，IntScope维护一个符号环境，该环境是符号化地址与符号值的映射。所有的外界读入的数据都被当作符号值。对于路径上的每条中间代码，根据语句的语意，IntScope更新符号环境。对于分支指令，IntScope会根据可满足性模理论决策程序判断路径的可行性，IntScope以深度优先遍历方式只分析可行路径。

$$
\begin{aligned}
&\textit{statement} ::= \textit{var} :: \tau[e] \mid \textit{val} := \textit{var} \mid \textit{var} = e \mid \text{if}\,(e \Delta_b e)\ \text{then}\ l_1\ \text{else}\ l_2 \mid \\
&\qquad \text{goto}\ l_1 \mid \textit{var} = \textit{function}(e, e, ...) \mid \text{return} \\
&e ::= \text{num} \mid \textit{val} \mid e \Delta_{op} e \\
&\tau ::= \text{qword} \mid \text{dword} \mid \text{word} \mid \text{byte} \\
&\Delta_b ::= ==, !=, <_s, \leq_s, >_s, \geq_s, <_u, \leq_u, >_u, \geq_u \\
&\Delta_{op} ::= +,\ -,\ *,\ /,\ \&,\ |,\ \oplus,\ <<,\ >> \\
&\textit{val}, \textit{var}, \textit{function} ::= \text{string (value,variable and function's name)}
\end{aligned}
$$

图 6.10: PANDA中间代码语法规则

在符号表达式的设计上，IntScope采用了开源符号计算平台GiNaC[13]。GiNac是一个C++库，提供了灵活的表达式化简和规范化处理能力。然而，GiNaC符号运算库中并不支持比特矢量符号类型。换言之，在GiNaC环境中，一个整数变量x的取值范围是$[-\infty, \infty]$；然而在计算机运算环境中，一个32bit的整形变量有着明确的取值范围。整数溢出问题正是由于整形变量的取值超过了改类型所描述范围导致的。

为了克服GiNaC的不足，IntScope记录了每个GiNaC符号表达式的比特长度，并实现了GiNaC表达式到矢量约束求解器STP[85]的转换。也是就是，IntScope采用GiNaC构建符号表达式，记录符号运算关系和收集路径约束信息；然后将这些符号信息转换到STP接收的形式，由矢量求解器判断路径的可行性和是否会发生整数溢出。

循环结构是静态符号执行的一个难题。对于常数边界的循环，如果循环变量在每执行一次循环后都跟接近于常数边界，IntScope尽量精确模拟该循环；对于符号边界循环，IntScope只遍历该循环体一次。

在符号执行的同时，IntScope使每个符号值都绑定一个污点属性，跟踪污点数据的扩散。程序调用污点数据引入函数时，Intscope标记相应的数据为taint；随着程序的执行，污点数据属性通过数据依赖关系传播。对于无法精确模拟的API函数，IntScope仅跟踪数据污点属性的传播，忽略符号值的约束关系。

6.5.4 惰性检查策略

IntScope采用惰性检查策略检查程序中是否存在整数溢出漏洞。由于整数溢出漏洞是因为由污点数据引起计算溢出，溢出结果又被用于敏感操作，所以惰性检查的核心思想是: 只有当具有污点属性的值应用于敏感操作时，再检查是否会发生溢出。

惰性检测策略有两个主要优势。首先，惰性检测策略避免了对所有算术运算都进行整数溢出检测。二进制程序中存在大量的算术运算，栈帧变化、地址寻址等都涉及到算术运算。如果对每个运算都检查是否会发生整数溢出，必然会降低分析的效率。

```
1   mov     eax, x
2   shl     eax, 2
3   add     eax, 4
4   push    eax
5   call    malloc  // malloc(x*4+4)
```

图 6.11: 惰性检查示例

与之相反，IntScope并不会检测所有运算指令是否会溢出，而是在溢出值敏感操作点检查是否会有溢出。例如，图6.11中x代表一个符号值；在对这段指令序列的符号分析过程中，IntScope并不会依次检测shl和add指令运算过程中发生整数溢出，而是先收集符号运算关系；直至遇到调用malloc时，IntScope会发现传入malloc的内存配额为x*4+4；由于malloc 是一个溢出值敏感点，IntScope会检查当前路径约束能否防止x*4+4溢出。

其次，惰性检测策略可以充分利用恢复出来的类型信息。常见的溢出值敏感点都提供了一定的类型信息。例如，对于内存分配函数，如malloc等，都接受无符号整数作为内存配额参数。X86的条件跳转指令分有符号跳转和无符号跳转两类，其中JG、JNLE、JGE、JNL、JNGE、JLE、JNG、JE 和JNE都是有符号跳转指令，意味比较指令(如cmp)的操作被解释为有符号数；JA、JNBE、JAE、JNB、JB、JNAE、JBE、JNA、JE和JNE 是无符号跳转指令，意味着比较操作数被解释为无符号数。

6.6 实验结果

6.6.1 实验设置

实验部分，我们将对IntScope的有效性和性能进行评测。所有实验在运行Linux Kernel 2.6.18的服务器(CPU：AMD Opteron 2.6 GHz，内存：8GB) 上进行。为了验证系统的有效性，我们首先选择Windows操作系统动态链接库Gdi32.dll (V5.1) 和Comctl32.dll (V5.82)，用IntScope在这两个二进制库中查找已知的整数溢出漏洞。进一步，我们对多款流行应用程序分析，使用IntScope挖掘这些应用程序的整数溢出问题。本节最后报告IntScope的性能分析结果。

6.6.2 已知漏洞分析

Comctl32.dll中DSA_SetItem函数中存在一个整数溢出漏洞[42]。DSA_SetItem函数用于设置动态结构体数组的指定元素。为更好理解这个漏洞，我们在图6.12中给出了这个函数的伪代码。DSA_SetItem函数有三个参数。第一个参数hdsa是指向动态结构体数组的指针；第二个参数index是待修改的元素索引；最后一个参数pItem是指向结构体元素指针，将用于覆盖index索引的元素。

```
int DSA_SetItem(HDSA hdsa, int index, void *pItem) {
    HLOCAL hMem;
    int nNewItems;
L1 if (index < 0)  return 0;
L2 if (index >=hdsa->nItemCount) {
L3   if(index + 1>hdsa->nMaxCount)
     {  nNewItems=((index + hdsa->cItemGrow)/
                    hdsa->cItemGrow)* hdsa->cItemGrow;
        hMem = ReAlloc(hdsa->hMemArrayData,
                nNewItems * hdsa->nItemSize);
        //ignore some statements here
     }
L4   hdsa->nItemCount = index + 1;
   }
L5  memmove((void*)((index*hdsa->nItemSize)+
          (DWORD)hdsa->hMemArrayData),pItem, hdsa->nItemSize);
```

图 6.12: DSA_SetItem伪代码

图6.12中L1处的检查语句首先会判断index是否为负数；进一步当index超过了hdsa中已有元素个数nMaxCount时，代码重新计算hdsa中动态结构体元素需要的内存空间大小，并调用ReAlloc函数重新分配内存。在L5处，代码调用memmove函数，用pItem所指向的结构体单元覆盖hdsa中index索引的元素。

虽然代码多次对index进行检查，过大的index以及结构体元素占用空间较多时，ReAlloc函数的配额参数“nNewItems * hdsa->nItemSize”就会整数溢出，导致ReAlloc分配少于预期的内存空间，后继调用memmove时，就会发生缓冲区溢出。

图6.13左边是IDA Pro[95]对DSA_SetItem函数的反汇编结果，与之对比，图6.13右边是IntScope对该函数反编译至中间代码的结果。由于DSA_SetItem函数是一个公开API函数，IntScope将其参数视为符号值。

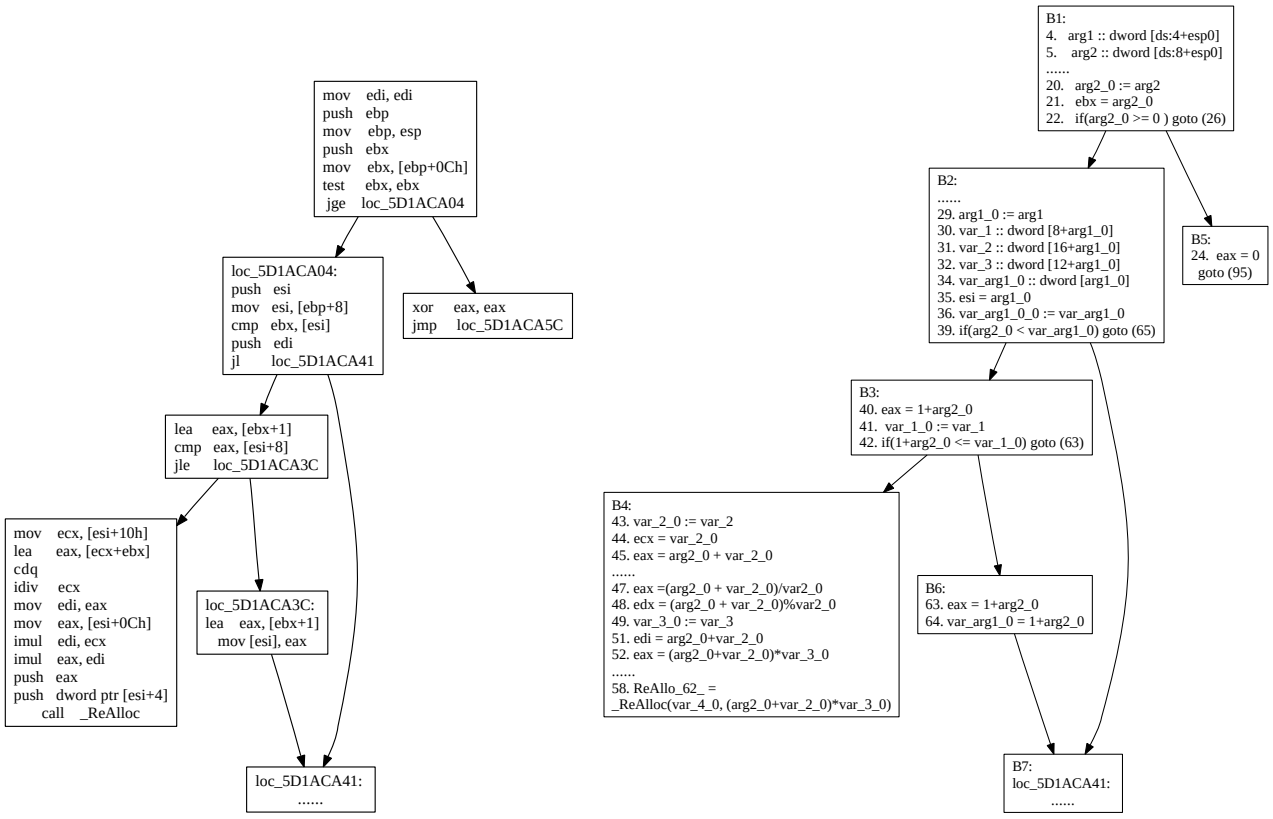


图 6.13: DSA_SetItem二进制反汇编结果与反编译PANDA结果

在对DSA_SetItem函数的分析过程中，IntScope首先准确定位了ReAlloc参数整数溢出问题。在路径< $B1, B2, B3, B4$ >上，IntScope收集到的路径约束为：arg2_0>=0 && arg2_0>=var_arg1_0 && (1+arg2_0)>var_1_0，而ReAlloc分配内存大小为(arg2_0+var_2_0)*var_3_0。该路径上的约束条件并不能防止整数溢出漏洞，而ReAlloc 是对溢出值敏感的API，因此IntScope将该路径作为疑似漏洞路径输出。

除了这条路径以外，IntScope将对应于图6.12中<L1，L2，L5>和<L1，L2，L4，L5>的两条路径作为疑似漏洞路径，原因是在memmove函数中存在index*hdsa->nItemSize乘法溢出。虽然代码中对index做了检查，由于缺乏对每个结构体元素的大小的先验约束，IntScope认为上述乘法计算依然存在潜在溢出问题。

类似地，IntScope在Gdi32.dll中准确检测到了整数溢出漏洞CVE-2007-3034，这里不再赘述。此外，IntScope在Gdi32.dll发现另外一处疑似整数溢出漏洞。

IntScope发现一个名为pmetalink16Resize的函数调用了内存分配函数LocalReAlloc，传入的内存大小参数为20+4*arg2_0。IntScope发现路径约束并不能防止20+4*arg2_0发生溢出。然而，基于IntScope的分析结果，我们发现符号值arg2_0来源于Gdi32.dll的一个全局数组。由于缺乏运行时信息，IntScope并不能判定该全局数组是否完全受外界控制，也不确定全局数组的内容是否有其他限制，故不能确定pmetalink16Resize函数是否真的存在整数溢出问题。IntScope将这些疑似点作为疑似安全漏洞待以后进一步分析。

6.6.3 零日漏洞分析

IntScope在对几款流行的应用软件检测中，发现了20个未知的整数溢出漏洞，均被软件开发方或具体触发样本证实。国际漏洞公布组织(CVE)、法国安全事件应急小组(FrSIRT)等机构为此先后发布了多项安全公告(如CVE-2008-4201,VUPEN/ADV-2008-2919),相关软件商也迅速发布了安全补丁。

表 6.2: IntScope漏洞挖掘结果

Name	Version	Entry Function	Paths#	Total#	Confirmed #	Suspicious#
GDI32.dll	5.1.2600.2180	CopyMetaFile	452	3	1	2
comctl32.dll	5.82.2900.2180	DSA_SetItem	3	2	1	1
QEMU Xen	0.9.1 3.2.1	bochs_open	3	1	1	0
		cloop_open	1	1	1	0
		parallels_open	2	1	1	0
		qcow_open(for qcow2 format)	3	1	1	0
		vmdk_open	20	2	1	1
		vpc_open	1	1	1	0
Xine	1.1.15	ff_audio_decode_data	10	1	1	0
		process_commands	2	2	2	0
Xine-ui	0.99.5	_LoadPNG	4	1	1	0
MPlayer	1.0rc2	dumpsub_gab2	1	1	1	0
		init_registry	3	1	1	0
Mpd	0.13.2	mp4_decode	2	1	1	0
Goom	2k4	gsl_read_file	1	1	1	0
Cximage	600_full	ConvertWmfFiletoEmf	1	1	1	0
faad2	2.6.1	decodeMP4file	36	3	2	1
		mp4ff_read_stts	1	1	1	0
hamsterdb	1.0.4	btree_find_cursor	3	1	1	0

具体实验结果如表6.2所示。第一列是应用程序名称，第二列给出了版本信息；第三列“Entry Function” 指的是IntScope将这些函数作为遍历的入口，也就是说这些函数是脆弱性包络的入口；第四列给出IntScope发现的疑似漏洞路径的总数，而第五列报告了IntScope发现的漏洞总数。由于IntScope采用路径敏感的分析方式，很多疑似漏洞路径都终止于同一个脆弱点。因此疑似漏洞路径的总数远多于漏洞总数。但由于缺乏运行时信息，IntScope并不能完全确认这些漏洞的真实性。第六列给出已经确认为真实漏洞的数目，最后一列是尚未确认的疑似整数溢出漏洞的数目。

Qemu[33]和Xen[15]虚拟机工具已经成为了云计算的基础平台。虚拟机工具的安全性直接关系到上次云计算平台的安全性。IntScope对Qemu和Xen进行了检测，发现了多个零日安全漏洞。法国计算机安全响应组织（French Security Incident Response Team，现已更名为VUPEN）已经为这些漏洞发布了安全公告VUPEN/ADV-2008-2919。

Qemu支持很多种硬盘镜像格式，包括`qcow`、`vmdk`和`raw`等。IntScope发现Qemu在解析磁盘镜像格式时存在多处整数溢出漏洞，给基于Qemu的云计算平台带来严重隐患。图6.14给出了IntScope在`qcow_open`函数中发现的整数溢出漏洞的代码片段。值得说明的是，这里给出漏洞的源代码主要是为了解释该漏洞成因；而IntScope是直接分析二进制文件发现的这些安全问题。

图6.14第194行的`s->hd`是磁盘镜像文件句柄。函数`qcow_open`从镜像文件中读入文件头值`header`中。因此，IntScope将`header`中所有数据内容当做符号值。在第246行，函数`qcow_open`对`header`中数据进行了检查，但是这些检查不能防止后第249行中`qemu_malloc`参数的乘法溢出。一旦发生整数溢出，就到导致`qemu_malloc`仅分配很少的内存；后继对分配的内存写操作过程中就会发生缓冲区溢出。

IntScope在Qemu解析各种磁盘镜像文件的代码中发现了多个安全漏洞。IntScope在对Xen的检查过程中发现了完全一样的安全漏洞。经过进一步分析，我们发现Xen重用了Qemu的部分代码，因此Qemu中的整数溢出漏洞也同样存在于Xen平台中。

```
194   if (bdrv_pread(s->hd, 0, &header, sizeof(header))
            != sizeof(header))
195       goto fail;
      ......
241    s->l1_size = header.l1_size;
246    if (s->l1_size < s->l1_vm_state_index)
247        goto fail;
249    s->l1_table=qemu_malloc(s->l1_size*sizeof(uint64_t));
250    if (!s->l1_table)
251        goto fail;
252    if (bdrv_pread(s->hd, s->l1_table_offset,
            s->l1_table,s->l1_size * sizeof(uint64_t)) !=
253         s->l1_size * sizeof(uint64_t))
254        goto fail;
255    for(i = 0;i < s->l1_size; i++) {
256        be64_to_cpus(&s->l1_table[i]);
257    }
```

图 6.14: qcow_open整数溢出漏洞代码片段(qemu-0.9.1/block-qcow2.c)

表6.2中给出了IntScope对其他流行应用程序的检测结果。对每个漏洞的分析结果参见文献[159]。总体而言，IntScope在这些软件中发现了26个疑似整数溢出漏洞，其中20个已经被手工分析或软件开发方确认。其余6个疑似安全漏洞处于尚待确认过程中。

6.6.4 系统性能分析

为说明全路径遍历的不可行性，我们在图6.15中报告IntScope从Qemu程序的`main`函数开始遍历所有可行路径的性能信息。

IntScope以深度优先的模式遍历代码执行空间。图6.15的横轴代表经过IntScope分析的分支基本块的数目，纵轴代表IntScope完成遍历所用时间。由图中可见，IntScope遍历时间随着分支基本块数的增长，遍历时间呈近指数增长趋势。分支基本块的数目在一定程度上可以反映程序路径的数目。最坏情况下，程序路径数

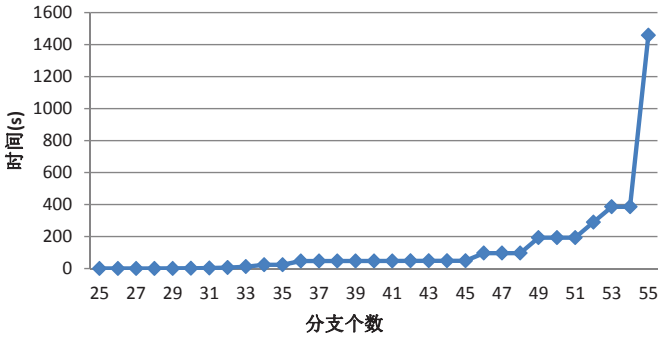


图 6.15: IntScope全路径遍历性能

与分支基本块数目是指数关系。因此，尝试以路径敏感的方式对程序代码空间穷尽遍历是不可行的。

表 6.3: IntScope系统性能测评结果

Name	Executable	File Size	Binary-to-IR time (s)	IR Size	Traversing Time (s)
GDI32.dll	GDI32.dll	271KB	614	7.61 MB	574
comctl32.dll	comctl32.dll	597 KB	1131	13.7 MB	0.1
QEMU	Qemu-img	341 KB	124	12.8 MB	358
Xine	cdda_server	14.5 KB	4	116 KB	26
	xine	966 KB	590	12.9 MB	327
Mplayer	avisubdump	14.2 KB	1	36.8 KB	0.3
MPD	mpd	243 KB	131	2.74 MB	667
GOOM	libgoom2.so	439KB	94	1.42 MB	445
faad2	faad	57.6 KB	29	693 KB	113
hamsterdb	libhamsterdb.so	260 KB	164	3.46 MB	426
Average		320.3KB	288.2	5.46MB	293.6

但是，由于提取脆弱性包络可以有效避免路径爆炸问题，IntScope可以短时间内对脆弱代码空间遍历。表6.3的前两列给出了软件包以及具体的二进制文件信息。第三列是二进制文件的大小。IntScope需要将二进制文件反编译至中间代码后进一步分析，第四列给出了反编译的时间花费。由于IntScope的反编译前端需要对结果进行控制流分析、静态单一赋值分析等，该阶段较为耗时。反编译过程输出文件即PANDA中间代码。`IR Size`列给出了反编译输出文件的大小。反编译输出文件通常远大于原始二进制文件。主要原因在于，一条二进制指令，通常会被翻译至多条中间代码语句。由于IntScope仅对脆弱性包络进行分析，所以通常

仅需要几分钟即可完成遍历。

6.7 本章小结

传统的基于静态分析技术的漏洞检测方法存在高误报率等问题。虽然随着符号执行技术的再次兴起，很多系统采用了路径敏感的分析方法，一定程度上降低了分析的误报率，但是路径敏感分析又面临执行路径爆炸问题。

为降低静态漏洞挖掘技术的误报率，同时缓解执行路径爆炸问题，本章以整数溢出漏洞为对象，重点研究如何结合漏洞特征指导静态检测。整数溢出漏洞近些年来增长迅速，危害严重，对整数溢出漏洞的检测和防护已经成为研究热点。

基于对大量真实整数溢出漏洞深入分析的基础上，本文将整数溢出漏洞抽象为一个source-sink模型。在该模型指导下，提出程序脆弱性包络提取方法；进而，通过静态符号执行技术遍历脆弱性包络，避免了对整个代码空间遍历，有效缓解了路径爆炸问题。

本文实现了面向二进制程序的整数溢出漏洞检测系统IntScope。IntScope在对Windows操作系统动态链接库分析中，准确检测到了已公开的整数溢出漏洞。在对包括虚拟机平台Qemu、Xen等重要应用程序的分析过程中，IntScope已经检测到20个零日安全漏洞。国际漏洞公布组织(CVE)、法国安全事件应急小组(FrSIRT)等机构为此先后发布了多项安全公告(如CVE-2008-4201, VUPEN/ADV-2008-2919),相关软件商也迅速发布了安全补丁。

第七章　总结

7.1　本文工作总结

本文以面向二进制程序漏洞挖掘为总体目标，针对静态分析高误报率、静态符号执行面临的执行空间爆炸和校验和机制导致模糊测试失效等问题，重点研究了细颗粒度污点跟踪、混合符号执行、校验和感知模糊测试、反馈式畸形样本生成、整数溢出漏洞建模与检测等关键技术，对上述问题提出了解决方法。本文主要包括以下研究内容：

1. 本文首次提出了一种校验和感知的模糊测试技术。该技术综合运用了混合符号执行与细颗粒度动态污点跟踪方法，通过自动定位程序中校验和检测点、修复畸形样本的校验和域，成功绕过程序中的校验和检查机制。该技术为运用传统模糊测试发现深藏的软件漏洞扫除了障碍，扩展了模糊测试技术的应用范围，同时也提高了模糊测试的漏洞挖掘效率。该项研究发表于信息安全领域顶级会议IEEE Symposium on Security and Privacy 2010，并荣获最佳学生论文奖。这是中国大陆31年来首次在该会议上发表论文。

2. 提出了两种反馈式畸形样本生成技术：基于细颗粒度污点分析的导向性测试例生成和基于混合符号执行的测试例生成技术。基于细颗粒度污点分析的导向性测试例生成有效地降低了变异空间范围，提高了模糊测试的效率；作为导向性模糊测试的重要互补，基于混合符号执行的测试例生成技术能够对单条执行轨迹进行深度安全分析并生成高代码覆盖率的样本。此外，这两种样本生成技术都可以与校验和感知模糊测试技术相结合，提高模糊测试系统漏洞挖掘能力。该项研究得到了国家自然科学基金(61003216)资助，申请发明专利一项(201010171996.4)；研究论文已被信息安全领域顶级期刊ACM Transactions on Information and System Security (TISSEC)录用。

3. 将整数溢出漏洞抽象为一个source-sink模型。在该模型指导下，提出程序脆弱性包络提取方法；进而通过静态符号执行技术遍历脆弱性包络，避免了对整个代码空间遍历，有效缓解了路径爆炸问题。该项研究成果成功应用于国家863项目(2006AA01Z402)，项目验收结论指出该成果“能够有效提升我国在安全漏洞与恶意代码研究方面的能力”；同时获得授权专利1项(200910076769.0)，相关研究论文发表于系统安全领域国际高水平会议Annual Network and Distributed System Security Symposium (NDSS) 2009，是16年来中国大陆研究人员首次以第一作者身份在该会议上发表论文。

4. 提出一种基于roBDD的污点分析方法，并实现了原型系统TaintReplayer。TaintReplayer采用离线污点分析模式，在对程序执行轨迹的重放过程中，采用roBDD描述污点属性，进行细颗粒度污点分析。实验结果表明，roBDD结构降低了污点分析对内存的需求，显著提高了污点分析的性能。相关研究论文被北京大学学报录用，申请发明专利一项(201010171979.0)。

5. 本文设计实现了软件安全漏洞动态挖掘系统TaintScope和整数溢出漏洞静态挖掘系统IntScope，应用这些系统在Microsoft、Adobe、Google等著名IT公司的产品中发现数十个零日漏洞，大部分漏洞已被国家安全漏洞库、国际安全漏洞机构CVE等权威漏洞管理组织收录。

7.2 下一步工作展望

在本文工作基础上，未来我们将对以下几个问题深入研究：

1. 动-静态漏洞挖掘系统整合和支撑分析平台构建。本文在动态漏洞挖掘和静态漏洞挖掘方法上进行了研究，但由于动态、静态漏洞挖掘方法各有利弊，在未来工作中我们将重点研究如何更好的结合两类漏洞挖掘的方法，发挥两类挖掘技术各自优势、弥补两类技术的不足，实现灵活高效的动静结合的漏洞挖掘方法，并对已有挖掘系统进行整合和改进。进一步，我们将研究构造健壮的、功能丰富的、可扩展性强的综合性漏洞挖掘研究平台，从基础设施

层面为漏洞挖掘提供控制流分析、数据流分析、切片分析、代码插桩等支撑技术。

2. 其他复杂漏洞模型分析与检测方法研究。对于复杂机理的安全漏洞，现有的动静态挖掘技术仍然面临很多其他难题。未来工作中，我们将总结提炼更多的漏洞模型，涵盖代码设计、实现、编译等各个阶段引入的安全漏洞的模式，并研究改进漏洞检测技术，逐步提高漏洞挖掘方法的效率和准确性。

3. 并行化二进制漏洞挖掘研究。本文对静态漏洞挖掘过程中路径爆炸问题提出了缓解方法，但是不可否认，面向大规模软件进行复杂机理的漏洞挖掘仍然饱受路径爆炸问题的困扰。利用大规模计算集群来加速漏洞静态分析的效率是一个可能的解决方向。随着云计算的不断发展，基于云计算平台并行化漏洞挖掘工作吸引了众多研究人员的高度关注。如何结合不同漏洞模型、设计并行化漏洞挖掘算法是本文未来工作的一个重点。

4. 对抗策略下漏洞可利用性研究。漏洞攻防技术在博弈过程中不断发展。随着地址随机化、数据执行保护等防护策略的广泛应用，漏洞利用技术遇到了巨大的挑战。如何自动化判定漏洞是否可以被利用，以及如何自动生成攻击样例，对攻防双方都有着重要的意义。在操作系统防护能力不断增强、攻击越来越困难的情况下，漏洞可利用性分析将成为下阶段的研究重点。

参考文献

[1] 卢凯 张英卢锡城, 李根. 面向高可信软件的整数溢出错误的自动化测试. 软件学报, 21(2):179–193, 2010.

[2] 韦韬. 面向二进制代码安全分析的反编译关键技术研究. 博士论文, 北京大学, 2007.

[3] 王嘉捷. 多重循环程序内存访问越界增量检测方法. 博士论文, 中国科学技术大学, 2008.

[4] 陈海波. 云计算平台可信性增强技术的研究. 博士论文, 复旦大学, 2008.

[5] 梅宏，王千祥，张路，王戟. 软件分析技术进展. 计算机学报, 32(9):1697–1710, 9 2009.

[6] 吴世忠. 信息安全漏洞分析回顾与展望. 清华大学学报, 49(S2):2065–2072, 2009.

[7] 李佳静，王铁磊，韦韬，凤旺森，邹维. 一种多项式时间的路径敏感的污点分析方法. 计算机学报, 32:1845–1855, Sept 2009.

[8] ASTR*É*E. "http://www.astree.ens.fr/".

[9] BuDDy: A BDD package. http://buddy.sourceforge.net/.

[10] Coverity Inc. "http://www.coverity.com/".

[11] CVE: Vulnerability Type Distributions. "http://cwe.mitre.org/documents/vuln-trends/index.html".

[12] FAAD2: MPEG-4 and MPEG-2 AAC Decoder. "http://www.audiocoding.com/faad2.html".

[13] GiNaC: A Free Computer Algebra Aystem. "http://www.ginac.de/".

[14] NVD: National Vulnerability Database. "http://nvd.nist.gov/".

[15] The Xen Hypervisor. "http://www.xen.org/".

[16] UQBT: A Resourceable and Retargetable Binary Translator. "http://www.itee.uq.edu.au/~cristina/uqbt.html".

[17] Vine: BitBlaze Static Analysis Component. "http://bitblaze.cs.berkeley.edu/vine.html".

[18] Xine: A Free Video Player. "http://xinehq.de/".

[19] 010editor. The professional text/hex editor with binary templates. http://www.sweetscape.com/010editor/.

[20] Humberto Abdelnur, Radu State, and Olivier Festor. Advanced fuzzing in the voip space. *Journal in Computer Virology*, 6:57–64, 2010.

[21] AdobeFlash. Flash Player penetration. http://www.adobe.com/products/player_census/flashplayer/.

[22] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25:246–256, June 1990.

[23] Afred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools (Second Edition)*. Addison-Wesley, 2006.

[24] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[26] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *In IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[27] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32:23:1–23:84, August 2010.

[28] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Automatic theorem proving for predicate abstraction refinement. In *16th Conference on Computer Aided Verification (CAV'04)*, 2004.

[29] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Programming Language Design and Implementation (PLDI'01)*, 2001.

[30] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of programming languages (POPL'03)*, 2003.

[31] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[32] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin javascript capability leaks: detection, exploitation, and defense. In *Proceedings of the 18th conference on USENIX security symposium*, pages 187–198, Berkeley, CA, USA, 2009. USENIX Association.

[33] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.

[34] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.

[35] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley Professional, December 2002.

[36] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM.

[37] Blexim. Basic integer overflows. *Phrack Magazine*, 60(10), 2002.

[38] T. Boutell. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083, Internet Engineering Task Force, March 1997.

[39] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, New York, NY, USA, 1975. ACM.

[40] David Brumley, Tzi cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, San Diego, CA, 2007.

[41] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Bitscope: Automatically dissecting malicious binaries, 2007. Technical Report CMU-CS-07-133, Carnegie Mellon University.

[42] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 143–157, Washington, DC, USA, 2008. IEEE Computer Society.

[43] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *the 20th IEEE Computer Security Foundations Symposium*, 2007.

[44] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[45] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24:293–318, September 1992.

[46] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[47] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babic, and Dawn Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communication Security*, Chicago, IL, October 2010.

[48] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, New York, NY, USA, 2007. ACM.

[49] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, USA, 2008.

[50] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):1–38, 2008.

[51] Ebrima Ceesay, Jingmin Zhou, Michael Gertz, Karl Levitt, and Matt Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in c programs. In *Detection of Intrusions and Malware & Vulnerability Assessment*, volume 4064 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2006.

[52] Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in metal. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 51–60, New York, NY, USA, 2002. ACM.

[53] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of c code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04)*, San Diego, CA, 2004.

[54] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, 2002.

[55] Brian Chess and Jacob West. *Secure programming with static analysis*. Addison-Wesley, 2007.

[56] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using meta-level compilation to check flash protocol code. In *ASPLOS-IX: Proceedings of*

the ninth international conference on Architectural support for programming languages and operating systems, pages 59–70, New York, NY, USA, 2000. ACM.

[57] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[58] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, QUEENSLAND UNIVERSITY OF TECHNOLOGY, Australian, 1994.

[59] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[60] James Clause and Alessandro Orso. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 249–260, New York, NY, USA, 2009. ACM.

[61] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998.

[62] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 110–125, Washington, DC, USA, 2009. IEEE Computer Society.

[63] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[64] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 7–9, New York, NY, USA, 2007. ACM.

[65] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'79)*, pages 269–282, New York, NY, USA, 1979. ACM.

[66] Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'00)*, pages 12–25, New York, NY, USA, 2000. ACM.

[67] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 235–248, New York, NY, USA, 2005. ACM.

[68] Jedidiah R. Crandall, Shyhtsun Felix Wu, and Frederic T. Chong. Experiences using minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, pages 32–50, 2005.

[69] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: automatic reverse engineering of input formats. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, New York, NY, USA, 2008. ACM.

[70] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[71] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, pages 57–68, New York, NY, USA, 2002. ACM.

[72] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[73] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[74] Will Dormann and Dan Plakosh. Vulnerability detection in activex controls through automated fuzz testing, 2008. `www.cert.org/archive/pdf/dranzer.pdf`.

[75] Will Drewry and Tavis Ormandy. Flayer: exposing application internals. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[76] Dwheeler. Flawfinder software. `http://sourceforge.net/projects/flawfinder/`.

[77] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 129–140, New York, NY, USA, 2009. ACM.

[78] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, New York, NY, USA, 2004. ACM.

[79] Csaba Faragó. Union slices for program maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[80] FileFuzz. From iDefense Labs. `http://labs.idefense.com/software/fuzzing.php`.

[81] Halvar Flake. Structural comparison of executable objects. In *In Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA*, pages 161–173, 2004.

[82] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium*, 2000.

[83] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, 2002.

[84] FTPStress Fuzzer, 2006. `http://www.infigo.hr/en/in_focus/tools`.

[85] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[86] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[87] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*, ICICS '08, pages 238–255, Berlin, Heidelberg, 2008. Springer-Verlag.

[88] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[89] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 206–215, New York, NY, USA, 2008. ACM.

[90] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[91] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[92] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, pages 69–82, New York, NY, USA, 2002. ACM.

[93] G. Holzmann. Static source code checking for user-defined properties. In *World Conference on Integrated Design and Process Technology*, 2002.

[94] O. Horovitz. Big loop integer protection. *Phrack Magazine*, 60(9), 2002.

[95] IDAPro. The IDA Pro Disassembler and Debugger. `http://www.hex-rays.com/idapro/`.

[96] Intel HEX. `http://en.wikipedia.org/wiki/Intel_HEX`.

[97] Noah Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, Oakland, 2011.

[98] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

[99] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2011.

[100] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Second Workshop on Virtual Machine Security (VMSec)*, Chicago, IL, November 2009.

[101] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.

[102] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[103] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDIFF Generic Differencing and Compression Data Format. RFC 3284, Internet Engineering Task Force, June 2002.

[104] D Kroening, E Clarke, and K Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th design automation conference (DAC'03)*, 2003.

[105] Corrado Leita, Marc Dacier, and Georg Wicherski. Sgnet: a distributed infrastructure to handle zero-day exploits. Technical Report EURECOM+2164, Institut Eurecom, France, 02 2007.

[106] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[107] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2008)*, Anchorage, Alaska, USA, June 2008.

[108] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[109] Guang-Hong Liu, Gang Wu, Zheng Tao, Jian-Mei Shuai, and Zhuo-Chun Tang. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In *Proceedings of the 2008 Third International Conference on Convergence and Hybrid Information Technology - Volume 02*, pages 491–497, Washington, DC, USA, 2008. IEEE Computer Society.

[110] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 190–200, New York, NY, USA, 2005. ACM.

[111] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ASE*, 2007.

[112] Mangleme, 2008. `http://freshmeat.net/projects/mangleme/`.

[113] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[114] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. *SIGOPS Oper. Syst. Rev.*, 41:78–86, January 2007.

[115] Barton P. Miller, Lars Fredriksen, and So Bryan. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[116] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison, 1995.

[117] MiniFuzz. File Fuzzer. `http://www.microsoft.com/security/sdl/getstarted/tools.aspx`.

[118] MoBB. Month of browser bugs. `http://browserfun.blogspot.com`.

[119] MOKB. Month of Kernel Bugs. `http://projects.info-pull.com/mokb/`.

[120] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[121] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[122] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. Cmc: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, 2002.

[123] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[124] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[125] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[126] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: automatic protocol replay by binary analysis. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 311–321, New York, NY, USA, 2006. ACM.

[127] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, San Diego, CA, February 2005.

[128] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3(2):58–62, 2005.

[129] Tavis Ormandy. Making Software Dumberer. `http://taviso.decsystem.org/making_software_dumber.pdf`.

[130] Heewan Park, Seokwoo Choi, Sunae Seo, and Taisook Han. Scv: Structure and constant value based binary diffing. In *Proceedings of the 2008 International Conference on Information Security and Assurance (isa 2008)*, pages 32–35, Washington, DC, USA, 2008. IEEE Computer Society.

[131] PCAP. Next Generation Dump File Format. `www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html`.

[132] Peach. Fuzzing Platform. `http://peachfuzz.sourceforge.net/`.

[133] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007.

[134] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40:15–27, April 2006.

[135] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[136] Noam Rathaus and Gadi Evron. *Open Source Fuzzing Tools*. Syngress, 2007.

[137] David Rice. *Geekonomics: The Real Cost of Insecure Software*. Addison-Wesley Professional, 2007.

[138] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen. Protos - systematic approach to eliminate software vulnerabilities. In *Invited presentation at Microsoft Research*, Seattle, USA, 2002.

[139] Dipanwita Sarkar, Muthu Jagannathan, Jay Thiagarajan, and Ramanathan Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. In *Proceedings of the 25th conference on IASTED International Multi-Conference*, pages 334–340, Anaheim, CA, USA, 2007. ACTA Press.

[140] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 225–236, New York, NY, USA, 2009. ACM.

[141] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[142] Secunia. Secunia Website. `http://secunia.com/`.

[143] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[144] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[145] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[146] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (Oakland'09)*, May 2009.

[147] SPIKE. Spike Fuzzing Platform. `http://www.immunitysec.com/resourcesfreesoftware.shtml`.

[148] Lance Spitzner. The value of honeypots, part one: Definitions and values of honeypots, 2001. `http://www.securityfocus.com/infocus/1492`.

[149] William Stallings. *Cryptography and Network Security (4th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[150] StaticAnalysis. List of tools for static code analysis. `http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`.

[151] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGPLAN Not.*, 39:85–96, October 2004.

[152] SWF. The SWF File Format. http://www.adobe.com/devnet/swf/.

[153] SWFTools. SWF manipulation and generation utilities. http://www.swftools.org/.

[154] TAR. The gnu version of the tar archiving utility. http://www.gnu.org/software/tar/.

[155] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. http://www.nist.gov/director/planning/upload/report02-3.pdf.

[156] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests with unit meister. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, New York, NY, USA, 2005. ACM.

[157] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, ACSAC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[158] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland'10)*, May 2010.

[159] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2009.

[160] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 561–561. Springer Berlin / Heidelberg, 2005.

[161] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of Annual Network and Distributed System Security Symposium (NDSS'06)*, San Diego, CA, February 2006.

[162] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A new algorithm for identifying loops in decompilation. In *14th International Static Analysis Symposium (SAS 2007)*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2007.

[163] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. Structuring 2-way branches in binary executables. In *31st Annual International Computer Software and Applications Conference (COMPSAC'07)*, 2007.

[164] Tao Wei, Tielei Wang, Xinjian Zhao, and Wei Zou. Automatically resolving virtual function calls in binary executables. In *2011 International Conference on Data Engineering and Internet Technology*, Bali, Indonesia, March 2011.

[165] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, 1979.

[166] C WEISSMAN. System security analysis/certification methodology and results. System Development Corporation, Santa Monica, CA, October 1973.

[167] C WEISSMAN. *Handbook for the Computer Security Certification of Trusted Systems.* Naval Research Laboratory Technical Memorandum, Jan 1995.

[168] Jeffrey Wilhelm and Tzi-cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *RAID'07: Proceedings of the 10th international conference on Recent advances in intrusion detection*, pages 219–235, Berlin, Heidelberg, 2007. Springer-Verlag.

[169] Rafal Wojtczuk. Uqbtng: a tool capable of automatically finding integer overflows in win32 binaries. In *22nd Chaos Communication Congress*, 2005.

[170] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic Network Protocol Analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[171] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[172] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, 2003.

[173] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004.

[174] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[175] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of 15th Annual Net-*

work and Distributed System Security Symposium, San Diego, CA, February 2008.

[176] Heng Yin, Dawn Song, Manuel Egele, and Christopher Kruegeland Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, October 2007.

[177] Z3. An Efficient SMT Solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

[178] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of European Symposium on Research in Computer Security (ESORICS'10)*, Athens, Greece, September 2010.

[179] Xiangyu Zhang, R. Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pages 319 – 329, May 2003.

[180] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 502–511, Washington, DC, USA, 2004. IEEE Computer Society.

[181] David Zimmer. Comraider: Com object fuzzer, 2008. `http://labs.idefense.com/software/fuzzing.php#more_comraider`.

[182] Zzuf. Transparent application input fuzzer, 2007. `http://caca.zoy.org/wiki/zzuf`.

攻读博士学位期间发表论文

已发表/录用期刊论文

1. **Tielei Wang**, Tao Wei, Guofei Gu, Wei Zou. Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution. ACM Transactions on Information and System Security (TISSEC), accepted.

2. **王铁磊**, 韦韬, 邹维. roBDD-Based Fine-grained Dynamic Taint Analysis. 北京大学学报，已录用. (EI Index)

3. 李佳静，**王铁磊**，韦韬，凤旺森，邹维。一种多项式时间的路径敏感的污点分析方法。计算机学报，Vol 9,2009. (EI 20094512438443).

4. 段镭，韦韬，**王铁磊**，郭天放，邹维. 一种基于指令填充随机化的JIT-Spraying防护方法。清华学报, 2010. (EI 20104813430874).

已发表会议论文

1. **Tielei Wang**, Tao Wei, Guofei Gu, Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, CA, May 2010. (Acceptance Ratio 10.9%=26/237, **Best Student Paper**，顶级学术会议，中国大陆研究机构31年来首发，EI 20103213125503).

2. **Tielei Wang**, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In Proceedings of the 16th Network and Distributed System Security Symposium (NDSS'09), San Diego, CA, February 2009. (AR: 11.7%=20/171，系统安全顶级会议，中国大陆研究机构16年来首次以第一作者单位发文).

3. Tao Wei, **Tielei Wang**, Lei Duan, Jing Luo. Secure Dynamic Code Generation against Spraying. In 17th ACM Conference on Computer and Communications Security (CCS'10), Chicago, USA, OCT, 2010. (EI 20102012932884).

4. Chao Zhang, **Tielei Wang**, Tao Wei, Yu Chen, Wei Zou. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. In European Symposium on Research in Computer Security (ESORICS'10), Athens, Greece, September 2010. (AR 20.8%=42/201, EI 20104513355740).

5. Yu Ding, Tao Wei, **Tielei Wang**, Zhenkai Liang, Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In Annual Computer Security Applications Conference (ACSAC), Austin, Texas, 2010. (AR 17%=39/227, EI 20110413614188).

6. Tao Wei, **Tielei Wang**, Xinjian Zhao, Wei Zou. Automatically Resolving Virtual Function Calls in Binary Executables. 2011 International Conference on Data Engineering and Internet Technology, Bali, Indonesia, March 2011. (EI Index)

7. Tao Wei, **Tielei Wang**, Lei Duan, Jing Luo. INSeRT: Protect Dynamic Code Generation Against Spraying. In 2011 International Conference on Information Science and Technology (ICIST), Nanjing, China, March 2011. (EI Index).

8. Shuaifu Dai, Yaxin Liu, **Tielei Wang**, Tao Wei, Wei Zou. Behavior-Based Malware Detection on Mobile Phone. In 6th International Conference on Wireless Communications Networking and Mobile Computing(WiCOM), Chendu, China, September 2010. (EI 20104713417212).

9. 赵新建，**王铁磊**，韦韬，邹维. 程序静态分析工具的分类研究，第六届中国信息和通信安全学术会议, 2009.

已投稿论文

1. Chao Zhang, **Tielei Wang**, Tao Wei, Yu Chen, Wei Zou. Using Type Analysis in Compiler to Defeat Integer-Overflow-to-Buffer-Overflow Threat. Submitted to Journal of Computer Security (JOCS).

专利

1. 一种程序分解方法. **王铁磊**, 韦韬, 邹维, 毛剑, 李佳静, 赵新建, 张超, 戴帅夫. **已授权**(200910076769.0).
2. 一种复合条件分支结构的识别方法. 韦韬, **王铁磊**, 毛剑, 李佳静, 王伟, 邹维. **已授权**(200610169677.3).
3. 多路分支结构的识别方法. 韦韬, **王铁磊**, 毛剑, 邹维, 李佳静, 王伟. **已授权**(200710090004.3).
4. 一种嵌套循环结构的识别方法. 韦韬, 李佳静, 毛剑, 邹维, **王铁磊**, 王伟. **已授权**(200710090003.9).
5. 一种Web通信加密方法. 韦韬, 毛剑, 邹维, **王铁磊**, 李佳静, 王伟. **已授权**(200610170718.0).
6. 一种恶意代码检测方法. 李佳静, 张超, 韦韬, **王铁磊**, 邹维, 毛剑, 戴帅夫, 赵新建. **已授权**(200810089576.4).
7. 一种高效动态软件漏洞挖掘方法. **王铁磊**, 李义春, 韦韬, 邹维, 戴帅夫, 张超, 丁羽. 已公示(201010171996.4).
8. 一种软件漏洞挖掘方法. **王铁磊**, 韦韬, 邹维, 张超, 戴帅夫, 丁羽, 李义春. 已公示(201010171979.0).
9. 一种可信电子交易方法及其交易系统. 毛剑, 韦韬, 戴帅夫, 邹维, **王铁磊**, 张超, 赵新建, 李佳静. 已公示(200810167891.4).

作者简历

个人信息

姓名：王铁磊　　性别：男　　出生日期：1985年7月　　籍贯：内蒙古赤峰

教育背景

2006.9 - 今	北京大学计算机科学技术研究所	博士
2002.9 - 2006.6	北京大学物理学院	物理学学士
2003.9 - 2006.6	北京大学经济研究中心	经济学学士

荣誉奖励

攻读博士学位期间，荣获北京大学创新奖(2010)、北京大学三好学生(2009)、北京大学学习优秀奖(2007)、31st IEEE S&P会议最佳学生论文奖(2010)、北京大学信息科学技术学院学术十杰(2010)、长飞奖学金(2010)、华为奖学金(2009)等荣誉奖励。

科研项目经历

1. 2006-2008，国家863计划项目：二进制程序中的安全漏洞挖掘技术研究（2006AA 01Z402），主要参与者；
2. 2006-2008，总参预研项目：XXX，主要参与者；
3. 2007-2010，国防预研项目：XXX，主要参与者；
4. 2010-2011，自然科学基金：基于汇点脆弱性评估的可伸缩并行化漏洞挖掘方法研究（61003216），主要参与者。

致谢

从本科入学算起，我在北大已经度过了九年时光。这九年的求学经历，是我生命中宝贵的财富，让我更加懂得生活中要怀有一颗感恩、知足和进取的心。

本文是在我的导师邹维研究员悉心指导下完成的，凝聚了邹老师大量的心血。邹老师对我的谆谆教导和无微不至的关怀让我永生受益。邹老师高尚的人格魅力、忘我的工作态度、严谨的治学精神是我人生路上的指路灯。师恩如山，难于言表!

特别感谢韦韬博士。韦韬博士与我亦师亦友。早在2005年，就是经过韦韬博士面试，我才进入研究所实习，可以说是韦韬博士将我领进了计算机科学的殿堂。韦韬博士渊博的学识、敏锐的思维、犀利的观点、正直的人品、乐观的态度都是我所学习的榜样。

衷心感谢课题组张超博士、戴帅夫博士、李佳静博士及其他同学。能和你们一起共事学习是我莫大的荣幸。我们之间兄弟姐妹一样的友谊是我永远的骄傲和财富。感谢陈昱老师、张建宇博士、韩心慧博士、诸葛建伟博士以及信安中心的其他老师和同学。是你们的帮助和鼓励，让我始终觉得中心就像一个大家庭，时时刻刻感受到团队的力量和温暖。

感谢美国普度大学林志强博士、Texas A&M 大学顾国飞教授、新加坡国立大学梁振凯教授、UC Berkeley Dawn Song教授.。你们无私的帮助让我更好的了解国际学术氛围和研究潮流；与你们的交流和合作让我受益良多。

最后，我要感谢的父母、我的妻子和我的兄长。你们对我的支持、理解、关爱是我永远向前的动力。

北京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，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过的作品或成果。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。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校□一年/□两年/□三年以后，在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：

日期：　　年　　月　　日