一种抵抗符号执行的路径分支混淆技术

王 志 贾春福 刘伟杰 王晓初 张海宁 于晓旭 陈 ▮
(南开大学 计算机与控制工程学院,天津 300071)

摘 要: 程序在动态执行过程中泄露了大量的路径分支信息,这些路径分支信息是其内部逻辑关系的二进制表示.符号执行技术可以自动地收集并推理程序执行过程所泄露的路径信息,可用于逆向工程并可削弱代码混淆的保护强度.哈希函数可以有效保护基于等于关系的路径分支信息,但是难以保护基于上下边界判断的不等关系的路径分支信息.将保留前缀算法与哈希函数相结合提出了一种新的路径分支混淆技术,将符号执行推理路径分支信息的难度等价到逆向推理哈希函数的难度.该路径分支混淆方法在SPECint-2006程序测试集上进行了实验,试验结果表明该混淆方法能有效保护程序路径分支信息,具有实用性.

关键词: 代码混淆; 符号执行; 哈希函数; 保留前缀加密
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2015)05-0870-09
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2015.05.006

Branch Obfuscation to Combat Symbolic Execution

WANG Zhi, JIA Chun-fu, LIU Wei-jie, WANG Xiao-chu, ZHANG Hai-ning, YU Xiao-xu, CHEN Zhe
(*College of Computer and Control Engineering, Nankai University, Tianjin* 300071, *China*)

Abstract: At run time, a large number of program branching information is leaked. Branching information is the binary representation of program internal logic. Symbolic execution could automatically collect and reason about the leaked branch information, which could be used for reverse engineering and weaken the strength of code obfuscation. Hash function can effectively safeguard equal branch conditions, but it can't be used to protect branching information containing unequal trigger conditions, such as greater than or less than. In this paper, a new branch obfuscation approach combining prefix-preserving algorithm and hash function, which extends the protection scope of hash function. The strength and resilience of the branch obfuscation are discussed. This branch obfuscation approach has been tested on 7 programs from the SPECint-2006 benchmark suite, and the experimental results show that this approach could effectively mitigate branch information leaking, yet practical in terms of performance.

Key words: code obfuscation; symbolic execution; Hash function; prefix-preserving encryption

1 引言

代码混淆技术可以增加代码逆向分析和推理的难度和时间,是当前信息安全领域的研究热点之一.逆向工程的威胁主要来自受控的主机环境,如图1所示.Falcarin等人[1]将图1所示的安全威胁模型称为MATE(Man At The End)攻击,该攻击有多种表现形式,其中主要包括:逆向工程、代码盗用、恶意篡改和盗版.

MATE攻击的基础是逆向工程,逆向工程是代码盗用、恶意篡改和盗版的前提条件.逆向工程存在的根本原因有两点:(1)二进制代码中蕴含着大量的路径信息,例如算法、软件架构和实现方式等;(2)程序在受控的计算环境中缺乏自我保护能力,不能对用户的恶意行为进行有效约束.当程序进入攻击者的计算机,攻击者可以对程序进行全面地监控和逆向工程,例如,攻击者可以使用反汇编和反编译工具分析程序的代码,利用动态调试和跟踪工具监控程序的行为.该逆向技术也被杀毒软件公司用于恶意代码分析.商业软件联盟(BSA)的研究报告显示,全球个人计算机上安装的软件中,有43%的软件存在着版权问题[2],MATE攻击在2010年和2013年给全球软件产业造成了高达588亿美元[3]和627亿美元[2]的经济损失.

收稿日期:2014-02-24;修回日期:2014-08-04;责任编辑:马兰英
基金项目:国家自然科学基金(No.61300242,No.61272423,No.60973141);国家"973"重点基础研究发展计划(No.2013CB834204);中央高校基本科研业务费专项资金(No.65121012);南开大学-腾讯联合项目

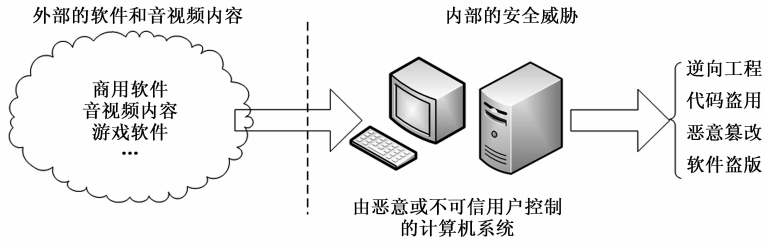

图1　恶意或不可信用户对软件知识产权的安全威胁

当前,随着符号执行[4]、动态污点分析[5]、形式化推理[6]和二进制代码插装[7]等程序分析技术,以及虚拟机和云计算技术在程序逆向工程领域的成功应用,逆向工程的自动化程度、分析的全面性、推理的准确性都得到了大幅提升,给代码混淆技术提出了更高的要求.

上述各种逆向工程新技术的基础是程序在执行过程中的路径分支信息泄露问题.执行过程中,所有的逻辑关系都通过 CPU 的条件跳转指令以路径分支的形式泄露,这些路径分支信息易于收集和逆向分析.路径分支信息泄露问题已经在软件测试、漏洞发掘、恶意代码分析、协议分析和代码复用等安全领域得到广泛利用.要缓解路径分支信息泄露问题,理想的保护模型是使程序正向执行和逆向分析的复杂度不对称,越接近单向执行,保护的强度就越高,但同时也要保证程序的功能性和执行效率.

本文的研究目标是在路径分支信息中引入哈希函数,使程序执行过程具有一定的单向性,增加逆向工程的难度.因为哈希函数不具有保序性,其应用范围受到了很大的约束,例如,哈希函数不能保护大于或小于关系,当变量 a 的值大于变量 b 的值($a > b$),$Hash(a) > Hash(b)$是不一定成立的.本文提出了将保留前缀算法与哈希函数相结合的混淆策略,利用保留前缀算法,将基于边界值比较的大于或小于关系转换成基于前缀集合匹配的等于关系,有效扩展哈希函数的保护范围.路径分支混淆后,攻击者利用路径分支信息逆向推理程序内部逻辑关系的难度将等价到逆向推理哈希函数的难度.本文路径分支混淆算法的路径分支混淆策略,开发了路径分支混淆原型系统,并在 SPECint-2006 测试程序集上对路径分支混淆强度和开销进行了测试.测试结果显示路径分支混淆后,程序路径约束关系的复杂度增加了 5 到 5000 倍,基于符号执行的逆向工程无法求解混淆后的路径约束关系,混淆后程序平均增加的执行时间和文件长度分别只有 1 到 2 秒和不到 1000 个字节,具有实用性.

2　二进制代码的路径分支信息泄露问题

二进制代码比高级语言代码更难于分析,因为二进制代码中缺少高级语言的类型信息、结构信息、控制流信息.在高级语言代码中,可以直接分析出变量的类型、占用的存储空间、生存期、作用域等信息,而在二进制代码中,得到的仅仅是内存地址和寄存器的读写信息,难以获取隐含的类型信息.在二进制代码中,数据和指令交错地存储在线性的内存空间中,甚至在执行过程中,数据和指令还可以相互转换,缺少清晰的组织结构,如高级语言的结构、类、枚举、数组等结构信息.二进制代码中存在大量的间接跳转指令,加上数据和指令之间存在着动态的转换,其控制流分析的难度高于高级语言的控制流分析.虽然二进制代码和高级语言代码相比,其泄露的信息量很少,但是这并不意味着二进制代码很安全:(1)二进制代码在执行过程中会泄露类型信息和结构信息.例如,与操作系统的交互过程中,所调用的系统函数都有其标准的原型定义,通过分析系统函数的调用过程,可以推理出二进制代码所缺乏的数据类型、存储空间、语意、结构等信息;(2)二进制代码的控制流信息也存在着严重的泄露问题,包括有分支的控制流转移信息和无分支的控制流转移信息.对于保护无分支的控制流转移信息已经有了大量的研究,例如控制流退化(Degeneration of Control Flow)技术[8],分支反转(Branch Flipping)技术[9]、分支函数(Branch Functions)技术[9]、不透明谓词(Opaque Predicates)[10]、程序碎片[11]和跳转表欺骗(Jump Table Spoofing)技术等.

但是对于有分支控制流信息的保护基本上处于空白阶段,其难点是如何既隐藏控制流的分支条件,又能实现准确地控制流转移.有分支的控制流信息也称为路径分支信息,如图 2 所示,其由三部分组成:分支点、分支条件和分支入口点.分支点是产生路径分支的内存地址,它是静态分析中划分基本块(Basic Block)的重要依据.分支条件就是路径选择的约束关系,它是程序逻辑关系的二进制表示,其中既包括代码执行的前置和后置条件,也包括数据的边界信息等.分支入口点是当路径约束关系满足时,CPU 下一条指令的内存地址,使线性存储在内存中的指令能够实现非线性的控制流转移.路径分支信息易于被符号执行技术收集、分析和推理,如图 3 所示.路径分支信息的保护需要从两个方面入手:(1)增强抵抗动态分析的能力,增加路径分支信息的收集难度,例如,用非条件跳转指令的副作用替

换条件跳转指令,隐藏程序的路径分支信息;(2)增强抵抗静态分析的能力,增加路径分支信息的分析难度,例如,本文所提出的基于保留前缀算法和哈希函数的混淆技术.

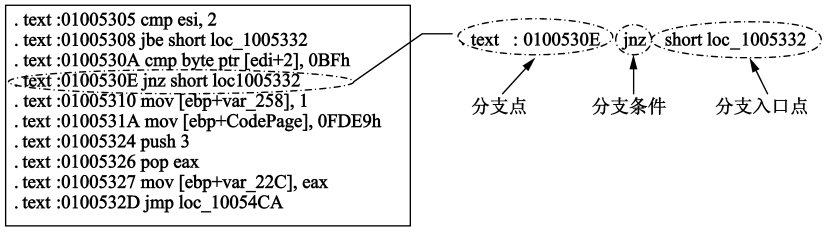

图2 路径分支信息的泄漏

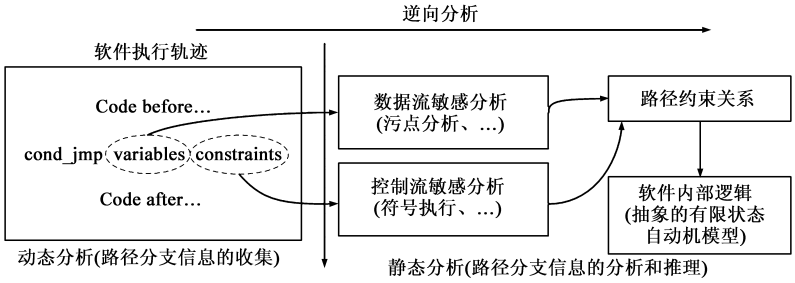

图3 基于路径分支信息泄漏问题的逆向工程

3 哈希函数和保留前缀算法

3.1 哈希函数

哈希函数近似于单向函数,其逆向运算的复杂度远远高于其正向计算的复杂度,定义如下:

函数 f 如果能具有以下两个特点,则称 f 为单向函数:(1)对于属于函数 f 定义域的任一值 x,都可以轻易地计算出 $y = f(x)$;(2)对于大部分属于函数 f 值域的任一值 y,很难在计算上求出满足 $f(x) = y$ 的值 x.

哈希函数可以把任意长度的输入数据打乱并混合,创建一个格式固定的信息摘要,称为哈希值.哈希函数具有近似的单向性.在代码混淆中,哈希函数只用于保护基于等于关系的路径分支条件,无法对大于或小于关系进行保护.

3.2 前缀算法

前缀算法能够从一个给定数据空间中找出其前缀特征,即前缀集合.前缀集合具有确定性,能够匹配该前缀集合的数据一定属于该数据空间,否则将一定不属于该数据空间.这样,对一个数据空间的描述既可以利用上下边界来表示,也可以使用该数据空间的前缀特征来表示.

算法 1 给出了计算给定区间前缀集合的伪代码.算法的输入是 n 位二进制数区间的起始值 $a_1a_2\cdots a_n$ 和结束值 $b_1b_2\cdots b_n$,二进制数的左边是高位,右边是低位,输出是该二进制区间的前缀集合.该算法是一个递归算法,首先,找到二进制区间的起始值和结束值的关键位,即从左到右数第一个不相等的比特位,如果没有找到关键位,则说明该二进制区间的上下边界是重合的,它的前缀特征就是该重合的点;然后,如果起始值的关键位 k 右边的各位都是 0,而且结束值的关键位 k 右边的各位都是 1,则该二进制区间的前缀就是关键位 k 之前的相似部分;接着,如果前两个条件都不满足,则将起始值和结束值的关键位右边的各位分别作为新的边界输入,递归调用前缀算法去分别计算两个子空间的前缀特征;最后,将子空间的前缀特征与关键位 k 左边的相似位进行整合,返回二进制区间的前缀集合.

算法 1 求解二进制数区间对应前缀集合的算法

输入:$a_1a_2\cdots a_n$// 一个二进制数区间的起始值

$b_1b_2\cdots b_n$// 一个二进制数区间的结束值

输出:Set: *Prefix* // 二进制数区间的前缀集合

```
Prefix Search_Prefix(a1a2…an, b1b2…bn){
//从第一位开始,找到关键位 k, ak < bk
  for (int k = 1;(k <= n) && (ak == bk);k++)
    if (k == (n+1)) return {a1a2…an};
  if ((akak+1…an == 00…0) && (bkbk+1…bn == 11…1)){
    if (k == 1)
      return {*};
    else
      return {a1a2…ak-1};
  }
      Set_PrefixA = Search_Prefix(ak+1ak+2…an, 11…1);
      Set_PrefixB = Search_Prefix(00…0, bk+1bk+2…bn)
  return{a1a2..ak-10 + Set_PrefixA, a1a2…ak-11 + Set_PrefixB};
}
```

前缀集合的大小对前缀算法的应用有很大的影

响,如果前缀集合过大,则前缀匹配过程会大幅增加程序的体积和执行开销.在找到一个数据区间的前缀集合后,前缀的数量会不会和数据区间的大小成正比,即数据区间越大前缀集合是否就越大?

前缀集合已经被证明有如下两个特点:(1)对于任意一个二进制整数区间$[a_1a_2\cdots a_n, b_1b_2\cdots b_n](n\geqslant 2)$,前缀的个数 $p\leqslant 2n-2$;(2)对于长度为 n 的二进制整数,考虑所有可能的区间$[a_1a_2\cdots a_n, b_1b_2\cdots b_n](n\geqslant 2)$,前缀平均数为$\frac{(n-2)2^{2n-1}+(n+1)2^n+1}{2^{2n-1}+2^{n-1}}$.当 n 很大时,前缀的平均个数趋近于 $n-2$.

由前缀集合的特点可见,即使对于一个很大的区间,前缀的个数也是有限的,前缀个数的最大值只同二进制数的位数 n 相关,前缀数量的最大值为 $2n-2$ 个,平均值约为 $n-2$ 个.对于二进制代码中常用的32位整数,无论整数区间的大小如何,前缀的个数最多为62个,平均约为30个.

3.3　基于保留前缀的哈希函数

保留前缀加密最早应用于IP地址匿名化[12]和外包数据库中密文数据的直接范围查询[13].保留前缀加密算法的定义:假设两个 n 位的二进制整数 a 和 $b(a=a_1a_2\cdots a_n, b=b_1b_2\cdots b_n)$的前 k 位都相同,而 $k+1$ 位不同.如果一个加密函数 E_p,能够保证加密后的 $E_p(a)$和 $E_p(b)$的前 k 位仍然相同而第 $k+1$ 位仍然不同,则称加密函数 E_p是保留前缀的.

保留前缀加密的一般形式定义如下:给定一个明文 $a=a_1a_2\cdots a_n$,和函数 f,f 是任意的可以接收输入 a 的函数,密文 $a'=a_1'a_2'\cdots a_n'$通过如下的方式计算得到:(1)给定 a_1'为一个常数(0或1);(2)$a_i'=a_i\oplus f(a_1a_2\cdots a_{i-1}), i=1,2,\cdots,n$.

假设两个 n 位二进制整数 a 和 $b(a=a_1a_2\cdots a_n, b=b_1b_2\cdots b_n)$的前 $k(k<n)$位相同,第 $k+1$ 位不同.当 $i\leqslant k$ 时,a 和 b 的前 k 位相同,所以 $a_1a_2\cdots a_{i-1}$和 $b_1b_2\cdots b_{i-1}$相同.因为函数的输入相同,$f(a_1a_2\cdots a_{i-1})$和 $f(b_1b_2\cdots b_{i-1})$的输出值也相等,而且,a_i与 b_i也是相等的,所以,$a_i\oplus f(a_1a_2\cdots a_{i-1})$与 $b_i\oplus f(b_1b_2\cdots b_{i-1})$的计算结果相同,即 a_i'等于 b_i'.

当 $i=k+1$ 时,因为 $a_1a_2\cdots a_{i-1}$和 $b_1b_2\cdots b_{i-1}$是相等的,所以 $f(a_1a_2\cdots a_{i-1})$和 $f(b_1b_2\cdots b_{i-1})$的值也是相等的.但是,a_i与 b_i不相等,因此,$a_i\oplus f(a_1a_2\cdots a_{i-1})$与 $b_i\oplus f(b_1b_2\cdots b_{i-1})$的计算结果肯定是不相等的,即 a_i'不等于 b_i'.

由上面的证明可知,该方案符合保留前缀加密算法的定义,而且,函数 f 是可以变换的,因此,本文将哈希函数通过函数 f 引入到保留前缀加密算法中.

4　基于保留前缀和哈希函数的路径分支混淆

哈希函数只能用来保护等于关系的路径分支条件,不能保护大于或小于关系的路径分支条件.当路径分支条件是判断某个变量值是否属于一个区间时,Sharif 等人[14]提出可以将这种不等关系转换成一组相等关系的集合,将该变量与区间内所有值一一进行比较,然后再使用哈希函数对每个比较过程进行保护.当区间较大时,该方法的时间开销和空间开销将非常大,例如,一个简单的路径分支条件 $32\leqslant x\leqslant 111$ 将会被替换成80个基于等于关系的路径分支条件.每个等于关系对应一个MD5值,每个MD5值占16个字节,因此该路径分支条件最少要占用 $80\times 16=1280$ 个字节,且输入变量 x 的MD5值要跟集合中的MD5值进行一一比较,增加了程序的空间和时间开销.

本文引入了保留前缀算法,将一个区间转换为一个前缀集合,把基于大于和小于关系的上下边界判断,转换为基于等于关系的前缀匹配判断.例如,区间[32,111]的前缀集合是{001＊＊＊＊＊,010＊＊＊＊＊,0110＊＊＊＊},因此存储空间的开销从1280字节缩减到 $3\times 16=48$ 字节,最多的比较次数从80减少到3次,优化了路径选择过程的时间和空间开销.但是,保留前缀算法是可逆的,如果攻击者获得了前缀集合,就能逆向推理出前缀集合所对应的数据区间.因此,还需要对保留前缀算法进行加密,即引入哈希函数.

保留前缀算法和哈希函数的基本思想是把哈希函数与保留前缀算法相结合,增强保留前缀算法的单向性,如图4所示.保留前缀算法与哈希函数相结合,$f(a_1a_2\cdots a_{i-1})$变成了 $\Gamma(Hash(a_1a_2\cdots a_{i-1}))$,其中 Γ 表示取某一有效位.上述加密方法综合了哈希函数的单向性和保留前缀算法的高效性,实现了对路径分支信息全面、高效的保护.表1是对哈希函数、保留前缀算法和本文的代码混淆算法的对比.第一行和第二行是比较次数的最高值和平均值,第三行是算法的单向性.

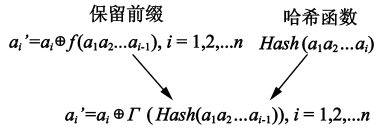

图4　路径分支混淆算法

表1　三种算法的属性对比(n 为二进制数的位数)

	哈希函数	保留前缀	分支混淆
最高值	2^n-1	$2(n-1)$	$2(n-1)$
平均值	2^n	$n-2$	$n-2$
单向性	√	×	√

哈希函数的比较次数与区间的大小成正比;本文的代码混淆算法的比较次数与输入的二进制数的位数成正比,因此,其时间开销和空间开销远远小于哈希函数,但是,保留前缀加密不具有单向性.路径分支混淆算法结合了前两种算法的单向性和低开销的优点.

本文的路径分支混淆过程分为4个步骤,如图5所示.首先,从二进制代码中定位大于或小于关系的路径分支条件,例如ja,jb,jge等条件跳转指令,这些路径分支条件是哈希函数不能直接进行保护的;接着,计算能够满足这些不等关系的数值区间,其中MAX和MIN分别表示 x 取值范围中的最大值和最小值;然后,利用保留前缀算法找到数值区间的前缀集合;最后,将路径分支条件替换成前缀匹配操作.

5 安全性分析和性能测试

5.1 安全性分析

保护程序的路径分支条件,要求即使攻击者能够进行多次用例测试,也无法根据运行结果准确推断分支条件.本文通过将路径分支混淆算法作用于分支条件对应的数值区间,形成一个或多个前缀集合,从而将输入数值与分支条件的大小关系比较转化为对其前缀的判定,以达到保护路径分支条件的目的.Amanatidis等人[15]定义了理想的保留前缀加密方案的安全目标,Xiao等人[16]证明了理想的保留前缀加密方案符合安全目标.本文在保留前缀加密方案的基础上引入了哈希函数,因此,本文的路径分支混淆方案与理想保留前缀函数具有最大相似性,可以保证在满足不改变程序执行路径的前提下,能够最大限度地保护分支条件信息.

路径分支混淆算法是一种抵抗逆向分析的技术,增加逆向分析工具通过路径分支结果(动态逆向分析)和路径分支代码(静态逆向分析)逆向推理路径分支约束条件以及程序内部逻辑关系的难度.

在逆向分析过程中,攻击者可见的路径信息有:保留前缀的加密算法和包含前缀信息的哈希值.其中,包含前缀信息的哈希值中,前缀的长度不是固定不变的,例如32位的哈希值,前缀的长度有可能是3位,也有可能是32位.哈希值中前缀位的长度取决于能够触发分支条件的输入数据的取值范围和加密算法,不同的取值范围和不同的加密算法,前缀位的长度有可能不同.

不仅仅哈希值中的前缀值的长度是变化的,而且通过哈希值的前缀信息逆向推理该前缀对应的输入数据的取值范围的难度等同于逆向分析哈希值的难度.Sharif的混淆策略[14]是本文方法的一种极限情况,Sharif的方法中前缀值的长度正好等于哈希值的长度,其保护对象是数据空间中的一个点.本文的方法,前缀值的长度与哈希值的长度可以不相等,因此保护对象不仅仅是空间中的一个点,还而且可以是空间中的一个区间.路径分支混淆的前提条件是不能改变程序的原始语义,因此对于黑盒攻击,例如穷举攻击,混淆算法的保护强度还取决于程序原始的语义,及要保护的输入数据的取值范围与整个数据空间的比值.随着被保护的取值范围占数据空间比例的增长,会相应提升穷举攻击的成功率.

5.2 抗符号执行分析效果

上一小节介绍了路径分支混淆算法的安全性,本节将使用二进制代码分析平台BitBlaze来测试路径分支混淆算法的抗符号执行分析效果.首先,设计了一个简单端口检测程序,如图6所示,该程序只有一个判断条件,用于判断端口号是否在[32,111]这个数值区间内.BitBlaze可以详细地记录图6中样本的二进制执行轨迹,并准确地计算出合法端口号的范围为[32,111].实验中,路径分支混淆使用哈希函数MD5作为单向函数.表2列出了程序混淆前后程序复杂度的变化,其中包括轨迹中的指令数、跳转指令数、条件跳转指令数、STP文件大小、约束关系数量和约束方程节点数量.

表2 路径分支混淆前后程序复杂度对比

测试程序	指令	跳转指令	条件跳转指令	STP文件	约束关系	节点
port_check	20338	3687	2672	506KB	49	2957
obf_port_check	2361212	286629	200319	1956KB	79	10595

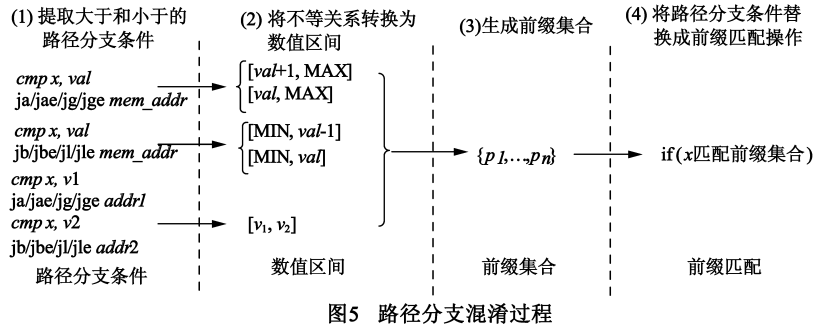

图5 路径分支混淆过程

```
void port_check (int port){
s1: if (port > = 32
        && port < 111){
s2: valid();
    }
s3: else{
s4: invalid();
    }
}
```

图6 简单的端口检测程序

从实验数据中可以发现混淆后的执行轨迹明显变长,指令数量增加了 200 多万条,增加了程序的执行开销,具体的执行开销分析将在 4.3 节中详细分析.轨迹中的跳转指令和条件跳转指令的数量反映了程序路径信息的复杂度,从表 2 中可知,路径分支混淆后的程序控制流明显复杂了,路径分支数比混淆前增加了大约 20 万个,即在程序的控制流图中新增了 20 万个节点.通过对程序执行轨迹的分析,BitBlaze 能够收集到程序的路径约束关系,并使用 STP 的语法格式记录在 STP 文件中.表 2 的数据显示,混淆后 STP 文件的大小增加了 3 倍左右,其中的约束关系数量也增加了 1 倍左右,因此,路径分支混淆后约束关系的复杂度明显增加.表 2 的最后一列是 STP 求解过程中所使用的节点数量,节点数直接反应了约束方程的复杂度,节点数越多约束方程越复杂,也就需要使用更多的内存空间和计算时间,数据显示路径分支混淆后的约束方程增加了 7000 多个节点,并且 STP 无法对约束关系进行有效的求解.

然后,我们将路径分支混淆算法在 SPECint-2006 程序测试集上进行了实验,选取了测试集中 7 个 *C* 语言程序进行测试:specrand、sjeng、mcf、lbm、hmmer、bzip2、h264ref.每个测试程序只选择了一个路径分支条件进行混淆,混淆过程分别使用哈希函数 CRC32、MD5、SHA1、SHA256 和 SHA512 作为单向函数.实验结果如图 7 所示,横坐标是 SPECint-2006 的测试样本,纵坐标是混淆后路径约束关系所增加的倍数,变化最多的是使用 SHA512 算法混淆的 specrand,混淆前后约束关系增加了 5000 多倍,最少的是 CRC32 算法混淆的 bzip2,混淆后约束关系增加了 5 倍,约束求解工具 STP 无法对所有被混淆的路径分支条件进行有效的求解.路径约束关系的复杂度取决于多个因素:首先,是哈希函数的复杂度,从图 7 可见,引入 CRC32 所增加的约束关系是最少的,最多的是 SHA512,因此,混淆后约束关系的复杂度与选取的单向函数的复杂度成正比;其次,是被混淆路径分支条件的执行次数,在 specrand 的执行轨迹中,被混淆的路径分支条件执行了 40000 多次,而 bzip2 的混淆条件只执行了 257 次,从图 7 可见,specrand 中约束关系的数量远远高于 bzip2 的,因此,约束关系的复杂度与被混淆的路径分支条件的执行次数成正比;然后,是前缀集合的大小,前缀集合越大,路径选择过程的比较次数就会越多,因此路径约束关系也就越复杂.

实验表明,本文提出的路径分支混淆方法大幅提升了程序路径约束关系的复杂度,可以有效地增加攻击者逆向工程路径分支信息的难度.同时,该策略增加了程序的执行开销.

5.3　程序开销测试

路径分支混淆会增加程序的额外开销,包括时间开销和空间开销两部分.时间开销是指路径分支混淆后程序执行时间的增加量,以秒为单位.在不同的程序中,路径分支混淆算法引入的时间开销取决于 3 部分:算法中所使用的哈希函数算法,哈希算法越复杂,时间开销越大;被混淆的路径分支的前缀数量,每次执行混淆代码,输入值的哈希结果要与所有的前缀进行一一匹配;被混淆路径分支的执行次数,每次路径分支判断过程都会引入时间开销.如果被混淆分支在程序的多层循环结构的最内层,在程序执行过程中混淆代码可能会被不断地重复执行,那么混淆代码对程序整体的执行时间影响很大.如果被混淆分支在程序整个执行过程中仅执行 1 次或几次,那么混淆代码对程序的整体执行时间影响很小.空间开销是指混淆后程序体积的增加量,以字节为单位.

实验的测试环境是 IntelCore2 Q9400 CPU, 4GB RAM, Windows XP.首先,我们将在一个简单的日期检测程序上对路径分支混淆的开销进行测试,然后,在 SPECint-2006 测试集上验证路径分支混淆的时间和空间开销.如图 8 所示,简单的日期检测程序输入为月份和日期两个变量,代码中有两个路径分支条件,共有 3 条不同的执行路径,分别触发行为 behavior_ *a*()、behavior_ *b*()和 behavior_ *c*().

实验过程中,程序会随机地选择 10000 组月份和日期数据输入给测试样本,然后,利用操作系统提供的 GetTickCount()函数,计算程序执行过程所使用的时间,单位是毫秒.路径分支混淆过程分别选择了 5 种哈希函数:CRC32、MD5、SHA1、SHA256 和 SHA512.为了避免不必要的干扰,实验将重复 3 次,并计算时间开销的平均

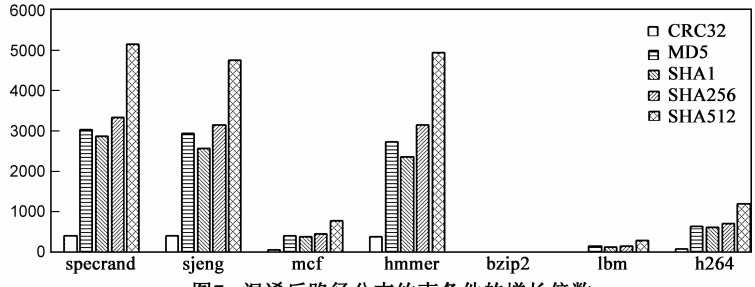

图7　混淆后路径分支约束条件的增长倍数

值,实验结果如表 3 所示.

```
void time_check(int month,
                int day){
s1:  if (month > 6
                && month < 10){
s2:    if (day > 15){
s3:      behavior_a();
       }
s4:    else{
s5:      behavior_a();
       }
     }
s6:  else{
s7:    behavior_c();
     }
   }
```

图 8 简单的日期检测程序

表 3 的第一行数据是路径分支混淆前样本的执行时间和文件大小,从表中数据可见,它的执行时间是最少的.第二行数据是采用了 CRC32 哈希函数混淆后样本的执行时间和文件大小,经过 3 次测试,平均执行时间增加了约 8.4s,是混淆前的 6.8 倍.因为二进制条件跳转指令的执行时间非常短,可以忽略不计,所以增加的 8.4s 是大约 2 万次输入与前缀集合进行匹配的时间开销,平均每次前缀匹配的时间不到 0.5ms.随着哈希函数复杂度的增加,程序的执行时间也在不断的增加,其中基于最复杂的 SHA512 哈希函数的路径分支混淆大约增加了 15.7 倍的程序执行时间,平均每次前缀匹配要花费 1ms 左右的时间.

表 3 路径分支混淆算法的路径分支混淆的开销

(时间单位:ms;长度单位:字节)

单向函数	时间 1	时间 2	时间 3	平均时间	文件长度
Original	2235	1078	1015	1442.7	180272
CRC32	10171	9797	9484	9817.3	237568
MD5	11328	10094	11250	10890.7	249908
SHA1	11562	10656	11718	11312.0	258103
SHA256	12234	10984	12687	11968.3	258102
SHA512	21907	24250	21875	22677.3	282678

如表 3 所示,路径分支混淆后程序体积的增加并没有像执行时间那样增加了 10 倍左右,体积仅仅增加了 30% 到 50%,而且体积并不会随着被混淆的路径分支数的增长而快速增长.因为路径分支混淆函数可以被重复调用,而且,定理 1 和 2 也证明了前缀的数量是有限的,所以,体积的增长也是有限的,不会像 Sharif 等人[14]提出的混淆方案那样无限地增长.去除引入单向函数所增加的程序体积,每混淆一个路径分支条件大约要增加 1000 个字节.

图 9 和图 10 是 SPECint-2006 测试程序在路径分支混淆后时间和空间开销的变化.图 9 的横坐标中 org 表示原始程序,其他的分别表示所引入的哈希函数,纵坐标是执行时间,单位是秒,不同的曲线对应着不同的测试程序.从图 9 可见,大部分程序混淆后的执行时间只增加了 1 到 2s.对于程序 specrand 和 hmmer,选择 SHA512 算法和 CRC32 算法的执行时间分别相差了 86.5s 和 25.7s,其原因是 specrand 和 hmmer 中被混淆的路径分支条件分别被执行了 48478 次和 5000 次.因此,当引入的混淆代码在执行轨迹中所占比例比较大时,选择不同复杂度的哈希函数对程序的时间开销影响很大.图 10 显示了测试程序混淆后体积增长的比例,横坐标是路径分支混淆所采用的哈希函数,纵坐标是体积增长的百分比.采用不同的哈希函数程序所引入的空间开销基本上是一条水平的直线,因此,对于空间开销的增长,采用不同的哈希函数差异不大.空间开销的增长比例与前缀集合的大小成正比,与程序混淆前的体积成反比.图 10 中体积增长比例最大的是 specrand,specrand 混淆前的长度是 7332 字节,混淆后引入了 31 个前缀,共增加 689 字节,体积增长了 9.40%.而 sjeng 的长度是 161851 字节,混淆后增加了 692 字节,体积只增长了 0.4%.虽然程序体积增长的比例不同,但是程序平均仅增加了不到 1000 个字节.

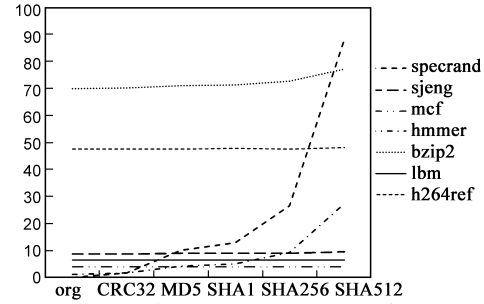

图9 路径分支混淆后SPECint-2006测试程序的时间开销

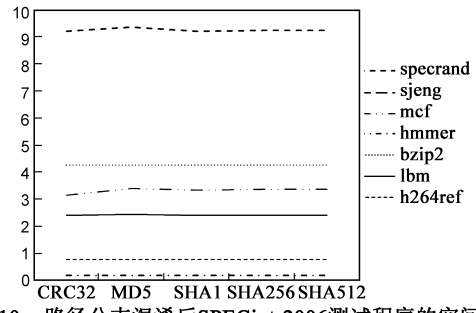

图10 路径分支混淆后SPECint-2006测试程序的空间开销

测试结果表明路径分支混淆算法的路径分支混淆

策略,使逆向工程无法求解混淆后的路径分析信息,混淆后程序增加的平均执行时间和文件长度分别只有 1 到 2s 和不到 1000 个字节,具有实用价值.

6 相关工作

程序路径分支混淆策略的思路有两种:(1)增加攻击者收集路径信息的难度;(2)增加攻击者分析路径信息的难度,如图 11 所示.

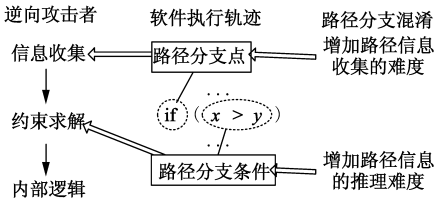

图11　路径分支混淆策略

增加攻击者收集路径信息的难度可以通过将程序的路径分支条件隐藏在非跳转指令的副作用中,例如,贾春福等人[17]通过隐式的 CPU 指令副作用替换了显式的二进制条件跳转指令.由于能够产生异常的 CPU 指令和执行环境非常多,攻击者要是穷举二进制代码所有可能的异常情况是不现实的,所以极大地增加了攻击者收集路径信息的难度.Bos H 教授的研究团队利用内存错误检测的难题,将分支条件隐藏于内存错误的触发条件中,增加分支信息的收集难度[18].王怀军等人[19]结合等价变形、控制流混淆、动态加解密技术,研发了二进制代码混淆保护原型系统 MEPE,实现对二进制代码的深入保护.何炎祥等人[20]利用程序流敏感分析方法选择比较重要的指令进行混淆,提出二步比较混淆模型,减少代码混淆造成的额外开销而又不影响代码混淆的质量.

Falcarin 等人[21]、Ceccato 等人[22]和王志等人[23]提出了基于代码移动性的二进制代码混淆策略.在不可信主机环境中的代码是不完整的,在程序执行过程中需要与远程的可信实体进行频繁的代码交互.可信实体是不受攻击者控制的,这样有效限制了攻击者对代码的可见度,增加了攻击者收集路径信息的难度.

增加攻击者分析路径信息的难度是指即使攻击者可以轻易地收集到软件的路径信息,但是要想通过这些泄露的路径信息还原程序逻辑关系是难以完成的.Sharif 等人[14]利用哈希函数对路径分支条件进行加密,该策略是本文方法的一种极限情况.Sharif 的保护对象是数据空间中的一个点,该方法可以看成其前缀位的长度正好等于哈希值的长度.本文的保护对象不仅仅是数据空间中的一个点,而且可以是空间中的一个区间,因此前缀位的长度与哈希值的长度可以不相等.王志等人[24]将未解数学难题引入到程序的路径信息中,使攻击者逆向求解路径分支信息的难度等价到求解未解数学猜想的难度.

7 结论

哈希函数具有很好的单向性,但是不具有保序性,只能用于保护等于关系的路径分支条件.保留前缀算法能够将一个数据区间转换成该区间的前缀集合,将区间的范围查询转换成前缀匹配操作.本文将保留前缀算法与单向哈希函数相结合,提出了路径分支混淆算法的路径分支混淆策略,将路径分支条件中的关系判断转换为哈希值的前缀匹配操作,扩展了 Sharif 提出的条件代码混淆策略,将 Sharif 的条件代码混淆从保护数据空间中的一个点,扩展到保护一个区间.路径分支混淆后,攻击者逆向分析程序路径分支条件的难度等同于根据输出结果逆向分析哈希函数的输入值的难度,可有效缓解程序路径分支信息泄露问题所造成的安全威胁.路径分支混淆是不改变程序的原始语义的,因此,其保护强度也取决于被保护的数据区间占整个数据空间的比例.实验结果表明该混淆技术能有效保护程序的路径分支信息,相对于 Sharif 的条件代码混淆,路径分支混淆更具有普遍性,其时间开销和空间开销有限,具有实用性.

参考文献

[1] Falcarin P, et al. Guest editors' introduction: software protection [J]. IEEE Software, 2011, 28(2): 24 – 27.

[2] The Compliance Gap: BSA Global Software Survey [EB/OL]. Washington, DC: BSA, June 2014 [2014-08-01]. http://globalstudy.bsa.org/2013/downloads/studies/2013GlobalSurvey_Study_en.pdf.

[3] Eighth Annual BSA and IDC Global Software Piracy Study [EB/OL]. Washington, DC: BSA, 2011 [2013-01-26]. http://portal.bsa.org/globalpiracy2010/downloads/study_pdf/2010_BSA_Piracy_Study – Standard.pdf.

[4] King J. Symbolic execution and program testing [J]. Communications of the ACM, 1976, 19(7): 385 – 394.

[5] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [A]. Proceedings of the Network and Distributed System Security Symposium [C]. Rosten, VA: Internet Society, 2005.

[6] Ganesh V, Dill D. A decision procedure for bit-vectors and arrays [A]. Proceedings of International Conference on Computer Aided Verification [C]. Berlin: Springer, 2007. 519 – 531.

[7] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation [A]. Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design

and Implementation[C]. New York: ACM, 2007. 89 – 100.
[8] Wang C, Davidson J, Hill J, et al. Protection of software-based survivability mechanisms[A]. Proceedings of the International Conference on Dependable Systems and Networks[C]. Piscataway, NJ: IEEE, 2001. 193 – 202.
[9] Linn C, Debray S. Obfuscation of executable code to improve resistance to static disassembly[A]. Proceedings of ACM Conference on Computer and Communication Security[C]. New York, NY: ACM, 2003. 290 – 299.
[10] Myles G, Collberg C. Software watermarking via opaque predicates: implementation, analysis, and attacks[J]. Electronic Commerce Research, 2006, 6(2): 155 – 171.
[11] Birrer B, Raines R, Baldwin R, et al. Programfragmentation as a metamorphic software protection[A]. Proceedings of the International Symposium on Information Assurance and Security[C]. Piscataway, NJ: IEEE, 2007. 369 – 374.
[12] Xu J, Fan J, Ammar M, et al. Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme[A]. Proceedings of the IEEE International Conference on Network Protocols[C]. Piscataway, NJ: IEEE, 2002. 280 – 289.
[13] Li J, Omiecinski E. Efficiency and security trade-off in supporting range queries on encrypted databases[A]. Proceedings of the Annual IFIP WG 11.3 Working Conference on Data and Applications Security[C]. Berlin: Springer, 2005. 69 – 83.
[14] Sharif M, Lanzi A, Giffin J, et al. Impeding malware analysis using conditional code obfuscation[A]. Proceedings of the Network and Distributed System Security Symposium[C]. Rosten, VA: Internet Society, 2008. 321 – 333.
[15] Amanatidis G, et al. Provably-secureschemes for basic query support in outsourced databases[A]. Proceedings of the Annual IFIP WG 11.3 Working Conference on Data and applications security[C]. Berlin: Springer-Verlag, 2007. 14 – 30.
[16] Xiao L, Yen, I. Security analysis and enhancement for prefix-preserving encryption schemes[R]. Richardson, Texas: Department of Computer Science, UT Dallas, 2012.
[17] 贾春福,王志,刘昕,等.路径模糊:一种有效抵抗符号执行的二进制混淆技术[J].计算机研究与发展,2011,48(11):2111 – 2119.
Jia Chunfu, Wang Zhi, Liu Xin, et al. Branch obfuscation: An efficient binary code obfuscation to impede symbolic execution[J]. Journal of Computer Research and Development, 2011, 48(11): 2111 – 2119. (in Chinese)
[18] Andriesse D and Bos H. Instruction-level steganography for covert trigger-based malware[A]. Proceedings of Conference on Detection of Intrusion and Malware & Vulnerability Assessment[C]. Berlin: Springer, 2014. 41 – 45.
[19] 王怀军,等.基于变形的二进制代码混淆技术研究[J].四川大学学报:工程科学版,2014,46(01):14 – 21.
Wang Huaijun, Fang Dingyi, Li Guanghui, et al. Research on deformation based binary code obfuscation technology[J]. Journal of Sichuan University (Engineering Science Edition), 2014, 46(01): 14 – 21. (in Chinese)
[20] 何炎祥,等.基于程序流敏感的自修改代码混淆方法[J].计算机工程与科学,2012,34(1):79 – 85.
HE Yan-xiang, CHEN Yong, WU Wei, et al. A program flow-sensitive self-modifying code obfuscation method[J]. Computer Engineering & Science, 2012, 34(1): 79 – 85. (in Chinese)
[21] Falcarin P, et al. Exploiting code mobility for dynamic binary obfuscation[A]. Proceedings of the World Congress on Internet Security[C]. Piscataway, NJ: IEEE, 2011. 114 – 120.
[22] Ceccato M, TonellaP. CodeBender: remote software protection using orthogonal replacement[J]. IEEE Software, 2011, 28(2): 28 – 34.
[23] Wang Zhi, Jia Chunfu, Liu Min, et al. Branch obfuscation using code mobility and signal[A]. Proceedings of IEEE Workshop on Security, Trust, and Privacy for Software Applications[C]. Piscataway, NJ: IEEE, 2012.
[24] Wang Zhi, Ming Jiang, Jia Chunfu, et al. Linear obfuscation to combat symbolic execution[A]. Proceedings of European Symposium on Research in Computer Security[C]. Berlin: Springer, 2011. 210 – 226.

作者简介

王 志 男,1981 年 8 月出生,山西长治人,现为南开大学计算机与控制工程学院讲师,主要研究方向为二进制代码混淆和恶意代码分析与防治.
E-mail: zwang@nankai.edu.cn

贾春福(通信作者) 男,1967 年 5 月出生,河北文安人,现为南开大学计算机与控制工程学院教授,博士生导师,主要研究方向为计算机网络与信息安全、可信计算、恶意代码分析.
E-mail: cfjia@nankai.edu.cn