基于延后策略的动态多路径分析方法

陈 恺[1),2),3)] 冯登国[1),2)] 苏璞睿[2)]

[1)]（中国科学院研究生院信息安全国家重点实验室 北京 100049）

[2)]（中国科学院软件研究所信息安全国家重点实验室 北京 100190）

[3)]（信息安全共性技术国家工程研究中心 北京 100190）

摘 要 多路径分析是弥补传统动态分析方法的不足、对可执行程序全面分析的重要方法之一.现有多路径方法主要采用随机构造或者根据路径条件构造输入进行路径触发，这两者均存在路径分析不全面和缺乏针对性的问题.文中通过对路径条件分析，确定了检测条件的基本组成元素，提出了弱控制依赖和路径引用集的概念和计算规则，并以此为基础提出一种延后策略的多路径分析方法.在程序分析过程中，对特定的程序检测点和检测点条件，有针对性地进行路径筛选，从语义上进行路径表达式简化，在保证检测点可达和检测表达式具有相同构造形式的前提下，简化检测表达式，减少分析路径的数量.对7款恶意软件的分析实验结果表明，该方法提高了分析效率和准确性.

关键词 多路径分析；可执行程序；漏洞检测；动态分析；延后策略

中图法分类号 TP311 **DOI号**：10.3724/SP.J.1016.2010.00493

Exploring Multiple Execution Paths Based on Dynamic Lazy Analysis

CHEN Kai[1),2),3)] FENG Deng-Guo[1),2)] SU Pu-Rui[2)]

[1)]（*State Key Laboratory of Information Security*, *Graduate University of Chinese Academy of Sciences*, *Beijing* 100049）

[2)]（*State Key Laboratory of Information Security*, *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190）

[3)]（*National Engineering Research Center for Information Security*, *Beijing* 100190）

Abstract Exploring multiple execution paths is an important method to analyze executable files. Most researchers use randomly generated input or construct input by path conditions to explore program paths. These methods suffer from two flaws: they cannot analyze all the paths while there are too many useless paths to analyze. This paper introduces weak control dependence and path reference set to analyze path conditions. It also ensures three basic kinds of elements in checked conditions. Lazy analysis is proposed based on these definitions and theories to explore multiple execution paths. When analyzing a program, it can choose suitable branch conditions to explore paths according to a program check point. In this way, the number of path conditions can be decreased without missing any necessary conditions that guarantee the program to run to the check point and the checked condition to have the same structures. A prototype is implemented to make some experiments on seven malwares. Taint analysis is used to trace the input from outer space such as tainted files in the overall analysis process. Shadow memory is also exploited to increase the managing speed. The results show that the method decreases the number of path conditions and increases the efficiency when exploring multiple paths.

Keywords multiple execution paths analysis; executable files; vulnerability detection; dynamic analysis; lazy analysis

---

收稿日期：2009-04-13；最终修改稿收到日期：2009-10-14.本课题得到国家自然科学基金(60703076,60970028)、国家"八六三"高技术研究发展计划项目基金(2006AA01Z412,2007AA01Z451,2007AA01Z475,2007AA01Z465,2007AA01A414)资助.**陈 恺**，男，1982年生，博士研究生，主要研究方向为信息安全、软件漏洞分析与检测、恶意代码分析与防范.E-mail：chenk@is.iscas.ac.cn. **冯登国**，男，1965年生，博士，研究员，主要研究领域为密码学与信息安全. **苏璞睿**，男，1976年生，博士，副研究员，主要研究方向为信息安全.

1 引 言

软件分析是检测软件漏洞、软件恶意行为等安全性问题的基础.按分析目标的不同,现有软件分析方法一般分为面向源代码的软件分析和面向可执行程序的软件分析.前者针对有源代码的程序,具有更加丰富的类型信息和结构信息,相对而言,分析准确性更高.但是现有软件多数不包括源代码,尤其是大型的商业软件和进口软件等;同时,即使部分软件包括了源代码,也不能保证使用的可执行程序和源代码之间有对应关系.对这一问题 Godefroid 等在文献[1]中进行了详细讨论.由于大量的应用软件无法获得源程序,而要用在一些重要领域,因此直接对其可执行程序进行安全性分析、确保其安全性显得极为重要.该问题也是国内外研究的热点问题.可执行程序分析方法一般分为静态方法和动态方法两种.

静态分析是将可执行程序的二进制代码转变为汇编语言并以此为基础进行分析.其优点是可以较为全面地分析程序代码,但是分析过程中需要进行大量的推理和符号演算,因此效率较低,且会造成一定数量的误报[2],另外,对于一些经过变形和混淆[3]技术处理的代码也不能很好地处理.动态分析方法的基本思想是利用程序运行时的数据提高分析的效率和准确性,同时避免由于变形等反静态分析技术带来的不可分析性.传统的动态分析一次只能分析一个运行实例,例如 Softice、Ollydbg 等.若要提高动态分析的全面性,就要构造执行应用程序的多种可能执行路径,即对可执行程序进行多路径分析.

按照动态多路径分析方法的发展,可以将其分为 3 类.第 1 类是通过将可执行程序放在可控环境(例如调试器等)中执行并手动更改分支语句的判断条件来进行多路径分析.这类方法需要大量的人工参与,非常繁琐且不具全面性.第 2 类是自动构造不同的输入,尝试触发程序执行的各种不同路径以暴露出程序潜在的安全问题.这类方法也称为 Fuzz 方法.虽然此类方法较之第 1 类方法在一定程度上提高了分析的效率和覆盖率,但是大多数情况下,此类方法仅能穷举有限个输入,并不能对所有的输入都进行测试.因此通过此方法验证的程序会有一定程度的漏报,而且会耗费大量时间重复测试.第 3 类方法是通过对程序执行过程中的路径条件的求解,有选择地对路径进行分析,较有代表性的如 EXE[2] 等.这类方法使用动态分析方式,提高了静态分析的准确性和效率,同时避免了 Fuzz 方法的随机性,增加了路径选择的效率.但是这类方法仍然会产生过多的分析路径,也没有对路径条件进行分析筛选,以至于难以应用到大型程序中[4].目前国际上部分学者使用启发式方法尝试减少路径分析数量[5],但是效果仍然不理想[4].本文所述方法是以第 3 类方法为基础,从路径条件与检测点之间关系入手,围绕着条件表达式的组成结构加以展开.

我们发现,多路径分析的作用之一在于确定某条语句或若干条语句的集合(称为程序检测点集合)在不同的执行路径下是否满足一定的需求,例如判断某条语句是否存在漏洞或者是否存在恶意行为等.此时在程序分析过程中,部分分支条件的取值并不会影响程序检测点的判断条件,因此产生了条件冗余.针对以上应用场景,本文提出了一种延后策略的可执行程序动态多路径分析方法.与传统多路径分析方法不同,本方法在程序的分支路径选择过程中,并不立即进行路径表达式求解和启用新进程进行多路径分析,而是仅记录分支条件;当遇到程序检测点时,有选择地对部分分支语句进行多路径分析,减少需要分析的路径数量和检测表达式长度,提高了分析效率和准确性.

本文的主要贡献如下:

(1) 对路径执行条件进行分析,确定了检测条件的基本组成元素,提出了弱控制依赖和路径引用集的概念和计算规则,并以此为基础对路径条件进行筛选,简化了检测条件表达式,提高了表达式求解的效率和准确性.

(2) 提出了一种延后策略的多路径分析方法.在动态分析的过程中,并不立即对分支语句进行多路径分析,而是在确定检测点位置和检测条件后,动态分析路径条件,有选择地对检测表达式有控制作用的分支语句进行多路径分析,提高了多路径分析的针对性,简化了多路径构造过程,改进了分析效率.

(3) 实现了一套基于延后策略的多路径分析原型系统.对 Perfect Keylogger 等 7 款具有恶意操作的软件进行分析,验证了其实际效果.

2 相关工作

在反汇编方面,目前已有了较为成熟的方法和工具,例如 IDA Pro① 等.威斯康星大学的 WiSA 项目组在可执行程序的静态分析方面做了大量的工

① IDA Pro Disassembler - multi-processor, windows hosted disassembler and debugger. http://www.datarescue.com/ida.htm.

作，提出了分析可执行程序的 VSA[6] 方法，并开发出 CodeSurfer/x86[7] 等工具. 在反汇编的基础上进行符号执行，可以对程序进行多路径分析[8-9]. 国内如夏一民等人在对漏洞进行检测时，提出了基于条件约束的方法[10]，此方法也可应用在多路径分析上. 静态分析虽然可以较为全面地分析程序代码，但是由于缺乏程序运行时的数据信息，所以分析效率较低，且会造成一定程度的误报[2]. 对于一些经过变形和混淆[3]技术处理的代码，静态分析也难以处理. 目前，人们使用一些基础理论方法，例如切片方法[11]，试图提高静态分析的准确性，但效果仍然不理想[12].

动态多路径分析是目前重要的多路径分析方法. 最初人们为了触发程序的不同路径，尽量多地构造出不同的输入，这类方法称为 Fuzz 方法. 较有代表性的是 Ghosh 的方法[13]，它将程序看作是黑盒，通过变换不同的输入，观察程序是否会出现异常，从而进行漏洞查找. 这类方式不需要对程序进行分析，而是对程序进行测试，所以执行速度较快. 但是大多数情况下，此类方法仅能穷举有限个输入，并不能对所有的输入都进行测试，因此通过此方法验证程序会存在一定程度的漏报. 同时，这类方法效率很低，例如程序中有分支语句 $x=10$，这类方法要对 x 进行 2^{32} 计算才能对不同分支进行处理. 之后人们通过对执行路径条件进行分析，产生了白盒 Fuzz 的方法.

白盒 Fuzz 方法是针对不同的分支条件求解，计算出可能的输入并尽量多地对不同分支进行多路径分析，较有代表性的是 EXE[2]、Moser[14] 和 SAGE[1]. 但是这类方法需要对遇到的每一个依赖于外部输入的分支都进行多路径分析，因此很大程度上影响了实现效率. Godefroid 使用了语法指导的 Fuzz 分析[15]，但是预先对缺乏源代码的可执行程序输入部分进行语法分析非常困难且准确性不高，同时这种方法仍然会产生较多的无用路径. 多种启发式方法[5,16-17]也被提出进行路径选择，试图在尽量短的时间内达到更多的代码覆盖率，但是这类方法仅是从搜索策略的角度入手，例如深度优先、广度优先、随机法或通过一定的算子计算出路径上的权值来进行路径选择，没有对路径本身的语句进行分析和优化处理，因此效果仍然不够理想[4]. RWSet 方法对指令引用集合和定义集合进行分析，从程序语义上尝试减少多路径分析的数量，具有一定效果[4]，但是它仅是从程序节点的数值依赖角度进行分析，没有对控制依赖进行分析，例如缺乏针对程序检测点的运行条件分析和对条件路径叠加的处理，因此仍然存在无用路径；且仅适用于源代码分析.

现有的多路径分析方法造成分析不全面和无用路径过多的原因在于：(1) 对条件路径不加选择地进行多路径扩展. 在程序执行过程中，遇到分支语句即对路径进行分解，进而对每个分支路径分别产生一个新进程进行计算. Cadar[2] 等人在此基础上对动态依赖于外部输入的分支语句进行路径分解，以减少路径数目，但是这类方法仍然会产生较多的无用路径. (2) 路径表达式过长以至无法求解. 在分支计算过程中，为了明确程序的执行路径，需要记录下分支条件，当分支数目增加时，条件表达式长度也随之增加. 条件表达式的数目和长度的增加，使得对程序进行多路径分析的难度显著增加，甚至在有限时间内不可求解. 本文针对以上问题，对程序检测点与路径条件的关系进行分析，确定了路径条件的组成，提出一种延后分析的策略.

3　路径条件的组成

本章以一示例为引，说明路径条件的必要性和分支选择的时机问题，并定义相关概念和具体的计算规则.

图 1(a)是一个典型的分支流程. 假设节点 1 是分支语句，其两个出口分别指向节点 2 和节点 3，节点 4 是后必经节点. 节点 1 的入口条件是 C，两个分支条件为 e 和 $\bar{e}$.

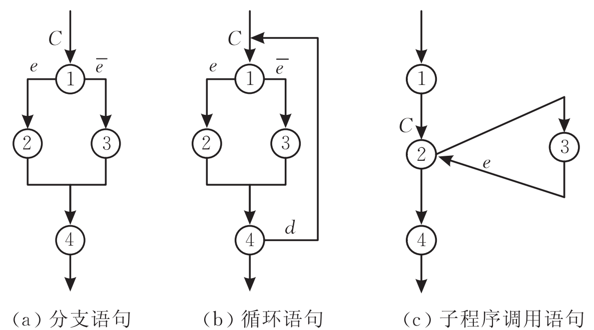


图 1　示例

现有多路径分析方法在程序运行时，会记录程序运行过程中所有不能静态确定分支方向的条件集合 CE. 例如图 1(a)中，假设条件 e 依赖于外部输入，表示为 $e \xrightarrow{\mathrm{D}} Input$，且 $\bar{e} \xrightarrow{\mathrm{D}} Input$. 当程序运行至节点 1 进行分支选择时，运行条件会变为 $C \wedge e$ 或 $C \wedge \bar{e}$，且会在节点 1 处产生新的进程进行多路径分析. 假设检测点在节点 2 或 3 时，以上分析过程是没有冗余的，因为条件 e 或 $\bar{e}$ 保证了检测点的运行

条件.但若检测点在节点4处,条件 e 和 $\bar{e}$ 的必要性并不明显,因为它们并未保证此时程序的运行条件.因此,需要对节点2~4中引用变量集和定义变量集进行更深入的分析方可确定条件 e 或 $\bar{e}$ 的必要性.易知,对于理论上最小路径条件集 CE',有 $CE'\subseteq CE$.

对于循环语句(图1(b))与子程序调用语句(图1(c)),以上情况均会出现.因此正确选择分支条件,可以减少分支的数目和路径表达式的长度,也避免产生无用的进程进行多路径分析.

定义1(路径). 路径 P 是若干条顺序执行的语句组成的序列,记为 $P=\{l_1,l_2,\cdots,l_i|i\in N\}$.用 $P[l_0,l_n]$ 表示所有以 l_0 为起点、以 l_n 为终点的路径.用 $P(l_0,l_n]$ 表示所有以 l_0 后继节点为起点,以 l_n 为终点的路径,用 $P[l_0,l_n)$ 表示所有以 l_0 起点,以 l_n 前驱节点为终点的路径.对于特定的 P,使用 $|P|$ 表示路径长度.

定义2(条件路径). 条件路径 CP 是路径 P 中顺序执行的分支语句组成的序列,记为 $CP=\{c_1,c_2,\cdots,c_i|i\in N\}$,其中 c_1 是程序执行路径上第一个分支语句.易知,$CP\subseteq P$.用 $CP[l_0,l_n]$ 表示所有以 l_0 为起点,以 l_n 为终点的条件路径.用 $CP(l_0,l_n]$ 表示所有以 l_0 后继节点为起点、以 l_n 为终点的条件路径,用 $C[l_0,l_n)$ 表示所有以 l_0 起点、以 l_n 前驱节点为终点的条件路径.对于特定的 CP,使用 $|CP|$ 表示条件路径长度.

定义3(检测条件). 检测条件指在程序运行至检测点 p 位置时,进行某些预定义检查所需的条件,例如判断当前状态是否会出现漏洞的条件等,用 CC_p 表示.

定义4(检测表达式). 对于程序执行路径 P 上的结点 l,程序检测表达式 $CE_l^P=C\wedge CC_l$.其中 C 保证程序在每次执行过程中均可到达 l,且 CC_l 具有相同的符号表示.在不发生歧义的情况下,将 CE_l^P 简记为 CE_l.

定理1. $c_i\in CC_p\Rightarrow c_i\in CE_p$.

由检测表达式的定义易证.

定义5(弱控制依赖). 在路径 $P=\{l_1,l_2,\cdots,l_i|i\in N\}$ 上,对于条件结点 l_k 及任意后续结点 l_j $(1\le k<j\le i)$,假设 l_k 的后必经结点为 l_m,若 $l_m\notin P[l_k,l_j]$,则称 l_j 弱控制依赖于 l_k,记为 $l_j\xrightarrow{\text{WCD}}l_k$.

根据弱控制依赖的定义可知其与控制依赖的关系:$l_j\xrightarrow{\text{CD}}l_k\Rightarrow l_j\xrightarrow{\text{WCD}}l_k$,其中 $l_j\xrightarrow{\text{CD}}l_k$ 表示 l_j 控制依赖于 l_k,但反之不然.

定义6(运行条件). 在条件执行路径 $CP=\{c_1,c_2,\cdots,c_n\}$ 上,对于检测点 p,如果 $p\xrightarrow{\text{WCD}}c_i$,则称条件 c_i 是检测点 p 的运行条件.用 RC_p 表示检测点 p 运行条件的集合.

定理2. $c_i\in RC_p\Rightarrow c_i\in CE_p$.

证明. 由运行条件的定义 $c_i\in RC_p\Rightarrow p\xrightarrow{\text{WCD}}c_i$ 可知,p 不是 c_i 的后必经节点,因此存在路径 $P'=\{l_1,l_2,\cdots,l_i,\cdots,l_n|i\in N\}$,其中 l_n 是结束语句,使得 $p\notin P'$.如果 $c_i\notin CE_p$,则不能保证程序运行到节点 p 处,且无法在 p 点进行表达式检测,所以定理2得证. 证毕.

在图1(a)中,如果在节点2处进行检测,有 $l_2\xrightarrow{\text{WCD}}l_1$,根据定理2,有 $e\in CE_2$.定理2是条件语句是否属于程序检测表达式的充分条件,但不是必要条件,因此对于检测点 p,当 $p\not\xrightarrow{\text{WCD}}c_i$ 时,无法确定 c_i 是否属于 CE_p.例如图1(a)中,如果在节点4处进行检验时,$l_4\not\xrightarrow{\text{WCD}}l_1$,将无法确定是否将 e 或者 $\bar{e}$ 放入 CE_4.此时,需要进一步分析程序运行过程中定义的变量集和引用的变量集.

引理1. 在条件执行路径 $CP=\{c_1,c_2,\cdots,c_n\}$ 上,假设 l 是分支节点 c_i 的第一个后必经节点,有 $\exists p(p\in P[c_i,l)\wedge V=USE(CC_l)\cap DEF(p)\wedge V\neq\varnothing\wedge\exists p'(p'\in P[c_i,l)\wedge p'\neq p\Rightarrow value(V_{p'})\neq value(V_p)))\Rightarrow c_i\in CE_l$,其中 $USE(C)$ 表示表达式 C 的引用集合,$DEF(C)$ 表示表达式 C 的定义集合,$value(V_p)$ 表示在 p 路径上 V 集合的变量值.

证明. 根据题设条件,对于程序的执行路径 $p\in P[c_i,l)$,存在不同的路径 p' 且 $p'\in P[c_i,l)$,在程序的两次执行过程中,当 c_i 取不同分支时,$value(USE(CC_l)_p)\neq value(USE(CC_l)_{p'})$,即 l 处 $USE(CC_l)$ 的值不同,因此为了保证在 l 处 CE_l 中 $USE(CC_l)$ 具有相同值,需要将 c_i 加入 CE_l. 证毕.

考虑到在不同路径上 V_p 和 $V_{p'}$ 具有相同值,在多路径分析时,添加上条件 c_i 也不会影响 CE_l 的解集,而且多数情况下编译器会将不同路径上具有相同变量赋值的语句提取出进行优化,因此引理1可以简化为定理3.

定理3. 在条件执行路径 $CP=\{c_1,c_2,\cdots,c_n\}$ 上,假设 l 是分支节点 c_i 的第一个后必经节点,则 $USE(CC_l)\cap DEF(P[c_i,l))\neq\varnothing\Rightarrow c_i\in CE_l$.

定理3仅针对分支语句的第一个后必经节点 l 进行计算,并未考虑 l 后续节点的条件依赖关系.假设分支语句 c_i 的第一个后必经节点是 l_1,检测语句为 l_1 之后的语句 l_2,且 $DEF(P[l_1,l_2))\cap$

$USE(CC_{l_2}) \neq \varnothing$，则需要考虑 $DEF(P[l_1,l_2)) \cap USE(CC_{l_2})$ 中变量定义集合与 $DEF(P[c_i,l))$ 之间的关系.

定义 7（路径引用集）.　在程序路径 $P=\{l_1,l_2,\cdots,l_i \mid i\in N\}$ 上，对于在语句 l 中使用的变量集合 s，s 依赖于路径 P 外的变量集合称为变量集合 s 在路径 P 上的路径引用集，用 $USE(P,s,l)$ 表示.

根据路径引用集的定义，可以获得 $USE(P,s,l)$ 的计算方法. 对于所有 $i\geq 1$，这里使用推导规则方式：

空语句推导规则：$\dfrac{}{USE(\varnothing,s,l):s}$；

单语句推导规则：

$$\frac{s_1=\{v \mid v\in s \wedge v\notin DEF(l_i)\},\quad s_2=\{v \mid v\in USE(l_i)\wedge DEF(l_i)\cap s\neq\varnothing\}}{USE(\{l_i\},s,l):\ (l_i\in CE_l)\mapsto(s\cup USE(l_i))\lozenge(s_1\cup s_2)};$$

多语句推导规则：

$$\frac{s'=USE(\{l_2,\cdots,l_i\},s,l)}{USE(\{l_1,l_2,\cdots,l_i\},s,l):USE(\{l_1\},s',l)};$$

多路径推导规则：

$$\frac{}{USE(P[l_i,l_j],s,l):\bigcup_{p\in P[l_i,l_j]} USE(p,s,l)};$$

其中，$c\mapsto x_1\lozenge x_2$ 表示当条件 c 成立时，选择 x_1，否则选择 x_2.

定义 8（值依赖条件）.　在条件执行路径 $CP=\{c_1,c_2,\cdots,c_n\}$ 上，假设 l_j 是分支节点 c_i 的第一个后必经节点，对于检测点为 l_k，如果 $k\geq j \wedge USE(P[l_j,l_k),USE(CC_{l_k}),l_k)\cap DEF(P[c_i,l_j))\neq\varnothing$，则称条件 c_i 是检测点 l_k 的值依赖条件. 用 VDC_l 表示检测点 l 值依赖条件的集合.

定理 4.　在条件执行路径 $CP=\{c_1,c_2,\cdots,c_n\}$ 上，对于检测点 l，有 $c_i\in VDC_l \Rightarrow c_i\in CE_l$.

证明.　因为 $c_i\in VDC_l$，根据值依赖条件的定义，存在 c_i 的后必经节点 l'，使得 $USE(P[l',l),USE(CC_l),l)\cap DEF(P[c_i,l'))\neq\varnothing$. 当 $l=l'$ 时，由空语句推导规则，有 $USE(P[l',l),CC_l,l)=USE(CC_l)$，根据定理 3，可知 $c_i\in CE_l$. 当 $l\neq l'$ 时，在路径 $P[l',l)$ 上由路径引用集的推导规则，可以求得集合 $s=USE(P[l',l),CC_l,l)$，不妨将 s 看作 $USE(CC_{l'})$，根据定理 3，可知 $c_i\in CE_l$.　　证毕.

定理 4 描述了在程序条件路径 CP 上，对于 $c_i\in CP$ 和检测点 l，当 $l \xrightarrow{WCD}\!\!\!\!\!\!\!\!/\ \ \ c_i$ 时，是否将 c_i 加入 l 点的检测表达式的充分条件. 至此，我们可以得到对于任意条件 c_i，有如下定理.

定理 5.　在程序执行路径 P 上，对于检测点 l，有 $CE_l=CC_l\cup RC_l\cup VDC_l$.

证明.　由检测表达式的定义，有 $c\in CE_l \Rightarrow c\in CC_l \vee c\in RC_l \vee c\in VDC_l$，充分性得证. 根据定理 1、定理 2 和定理 4，有 $c\in CC_l \vee c\in RC_l \vee c\in VDC_l \Rightarrow c\in CE_l$，必要性得证. 因此 $CE_l=CC_l\cup RC_l\cup VDC_l$.　　证毕.

4　延后的多路径分析方法

现在的多路径分析方法，一般是在分支语句处判断当前分支条件 c，如果 $c\xrightarrow{DD} Input$（其中 DD 表示数据依赖），则创建两个进程，分别执行并递归地进行多路径分析[2]. 当过多分支语句或者循环控制语句依赖于外部输入时，这类多路径分析方法会因为创建的分析进程数量过多而不实用（与分支数目成指数关系），这也是目前多路径分析方法难以应用于大规模程序的主要原因之一.

由定理 5，在程序执行过程中，对于条件 $c\xrightarrow{DD} Input$，若 $c\notin CE_l$，则表示 c 取值真假与 CC_l 无关，因此不需要在条件 c 处进行多路径扩展. 这里提出一种基于延后策略的多路径分析方法，针对已有的检测点，确定与检测点相关的分支语句. 此方法由两种模式组成：监控模式和检测模式. 在监控模式下，被分析程序正常运行，监控程序记录路径 P、条件路径 CP 等相关的信息. 检测模式指遇到程序检测点 l 后，判断条件 $c\in CP$ 与 CE_l 间关系的过程.

程序开始时以监控模式进行. 每当遇到指令 I 时，如果 I 是程序检测点，则转入检测模式，否则将 I 加入执行路径 P. 如果 I 是分支语句，且 $I\xrightarrow{DD} Input$，则将 I 加入条件路径 CP. 使用切片方法可以判断 I 是否依赖于外部输入，但其效率较低，且不准确. 这里我们使用污点传播方式判断 I 是否依赖于外部输入.

污点传播[18]是一种有效确认变量之间是否相关的分析方法. 通过设置污点源和定义污点传播规则，可以判断程序运行过程中变量是否与污点源相关. 污点源可以设置为外部文件、键盘输入和网络输入等. 表 1 是部分指令的污点传播规则，包括数据转移类指令、算术操作指令、逻辑操作类指令、比较类指令等. 为了能够更快地确定某块内存区域是否依赖于外部输入，我们使用了影子内存的方法. 影子内存为真实内存的镜像，其每块内存空间表示相对应的真实内存是否依赖于外部输入. 为了保持分析过程中的连贯性，我们对寄存器也实现了相应的影子机制.

表 1 污点传播规则

指令分类	传播规则	示例语句
数据转移指令	$T(op_1)\leftarrow T(op_2)$	mov op_1, op_2
算术操作指令	$T(op_1)\leftarrow T(op_1)\vee T(op_2)$; $T(\mu(add,PSW))\leftarrow T(op_1)\vee T(op_2)$	add op_1, op_2
逻辑操作指令	$T(op_1)\leftarrow T(op_1)\vee T(op_2)$; $T(\mu(and,PSW))\leftarrow T(op_1)\vee T(op_2)$	and op_1, op_2
比较类指令	$T(\mu(cmp,PSW))\leftarrow T(op_1)\vee T(op_2)$	cmp op_1, op_2

表1中 $T(op)\rightarrow BOOL$,表示 op 所指向的操作数是否为污点数据;PSW 表示机器状态字;$\mu(op,PSW)\rightarrow\mathbb{Z}$,表示 op 操作引发 PSW 中变化的位置,例如 add 操作会引起 C、P、A、Z、S、O 等标志位的改变.注意 add、and 等算术/逻辑指令会影响 PSW 的值,这里也必须引入,因为 jne 等跳转指令依赖于 PSW 的值.

由于缺少源代码,污点传播在分析过程中存在一定的局限性,主要包括如下几条:

(1)嵌套循环难以处理的问题.目前原型系统中我们采取的措施是将循环展开[19],但这样会造成需要多路径分析的分支数量过多的问题.在未来的工作过程中,我们将重点对循环进行处理,进一步减少循环对多路径分析的影响.

(2)操作指令集不完备的问题.由于 CPU 指令较多,目前的原型系统中完成了数据传输类指令、算数操作类指令、逻辑操作类指令和比较类指令等指令集合的污点传播分析,对于特殊的指令,例如 MMX 指令、FPU 指令等并没有完成污点传播分析,这可能会造成分析不全面的问题.同时我们仅支持 Intel CPU 的指令集,并不兼容其它类型的 CPU.在未来的工作中,我们将支持更多的指令集,以便完善我们的原型系统.

(3)系统调用的问题.目前原型系统仅在用户层进行分析,并没有对内核层进行污点传播处理.这可能会造成部分污点传播失效的问题,使得分析的全面性受阻.目前采用国际上使用较多的方式,对系统调用总结出"摘要"(summary)信息[20],即对部分系统调用增加一定程度的信息以保持污点传播分析的连续性.目前我们对 NtReadFile 等系统调用增加了"摘要"信息.

在目前软件安全逆向分析中,单一节点的检测往往不能作为漏洞点存在的依据,因此需要考虑多个检测点间的共同作用.假设需要共同作用的检测点集合为 $L=\{l_p,\cdots,l_q\}$,当程序运行至 L 时,进入检测模式.CP 中记录了执行路径 P 上遇到的所有依赖于外部输入的条件.不妨设 $CE_L=\bigcup_{l\in L}CE_l$.此时需要逐个判断 CP 中的条件 c 是否属于 CE_L.由定理5,$CE_l=CC_l\cup RC_l\cup VDC_l$,所以需判断条件 c 是否属于集合 $\bigcup_{l\in L}CC_l\cup RC_l\cup VDC_l$.

定义函数 $\sigma(a:P,b:P)\rightarrow\{true,false\}$ 表示结点 b 是否是结点 a 的后必经结点,如果是,则返回 true,否则返回 false.

对于路径 $P=\{l_1,l_2,\cdots,l_i,l\}$,其中 l 是 L 中执行流程序列的最后一条语句,从指令 l_i 开始逆序进行分析.对于第一条指令 l_i,根据不同情况:

(1)$l_i\in CP$ 时,$\sigma(l_i,l)\equiv false$,且由于 $|P[l_i,l]|=2$,因此 $l\xrightarrow{WCD}l_i$,即 $l_i\in RC_l$,根据定理2,将 l_i 加入 CE_l,用 $l_i\rightarrow CE_l$ 表示.若 $l_i\in L$,更新路径引用集 $s_i=USE(\{l_i\},s,l)\cup USE(CC_{l_i})$.继续分析 l_{i-1}.

(2)$l_i\notin CP$ 时,使用路径引用集的推导规则计算 $\{l_i\}$ 的路径引用集,即计算 $s_i=USE(\{l_i\},s,l)$,其中 $s=USE(CC_l)$.若 $l_i\in L$,则 $s_i=USE(\{l_i\},s,l)\cup USE(CC_{l_i})$.当 $\exists l_k(l_k\in CP\wedge\sigma(l_k,l_i)=true)$ 时,如果 $DEF(P[l_k,l_i))\cap s_i\neq\varnothing$,根据定理3可知 $l_k\in VDC_l$,$l_k\rightarrow CE_l$.按此步骤继续分析 l_{i-1}.

对于任意 $l_j\in\{l_1,l_2,\cdots,l_{i-1}\}$,根据如下方法分析:

(1)$l_j\in CP$,(i)若 $l_j\in CE_l$,更新路径引用集 $s_j=s_{j+1}\cup USE(l_j)$;(ii)若 $l_j\notin CE_l\wedge\exists l_k(l_k\in P[l_j,l]\wedge\sigma(l_j,l_k)=true)$,则 $l_j\in RC_l$,根据定理2,$l_j\rightarrow CE_l$,并更新路径引用集 $s_j=s_{j+1}\cup USE(l_j)$.若 $l_j\in L$,更新路径引用集 $s_j=s_j\cup USE(CC_{l_j})$.继续分析 l_{j-1}.

(2)$l_j\notin CP$,使用路径引用集的推导规则,计算 $\{l_j\}$ 的路径引用集,即 $s_j=USE(\{l_j\},s_{j+1},l)$,其中 s_{j+1} 在分析指令 l_{j+1} 时计算出.若 $l_j\in L$,更新路径引用集 $s_j=s_j\cup USE(CC_{l_j})$.当 $\exists l_k(l_k\in CP\wedge\sigma(l_k,l_j)=true)$ 时,如果 $DEF(P[l_k,l_j))\cap s_j\neq\varnothing$,根据定理5可知 $l_k\in VDC_l$,$l_k\rightarrow CE_l$.继续分析 l_{j-1}.

算法1是程序执行与检测表达式生成的算法框架.目前原型系统的实现对于分支路径采用动静结合方式判断相关的引用集合和定义集合,没有对 alias[21]问题进行分析.如果发现某分支语句的分析过程中,有不确定目标的 alias 操作,则将当前分支条件加入相应检测点的值依赖条件 VDC 中,因此可能会造成 VDC 偏大,但此方法不会失去准确性.实验结果表明,即使使用这种方式,计算出的需多路径分析的条件数量仍然小于 EXE 方式下的相应条件数量.在今后的工作中,可以使用更精确的方法对

alias 问题进行分析，进一步减少分支数目.

算法 1. 程序 P 执行与检测表达式生成算法.

```
Input：程序 P
 1. stack Path ←∅；stack CPath ←∅；
 2. while (P 未结束) {
 3.   l←当前执行语句；对 l 做污点传播处理；
          Path.push(l)；
 4.   if (l 是分支语句∧l --DD--> Input) {CPath.push(l)；}
 5.   if (l∈L) {//L 是检测点集合
 6.     CE_l←检测语句 CC_l；l_i←Path.pop()；
 7.     c_i←CPath.pop()；s_i=USE(CC_l)；
 8.     while (Path≠∅) {
 9.       l_i←Path.pop()；
10.       if (l_i≠c_i) {
11.         s_i=USE({l_i},s_{i+1},l)；
12.         if (l_i∈L)s_i=s_i∪USE(CC_{li})；
13.         if (∃l_k(l_k∈CPath∧σ(l_k,l_i)=true∧
              DEF(P[l_k,l_i))∩s_i≠∅)){
14.           CE_l=CE_l∧l_k；
15.         }
16.       } else{
17.         if (l_i∈CE_l){s_i=s_{i+1}∪USE(l_i)；}
18.         else{
19.           if (l --WCD--> l_i){CE_l=CE_l∧l_i；s_i=
              s_{i+1}∪USE(l_i)；}
20.         }
21.         if (l_i∈L)s_i=s_i∪USE(CC_{l_i})；
22.         if (l_i∈CE_l) 在 c_i 处进行多路径分析；
23.         if (CPath=∅){break}
24.         else {c_i←CPath.pop()；}
25.       }
26.     }
27.     输出 CE_l；
28.   }
29. }
```

在程序循环时，路径减少显得尤为明显. 下例是在循环程序的示例，对比了 EXE 方法和延后方法的不同处理方式.

例 1. 检测点与循环.

```
   int m=input；  //表示 m 依赖于外部输入
   int n=input；  //假设 n<20
   int a[20]；int k=0；
   if(n<20){
i：  for (int i=0；i<n；i++){
        k=i；//没有定义 m,但是定义了 k
     }
j：  a[m]=k；//检测语句,检测条件是 0≤m<20
   }
```

对于以上片段，假设检测点是 j，检测条件是 $0\le m<20$. 如果使用 EXE 的处理方式，在 i 处，需要分别对 $n=0,n=1,\cdots,n=19$ 进行多路径分析. 但是对于这种方式中的每一条路径，程序到达检测点 j 的检测条件均没有变化，仍然为 $0\le m<20$. 因此在 i 处对 n 进行多路径分析是没有必要的. 但使用延后分析方法，$i\notin RC_j \wedge i\notin VDC_j$，不需要在 i 处进行多路径分析，因此有效节省了计算资源.

5　实验与分析

为了测试本方法的有效性和性能，我们实现了延后策略多路径分析的原型系统. 原型系统由 4 部分组成，分别是加载模块、执行模块、污点传播模块和延后分析模块. 加载模块首先加载被分析的程序，再由执行模块执行，执行的基本单位为程序语句，每执行完一条语句后由污点传播模块进行污点源的识别与污点数据传播，之后进行延后策略的分析. 分析过程中使用缓存技术进行优化，当后续程序运行时使用到已分析过的结果时，直接取缓存结果即可. 整个系统使用 C++语言实现，约 1 万行代码. 计算机硬件使用 Dell Optiplex 360 计算机，配以 Intel Core 2 Duo CPU E7300 2.66GHz CPU，2GB 内存，操作系统使用 Windows XP SP3.

多路径研究的意义在于其进行软件安全漏洞检测、程序异常行为分析方面的贡献. 因此我们使用实际案例数据测试基于延后方法进行动态多路径分析的有效性. Win32. NetSky 是 2004 年传播最广泛的蠕虫之一，它在 Windows 环境下以邮件方式进行传播. 在我们的分析过程中，将 GetLocalTime 返回值作为污点数据，并以此为污点源进行分析. 使用随机的日期，程序流程以图 2 中最上一层的调用序列到达 0x100025EE. 此时路径上的分支语句有 0x100025A3、0x100025AD、0x100025BA、0x100025C3、0x100025CB、0x100025D3、0x100025DC，其中 0x100025AD 与污点数据无关. EXE[2] 方法与 Moser[14] 方法应用广泛，是目前国际上最有代表性的可执行程序多路径分析方法之一. 由于 Moser 方法原理与 EXE 方法类似，实验时仅将延后方法与 EXE 方法作对比. 使用 EXE 方法多路径分析时，需要在除 0x100025AD 外的其它分支语句处启用新的进程进行路径分析；若使用延后策略的方法，可以发现 0x100025A3、0x100025CB、0x100025D3、0x100025DC 不属于 0x100025EE 的运行条件或值依赖条件，因此不必在这些位置进行多路径扩展，仅需在 0x100025BA 和 0x100025C3 处进行多路径分析即可，提高了整体分析的效率. 实际分析时，在 0x100025BA 处进行多路径扩展将到达 NetSky 程序的攻击处 ATTACK.

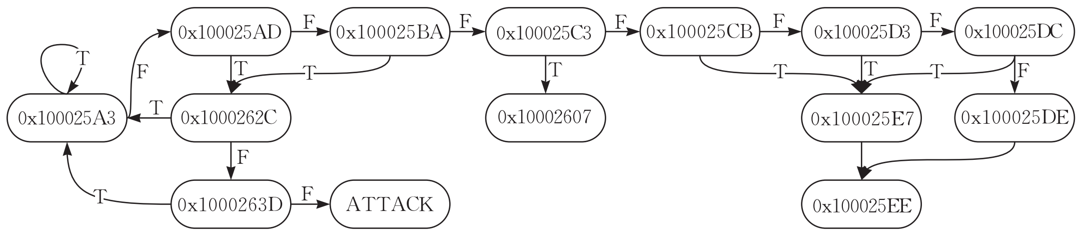


图 2 NetSky 多路径分析图

为了对比 EXE 方法和延后方法对于不同程序需要多路径分析的分支数量,我们对 7 款恶意软件进行测试. Perfect Keylogger 是一款记录键盘操作的软件;CodeRed、Nimda、Slammer、Maslan 为网络蠕虫;QQ Thief 和 ExploitIE 是黑客工具.

表 2 记录了各程序在运行过程中相应检测表达式中条件数量的平均值. 多路径分析针对不同的目标,检测点的选取是不同的,这里不考虑具体的分析目标,具体检测点的选取不是本论文的讨论重点. 目前国际上对程序检测点有多种选取方法[22],为了不失一般性,本文采用在程序中的随机检测点进行分析. 检测点的产生规则是在程序启动后,将每 100 条指令基本块为间隔作为一个程序检测点,各程序产生的检测点的数目随着程序的不同而不同. 多路径分析过程中,创建新进程的数量与条件数量成指数关系,因此条件数量在一定程度上反映多路径分析效率以及路径求解的效率. 在程序运行过程中,检测表达式中条件数量会不断变化,因此如果简单地对每个检测表达式中的条件数量进行平均,将会丧失结果的准确性. 这里以原始方法(Origin)为基础,对每个检测点在各种方法下的条件表达式数量进行归一化,最后取平均值,比较 EXE 方法和延后策略(LA)对表达式长度的缩减. 其中原始方法指对路径条件不做任何判断均进行记录.

表 2 3 种多路径方法的比较

程序名称	检测点数量	EXE/Origin	LA/Origin	(LA/EXE)/%
Perfect Keylogger	3193	0.1666	0.0256	15.37
CodeRed	1577	0.1767	0.0381	21.56
Nimda	1505	0.0709	0.0080	11.28
Slammer	1363	0.1562	0.0336	21.51
Maslan	1234	0.0572	0.0014	2.45
QQ Thief	1231	0.0171	0.0083	48.54
ExploitIE	1223	0.0217	0.0087	40.09

表 2 中第 2 列表示分析过程中,程序检测点总数;Origin 表示原始方法,这也是归一化处理的基础;第 3 列为使用 EXE 方法缩减后的条件表达式长度相对于 Origin 方法所占的比例;第 4 列是使用延后策略的条件数量相对于 Origin 方法所占的比例;最后一列以百分制表示延后策略检测表达式条件数量占 EXE 方法的比例. 从表 1 中可以看出,EXE 方法虽然使用了污点传播进行处理,但仍存在大量无用条件. 使用延后策略的方法能够减少大量的无用分支语句,简化了检测表达式,减少了分支的数量,其条件数量平均仅占 EXE 方法的 22.97%.

表 2 从整体上对比了 EXE 方法和延后方法之间对不同程序检测点中条件数量的差异,但并没有显示出在分析过程中,检测点条件数量的变化情况. 图 3 以 Perfect Keylogger 为例,从程序随时间执行的角度加以说明. 图中横轴记录的是程序顺序流程中随机选取的不同检测点,纵轴表示检测表达式中条件的数目. 由图中可知,使用 EXE 方法,程序的条件检测点数目随程序的执行几乎呈线性增长,当程序执行流程越长,检测表达式中条件数量越多. 值得注意的是,条件数量每增加 1,产生的进程数量并不是增加 1,而是在原有基础上增加 1 倍,即进程数量与条件表达式的数量之间为指数关系. 因此这也是 EXE 方法难以分析大型程序的原因. 从图中看出,采用延后策略的方法,检测点条件数量呈锯齿形分布. 其原因是检测表达式中条件数量与程序的执行过程的长短并无直接关系,仅与变量之间的依赖关系和运行条件相关. 由于程序中外界变量仅影响部分程序语句,随着程序的执行,此外界变量可能不

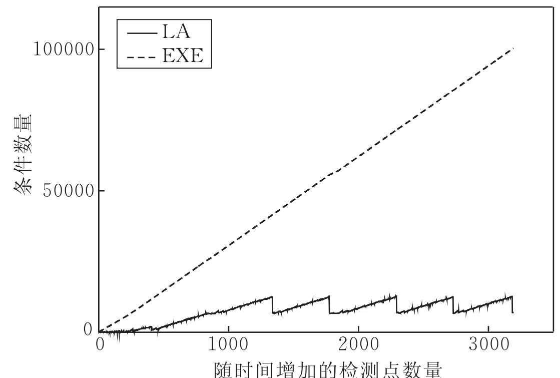


图 3 Perfect Keylogger 中 EXE 方法和 LA 方法对于不同检测点条件数量的时间流程图

被继续使用，因此后续阶段的检测表达式中的检测条件与前一阶段可能并不相关，这也是图中曲线呈锯齿型分布的原因. 因此，本方法更适用于大型程序.

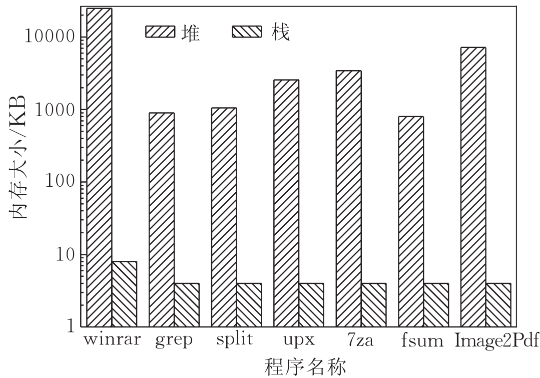


图 4　分析过程中各程序影子内存数量

由于影子内存需与真实内存保持对应关系，因此在分析过程中可能会占用较大空间. 在本原型系统中，采用实时记录的影子内存方法. 当有新的内存空间被申请时，增加相应的影子内存；当目标进程对应的内存空间被释放时，影子内存也随之释放. 所以原型系统不需要使用整个内存空间作为影子内存，而仅仅需要动态维护内存堆栈即可，更大程度地减少了原型系统的内存空间占用量. 为了确认分析过程中内存占用情况，我们对影子内存进行了分析. 图 4 是在分析过程中各程序影子内存占用内存空间的最大值(单位是 KB). 从图中可知，本原型系统使用影子内存机制所占的内存空间(包括堆栈)不超过 15MB，平均内存占用率仅为 6.47MB. 因此，本系统具有较高的空间利用率.

为了测试延后方法中检测点数量对检测表达式条件筛选的效率影响，表 3 对比了在延后策略下，当检测点数量发生变化时，筛选表达式条件所需的时间. 仍然对 7 款恶意软件进行测试，检测点数量选取 0、10、100、1000 个，时间单位为 s. 从表中可知，使用延后方法分析 10 个、100 个检测点，相对于 0 个检测点并没有在时间效率上减小许多. 即使分析 1000 个检测点，增加的额外检测时间也与检测点数量几乎呈线性关系. 因此延后方法具有一定的时间效率. 从表中可以看出，对于部分软件，例如 Nimda、QQ Thief 等，即使在 1000 个检测点时分析时间仍然较少. 这是由于算法在分析过程中，对部分中间过程的运算结果(例如后必经结点的计算结果)进行保留，并在下次运算时直接使用造成的.

表 3　各种方法运行时间比较

程序名称	运行时间/s			
	LA(0)	LA(10)	LA(100)	LA(1000)
Perfect Keylogger	191	195	208	313
CodeRed	53	56	68	149
Nimda	118	130	134	162
Slammer	51	56	64	217
Maslan	60	61	65	106
QQ Thief	574	585	636	759
ExploitIE	287	292	328	579

6　结　论

现有多路径分析方法主要存在路径分析不全面、分析效率较低、分析过程中存在大量无用路径等问题. 部分工作仅从检测表达式本身对路径表达式进行优化求解，并没有结合程序检测点和路径条件进行深入简化. 本文针对以上问题，提出了弱控制依赖和路径引用集的概念以及相应的计算规则，并在此基础上提出了一种延后策略的可执行程序动态多路径分析方法. 在程序分析过程中，不立即进行路径表达式求解和多进程构造，而是记录当前的分支条件. 当遇到程序检测点时，有针对性地根据检测点条件进行路径筛选，从语义上进行路径表达式简化，减少需要分析的路径数量，简化检测表达式. 本文实现了一套原型系统，并对 Perfect Keylogger 等 7 款恶意软件进行分析实验，结果表明本方法提高了多路径分析效率和准确性.

实验中发现，在循环处理过程中，会产生大量的条件表达式，如何对这类条件进行有效简化是未来的工作方向之一. 同时，结合现有漏洞检测方法，对程序检测点进行更细粒度的选择，更多地支持污点分析的指令数量和完善系统调用信息，进行程序安全性分析也是本文未来的工作方向.

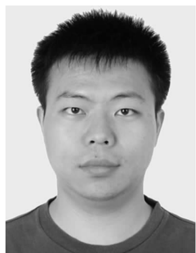
参　考　文　献

[1] Godefroid P, Levin M Y, Molnar D. Automated whitebox fuzz testing//Proceedings of the Network and Distributed System Security Symposium. San Diego, CA, 2008

[2] Cadar C, Ganesh V, Pawlowski P M, Dill D L, Engler D R. EXE: Automatically generating inputs of death//Proceedings of the 13th ACM Conference on Computer and Communications Security. Alexandria, VA, USA, 2006: 322-335

[3] Linn C, Debray S. Obfuscation of executable code to improve resistance to static disassembly//Proceedings of the 10th ACM Conference on Computer and Communications Security. Washington DC, USA, 2003: 290-299

[4] Boonstoppel P, Cadar C, Engler D. RWset: Attacking path explosion in constraint-based test generation//Proceedings of the 14th International Conference. TACAS, Budapest, Hungary, 2008: 351-366
[5] Xie T, Tillmann N, de Halleux J, Schulte W. Fitness-guided path exploration in dynamic symbolic execution. Microsoft: MSR-TR-2008-123, 2008
[6] Balakrishnan G, Reps T. Analyzing memory accesses in x86 executables//Proceedings of the 13th International Conference on Compiler Construction. Barcelona, Spain, 2004: 5-23
[7] Balakrishnan G, Gruian R, Reps T, Teitelbaum T. Codesurfer/x86—A platform for analyzing x86 executables//Proceedings of the 14th International Conference on Compiler Construction. Edinburgh, Scotland, 2005: 250-254
[8] Cova M, Felmetsger V, Banks G, Vigna G. Static detection of vulnerabilities in x86 executables//Proceedings of the Annual Computer Security Applications Conference (ACSAC). Miami, FL,USA, 2006: 269-278
[9] Anand S, Orso A, Harrold M J. Type-dependence analysis and program transformation for symbolic execution//Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems. Braga, Portugal, 2007: 117-133
[10] Xia Yi-Min, Luo Jun, Zhang Min-Xuan. Detecting out-of-bounds accesses with conditional range constraint. Journal of Computer Research and Development, 2006, 43(10): 1760-1766(in Chinese)
(夏一民,罗军,张民选.基于条件范围约束的越界访问检测方法.计算机研究与发展,2006,43(10):1760-1766)
[11] Weiser M. Program slicing. IEEE Transactions on Software Engineering, 1984, 10(4): 352-357
[12] Costa M, Castro M, Zhou L, Zhang L, Peinado M. Bouncer: Securing software by blocking bad input//Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles. Stevenson, Washington, USA, 2007: 117-130
[13] Ghosh A K, O'Connor T, McGraw G. An automated approach for identifying potential vulnerabilities in software//Proceedings of the IEEE Symposium on Security and Privacy. Oakland, California, 1998: 104-114
[14] Moser A, Kruegel C, Kirda E. Exploring multiple execution paths for malware analysis//Proceedings of the IEEE Symposium on Security and Privacy. Oakland, California, 2007: 231-245
[15] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. Tucson, Arizona, 2008: 206-215
[16] Sofokleous A A, Andreou A S. Automatic, evolutionary test data generation for dynamic software testing. The Journal of Systems & Software, 2008, 81(11): 1883-1898
[17] Burnim J, Sen K. Heuristics for scalable dynamic test generation//Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 2008: 443-446
[18] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software//Proceedings of the 12th Annual Network and Distributed System Security Symposium. San Diego, California, 2005: 134-150
[19] Brumley D. Analysis and defense of vulnerabilities in binary code [Ph. D. dissertation]. Stanford University, 450 Serra Mall, California, 2008
[20] Gulwani S, Srivastava S, Venkatesan R. Program analysis as constraint solving//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. Tucson, Arizona, 2008: 281-292
[21] Debray S, Muth R, Weippert M. Alias analysis of executable code//Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego, California, United States, 1998: 12-24
[22] Wang Li, Yang Xue-Jun, Wang Ji, Luo Yu. Automatedly checking function execution context of kernel programs in operation systems. Journal of Software, 2007, 18(4): 1056-1067(in Chinese)
(汪黎,杨学军,王戟,罗宇.操作系统内核程序函数执行上下文的自动检验.软件学报,2007,18(4):1056-1067)



CHEN Kai, Ph. D. candidate. His research interests include information security, vulnerability analysis and detection, malware analysis and prevention.

FENG Deng-Guo, born in 1965, Ph. D., professor. His research interests include information security, cryptography.

SU Pu-Rui, born in 1976, Ph. D., associate professor. His research interests focus on information security.

Background

Software testing is a crucial part of software development and a perennial problem as well. Manual testing is not only a labor job but a tedious job. Moreover, the results cannot cover all program paths. Thus, even software is checked

by manual testing, it cannot be guaranteed that no fault is in the software. Random testing is proposed to solve the efficiency problem in manual testing. However, it cannot explore all the program paths. For example, satisfying a 32-bit equality in a branch condition requires correctly guessing one value out of four billion possibilities. Then people use symbolic input instead of concrete value to run a program. When a branch is met and the variable in the branch conditions is dependent on the input, the symbolic input can be easily changed to explore the two possible paths of the branch. However, there are a lot of branches in a real program. So this method cannot be easily applied to these programs. In this way, some paths cannot be explored and some redundant paths are explored. Some heuristic methods are proposed to solve this problem but they cannot handle them well because they do not touch key problem. This paper proposes a lazy method to analyze the path constraints according to a chosen check point. It classifies path constraints into two types and chooses the necessary conditions that guarantee the program to run to the check point. The authors also implement a prototype and make some experiments. The results show that the method decreases the number of path conditions and increases the efficiency when exploring multiple paths. This work was supported by the National Natural Science Foundation of China under grant Nos. 60703076, 60970028 and the National High Technology Research and Development Program (863 Program) of China under grant Nos. 2006AA01Z412, 2007AA01Z451, 2007AA01Z475, 2007AA01Z465, 2007AA01A414.