ISSN 1000-1239/CN 11-1777/TP
48(11): 2111-2119, 2011

路径模糊:一种有效抵抗符号执行的二进制混淆技术

贾春福 王 志 刘 昕 刘昕海
(南开大学信息技术科学学院 天津 300071)
(cfjia@nankai.edu.cn)

Branch Obfuscation: An Efficient Binary Code Obfuscation to Impede Symbolic Execution

Jia Chunfu, Wang Zhi, Liu Xin, and Liu Xinhai
(*College of Information Technical Science, Nankai University, Tianjin* 300071)

Abstract Symbolic execution can collect branch conditions along a concrete execution trace of a program and build a logical formula describing the program execution path. Then the logical formula is used to reason about the dependence of control flow on user inputs, network inputs and other inputs from the execution environment, which can be used to effectively direct dynamic analysis to explore execution path space of the program. Symbolic execution has been widely used in vulnerability detection, code reuse, protocol analysis and so on. But it can be also used for malicious purposes, e.g., software cracking, software tampering and software piracy. The reverse engineering based on symbolic execution is a new threat to software protection. This paper proposes a novel binary code obfuscation scheme that obfuscates branch conditions to make it difficult for symbolic execution to collect branch conditions from the execution trace. It conceals branch information by substituting conditional jump instructions with conditional exception codes and uses exception handler to transfer control. It also introduces opaque predicates into the obfuscated program to impede statistical analysis. Furthermore, this paper provides insight into the potency, resilience and cost of the branch obfuscation. Experimental result shows that branch obfuscation is able to protect various branch conditions and reduces the leakage of branch information at run-time that impedes reverse engineering based on symbolic execution to analyze program's internal logic.

Key words software protection; code obfuscation; symbolic execution; exception handling; opaque predicate

摘 要 符号执行能够对软件的路径分支信息进行收集和形式化表示,然后通过路径可达性推理得到软件行为同用户输入、网络输入等外部执行环境间的依赖关系.这些依赖关系已被广泛地应用到漏洞发掘、代码复用、协议分析等领域.该逆向分析技术也可被黑客用于软件破解、篡改和盗版等,对软件知识产权的保护带来了新的威胁.提出了一种新的基于路径模糊的软件保护方法以抵抗基于符号执行的逆向分析:利用条件异常代码替换条件跳转指令来隐藏程序的路径分支信息,使用不透明谓词技术引入伪造的路径分支来弥补程序在统计属性上的差异,并对路径模糊技术的强度、弹性和开销进行了分析.实验结果表明路径模糊技术能保护各类路径分支条件,有效减少路径分支信息的泄露,抵抗基于符号执行的逆向分析.

关键词 软件保护;代码混淆;符号执行;异常处理;不透明谓词

中图法分类号 TP311

收稿日期:2010-09-01;**修回日期**:2010-12-30

基金项目:国家自然科学基金项目(60973141);天津市自然科学基金项目(09JCYBJ00300);高等学校博士学科点专项科研基金项目(20100031110030);计算机软件新技术国家重点实验室(南京大学)开放课题(KFKT2011B28)

保护软件的知识产权,避免软件被逆向分析而受到篡改、破解或盗版等威胁,已经成为软件安全领域备受关注的一个重要问题.软件产业界对此极为重视.

最近几年,基于符号执行[1-2]的二进制代码逆向分析研究进展迅速.该技术结合了传统的静态分析和动态分析的优点,能够对二进制代码进行全面而准确的逆向分析.具体方法是:首先,记录一条程序执行路径的详细信息,称为程序的一条执行轨迹;接着,使用符号执行对执行轨迹中的敏感路径信息进行收集和形式化表示;然后,进行路径可达性推理[3-5],得到软件的行为同用户输入、网络输入、系统状态等外部环境间的依赖关系.这些依赖关系已经被广泛地应用到软件测试[6-9]、漏洞发掘[10-13]、恶意代码分析[14-16]、协议分析[17]和代码复用[18]等安全领域.

通过与污点分析、虚拟机以及云计算等技术的结合,这种新的二进制代码逆向分析技术的能力得到了大幅提升.2008 年,CMU 的 Brumley 等人[11]使用符号执行技术分析程序安装安全补丁前后路径分支条件的差异,进而推理出安全补丁所对应的 1-day 漏洞(1-day 漏洞是指安全补丁刚发布但用户还未及时升级的漏洞).2009 年,UC Berkeley 的 Molar 等人[12]利用符号执行在亚马逊的 EC2(elastic compute cloud)云计算平台上使用 864 个计算小时找到了 77 个 0-day 漏洞.2010 年,UC Santa Barbara 的 Felmetsger 等人[13]基于符号执行开发出了 Waler 系统,用于检测 Web 应用程序的未知逻辑漏洞.2010 年,UC Berkeley 的 Caballero 等人[18]利用符号执行分析出程序某功能的相关二进制代码及其接口格式,并将该段代码直接复用到其他软件的开发中.

基于符号执行的逆向分析技术仅仅是一种程序分析方法,黑客也可以利用该方法达到恶意目的,例如发现并利用 0-day 或者 1-day 漏洞,逆向分析程序的内部逻辑从而破解软件保护、窃取软件内部信息、进行软件盗版等.该逆向分析技术对软件的知识产权保护带来了新的威胁.

2007 年,Popov 等人[19]提出基于异常处理的二进制代码混淆技术.该技术使用异常代码替换程序中的非条件跳转指令以抵抗静态逆向分析.由于无条件跳转指令不产生路径分支,该技术不能用于保护程序的路径分支信息.因此,它不能抵抗基于符号执行的逆向分析.2008 年,Sharif 等人[20]提出了一种抵抗符号执行的方法,他们利用散列函数对程序的分支条件进行加密,例如将分支条件 $X==c$ 变成 $Hash(X)==H_c$(其中 $H_c=Hash(c)$).由于散列函数的单向性,逆向分析不能从 c 的散列值 H_c 推导出满足分支条件的 X 的值.但是,散列函数不是线性函数,当 $x>y$ 时,$Hash(x)>Hash(y)$并不一定成立.因此,这种方法无法保护大于和小于等比较关系的分支条件.

本文提出了一种路径模糊技术,通过条件异常代码替换条件跳转指令来隐藏程序中的路径分支信息,并使用不透明谓词技术插入伪造的路径分支使混淆后的程序和原程序具有相似的指令分布规律.实验表明这种技术能有效减少软件内部路径分支信息的泄漏,阻止符号执行对路径分支信息的收集和形式化,增加逆向分析的难度.

1 路径模糊

二进制代码中的路径分支是通过条件跳转指令实现的,每个条件跳转指令会使程序产生 2 条分支路径.图 1(a)给出了一个简单的路径分支示例,其中有 2 个条件跳转指令:je 和 ja.当 $x>y$ 时,路径 1 被执行,触发行为 A 和 C.路径 1 的约束条件为 $(x!=y)\wedge(x>y)$.当 $x<y$ 时,路径 2 被执行,触发行为 A 和 D.当 $x==y$ 时,路径 3 被执行,只触发行为 B.在程序的执行轨迹中路径分支信息是易于发现的.符号执行可以有效地收集执行轨迹中泄漏的路径分支信息,并利用定理证明工具[3-5]对路径约束求解,推导出软件输入和控制流间的约束关系,进而以此关系对软件的路径空间进行高效地遍历.

本文提出的路径模糊技术可以减少软件路径分支信息的泄漏,增加利用路径分支信息进行逆向分析的难度.它通过系统的异常处理机制,使用条件异常代码替换条件跳转指令.路径模糊技术包括 3 个部分:条件异常、不透明谓词和异常处理.图 1(b)是路径模糊技术的一个简单例子.图 1(a)中的条件跳转指令 je 被替换成了 xor x,y;div x;.如果 $x==y$,x 与 y 异或的结果是 0,于是 div x 会产生除零异常.这时异常处理函数会截获该异常信息并将程序的控制权转移到 je 的跳转地址,行为 B 被触发.如果 x 不等于 y,xor x,y;div x;就不会产生除零异常,程序顺序执行,行为 A 被触发.我们可以看到,可执行代码的条件跳转指令被非跳转指令替换,

表面上只存在一条执行路径,没有分支.但实际上,这段代码的语义和图 1(a)的代码完全一致.

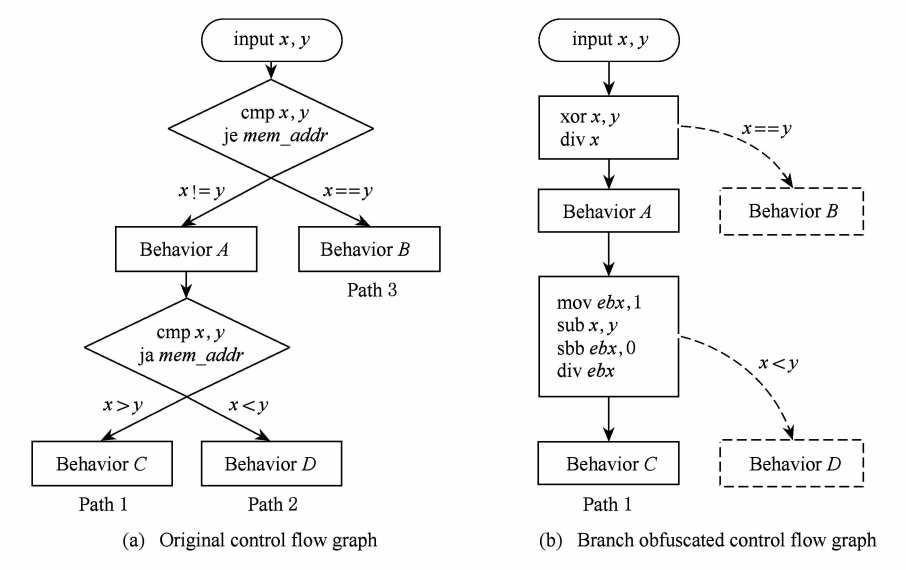


Fig. 1 Example of branch obfuscation.

图 1 路径模糊技术的简单例子

1.1 条件异常

为了抵抗符号执行,需要减少软件在执行过程中的路径分支信息泄露.符号执行是通过分析条件跳转指令来收集路径分支信息的.除了条件跳转指令,程序中能实现控制权转移的方法还有系统异常.Popov 等人[19]指出,很多指令和系统操作都可以被用来产生异常,例如非法指令、整数运算、浮点运算指令和内存访问操作等.本文在 Popov 等人[19]的研究基础上提出了条件异常技术.条件异常只在设定的条件满足时才会触发异常,可以用来隐藏路径的分支信息.

为了保持程序的语义,条件异常的触发条件必须与被替换的条件跳转指令的跳转条件一致.条件跳转指令以 EFLAGS 标志寄存器的标志位或者 *ecx*,*cx* 寄存器的值作为跳转的判断条件.EFLAGS 寄存器中用于条件跳转的标志位如表 1 所示:

Table 1 The EFLAGS Status Flags Used in Conditional Control Transfer

表 1 EFLAGS 中用于条件跳转的标志位

Flag	Position	Status Flags
CF	0	Carry Flag
PF	2	Parity Flag
ZF	6	Zero Flag
SF	7	Sign Flag
OF	11	Overflow Flag

条件异常代码可以直接访问 ecx,cx 寄存器,无法直接访问 EFLAGS 寄存器.但可以通过间接的方法获得 EFLAGS 标志位的状态.图 2 给出了 3 种间接获得图 2(a)中的跳转条件的例子:一种方法是用标志传送指令 lahf 读取 EFLAGS 中的标志位到 AH 寄存器,如图 2(b)所示;另外一种方法是使用条件传送指令,例如 cmove,cmova 等,如图 2(c)所示;除了以上 2 种方法外,还可以通过算术运算指令来判断跳转条件是否满足,如图 2(d)所示.因此,对于同一个跳转条件,可以通过多种间接方式进行判断.我们可以从中随机地选择一种判断方法来构造条件异常.条件异常的多样性增加了路径模糊技术的强度和弹性.

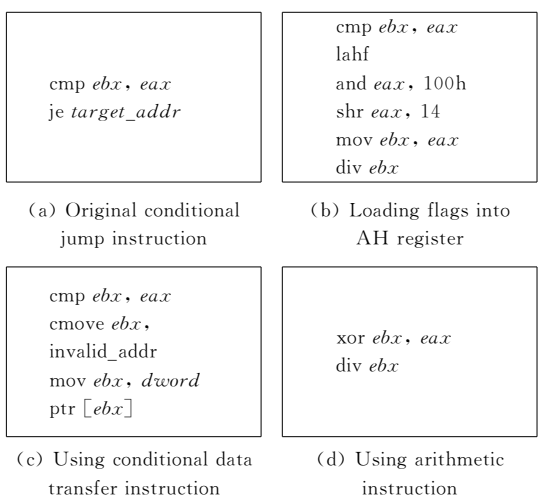
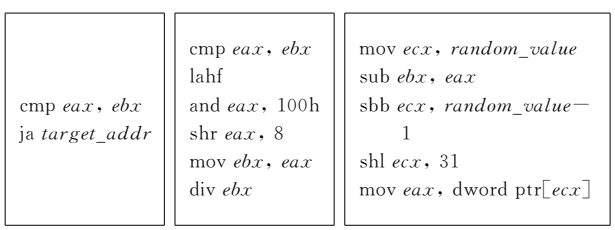


Fig. 2 Methods of determining branch condition.

图 2 几种判断分支条件的方法

判断路径的分支条件后,我们需要构建条件异常代码——在满足分支条件时产生异常.软件中的常用指令可以用来产生条件异常.如图 3(b)所示,div 指令可以用来构造条件异常:当操作数为 1 时,程序正常执行;当操作数为 0 时,产生除零异常.如图 3(c)所示,mov 指令也可以用来产生条件异常,其中 random_value 是一个随机的常数.当跳转条件满足时,即 *eax* 大于 *ebx*,*ebx* 减去 *eax* 将产生借位,EFLAGS 寄存器中的 CF 位被置位.下一条 sbb 指令是带借位的减法指令,*ecx* 不仅要减去 *random_value*−1,而且要减去 CF 位的值,*ecx* 的值变成 0.*ecx* 左移 31 位后,其值依然为 0.再下一条 mov 指令将读取 *ecx* 指向的内存数据到 *eax*,由于内存地址 0x00000000 是不可读的,因此会产生内存访问异常.如果跳转条件不满足,mov 指令读取的内存地址将会是 0x80000000,这个地址是可读的,不会产生内存访问异常.通常,程序中内存访问操作的频率远高于除法运算,因此多数情况下我们选择内存访问操作来构造条件异常.



(a) Original conditional jump (b) "div" conditional exception (c) "mov" conditional exception

Fig. 3 Examples of conditional exception.

图 3 用等价的条件异常替换条件跳转指令

1.2 不透明谓词

程序经过条件异常处理后,很多条件跳转指令被替换,这将导致程序的指令分布规律与普通程序的指令分布规律存在明显差异,从而可能被基于统计的异常检测技术[21-23]误报为恶意代码.这里,我们通过引入不透明谓词来消除指令分布上的差异.首先,计算指令分布在程序修改前后的差异.然后,插入不透明谓词,增加程序中条件跳转指令的数量,减少指令分布上的差异.

不透明谓词(opaque predicate)是一种控制流混淆技术.Collberg 等人[24-25]对不透明谓词技术在代码混淆中的应用做了比较完整的描述.谓词 P 在程序中的某一点 p,如果在混淆之后对于混淆者是可知的(基于先验知识)而对于其他人是难以获知的,则称该谓词为不透明谓词.不透明谓词技术可以被用来向顺序执行的代码中插入条件恒为真或者恒为假的路径分支,这些路径分支不影响代码的实际执行顺序,只是使代码的控制流变得复杂且难以分析.这种方法增加了条件跳转指令的数量,可用来弥补由条件异常替换引起的指令分布异常,从而提高了混淆代码的隐蔽性.

1.3 异常处理

在可执行代码中,条件跳转指令根据跳转条件是否满足将程序控制权转移给不同的分支路径.类似地,路径模糊技术用条件异常代码替换原程序的条件跳转指令,如果原跳转条件满足就产生异常,然后通过异常处理机制实现程序控制权的转移.路径模糊技术并不需要修改操作系统的内核也不依赖于特殊的驱动程序,它在异常处理机制中添加了一个新的异常处理函数.这个函数能根据异常产生的内存地址判断该异常是否由路径模糊代码产生.如果是,就把控制权转移到对应的分支路径并恢复程序的执行;如果不是,则调用系统中原有的异常处理函数进行处理.本节首先介绍操作系统的异常处理机制,然后描述路径模糊技术对异常处理机制的扩展.

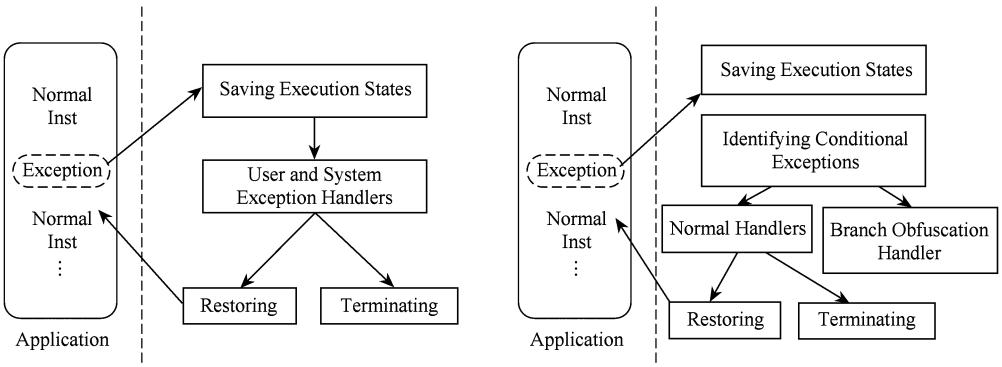
1) 异常处理机制

异常处理机制是操作系统的重要组成部分,是操作系统可靠性的重要保证.当发生硬件异常或软件异常时,CPU 将中止程序的执行,保存异常信息和当前程序状态并启动异常处理机制.异常处理机制尝试寻找能够处理此异常的异常处理函数.如果没有找到合适的异常处理函数,操作系统将终止程序的执行;如果找到相应的异常处理函数并且消除了异常,则操作系统将控制权转移到相应的返回地址,恢复程序的执行,如图 4(a)所示.

2) 路径模糊技术的异常处理

路径模糊技术用条件异常代码替换程序中的条件跳转指令,当原跳转条件满足时产生异常,并由异常处理函数实现跳转,如图 4(b)所示.异常处理函数在程序运行时自动地注册到系统异常处理机制中并且获得最高的处理优先级.如图 5 所示,异常处理函数的构造过程如下:首先,遍历可执行代码,找出被条件异常代码替换的路径分支指令;其次,提取路径分支指令的地址以及跳转的目的地址,构造一个映射表描述路径分支点和分支入口点的对应关系;再次,构造异常处理函数,根据映射表来确定异常的类型并采取相应的处理.如果异常产生的地址与映射表中的某个分支点相同,则表明该异常是路径

模糊代码产生的,将控制权转移到相应的分支入口点并恢复程序的执行;否则,如果不是路径模糊代码产生的异常,将该异常交给系统的异常处理函数进行处理.



(a) Normal exception handling (b) Branch obfuscation exception handling

Fig. 4 Exception handling: normal and branch obfuscation.

图 4 正常程序和路径模糊技术的异常处理

为了隐藏异常地址和分支路径的关系,我们利用散列函数对映射表中的路径分支点进行加密,并随机地向映射表中加入干扰项,如图 5 所示. 干扰项没有实际对应的路径分支点,而且分支路径的入口地址也是伪造的,在实际运行中是不可达的. 如果强制执行伪造的分支路径,将使程序崩溃. 由于散列函数的单向性,通过散列值反推路径分支点的地址是不可行的,再加上伪造路径分支的干扰,提高了路径模糊的强度和弹性,但是计算散列值的过程增加了程序的运行开销.

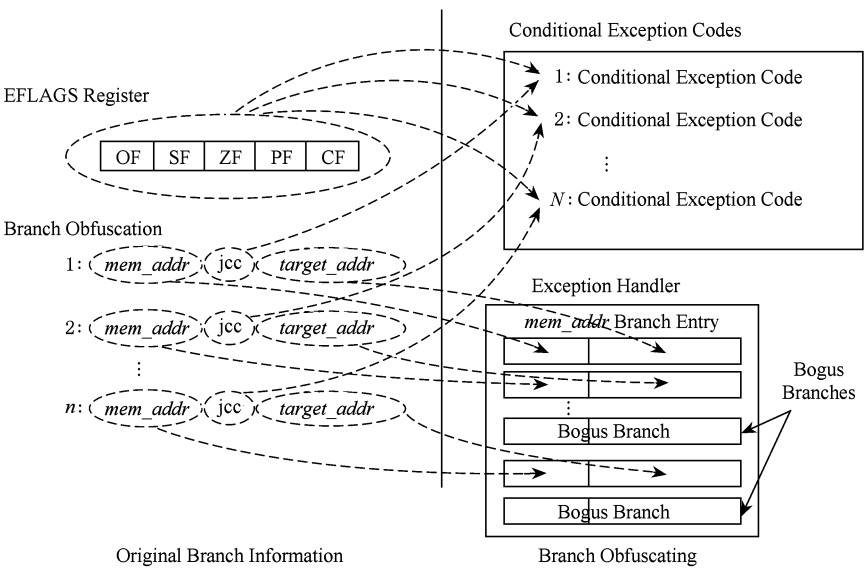


Fig. 5 Extracting original branch information and building branch obfuscation.

图 5 提取原程序中的路径分支信息用于路径模糊处理

2 技术评价

2.1 评价标准

Collberg 等人[24]提出了 3 个评价混淆算法的指标:强度(potence)、弹性(resilience)和开销(cost). 它们分别表示混淆算法为程序增加的复杂度,抵抗攻击的能力和运行时的额外开销. 本文在 Collberg 的评价指标基础上,结合符号执行的特点,对路径模糊技术进行了定性的评价.

强度是指路径模糊技术增加符号执行收集路径分支信息的复杂度. P 和 P' 分别表示原程序和路径模糊后的程序. 该复杂度由 3 部分组成:定位路径分支点的复杂度 $E_l(P')$、提取分支条件的复杂度 $E_c(P')$

以及确定程序路径的复杂度 $E_p(P')$.路径模糊 T 的强度为

$$T_{po}=\frac{E_l(P')+E_c(P')+E_p(P')}{E(P)}-1.$$

弹性是指路径模糊技术抵抗攻击的能力.它包括开发反路径模糊技术的难度 C_{dev} 和反路径模糊技术运行时的开销 C_{exe} 两部分.路径模糊技术的弹性为 $T_{re}(P)=Resilience(C_{dev},C_{exe})$.

开销是路径模糊技术给程序运行带来的额外开销.它包括 2 部分:时间开销 C_{tim} 和空间开销 C_{siz}.路径模糊技术的开销 $T_{co}(P)=Cost(C_{tim},C_{siz})$.

2.2 路径模糊的强度和弹性

构成条件异常代码的指令是程序中的常用指令,例如内存访问指令、算术运算指令等,如图 3 所示.符号执行难以从大量的常用指令中准确识别出条件异常代码,增加了符号执行定位路径分支点的复杂度.

此外,在异常处理函数的映射表中保存的是条件异常地址的散列值.由于散列函数的单向性,通过映射表中的散列值反推路径分支点的位置是困难的.而且,映射表中加入了伪造的对应关系,从映射表本身难以识别路径分支信息的真假.因此,符号执行难以从异常处理函数中准确判定路径分支点的位置.

另外,不透明谓词技术引入了许多干扰分支,平衡程序在混淆前后的指令分布,进一步增加了判定路径分支点的难度.

在路径模糊技术中,路径分支条件是通过间接读取 EFLAGS 标志位或者直接使用算术指令进行判断的.在程序中大部分指令都会影响 EFLAGS 标志位的状态,而且在程序中存在着大量的算术指令.在没有定位路径分支点的前提下,从大量的 EFLAGS 标志位的状态和算术指令中提取路径分支条件是不可行的.因此,路径模糊技术增加了符号执行提取路径分支条件的复杂度.

每一个确定的程序路径都对应着一组顺序执行的分支路径.虽然符号执行可以在异常处理函数的映射表中找到各个分支路径入口点,但是,无法确定分支路径的真伪,也得不到分支路径的执行顺序.因此,符号执行难以从映射表中获得程序路径的详细信息.

路径模糊技术的弹性取决于强度相关的 3 种复杂性.开发反路径模糊技术需要解决定位分支点、提取分支条件和确定程序路径 3 个问题,当前的符号执行技术无法通过逻辑推理解决这 3 个问题.Fuzzing[26-27] 技术是一种黑盒测试技术,可以绕过这 3 个问题进行软件分析.但是 Fuzzing 分析过程效率很低[6,28].Godefroid 等人[6] 指出 Fuzzing 技术测试一个 32 位的整数输入需要 2^{32} 次,存在着大量的冗余.Offutt 等人[28] 指出 Fuzzing 技术的代码覆盖率不高,难以对程序路径空间进行遍历.因此,路径模糊技术具有较强的弹性.

2.3 路径模糊的开销

路径模糊技术引入了条件异常代码、不透明谓词和异常处理函数,增加了原程序的运行开销和体积.但是,条件异常代码和不透明谓词一般只由很少的汇编指令构成(如图 2 和图 3 所示),而且路径模糊技术引入的异常处理函数功能简单,不会过多增加程序的运行开销,对程序体积的影响很小.因此,路径模糊是一种轻量级的二进制代码混淆技术.

3 实验结果与分析

我们使用了两段程序用于路径模糊技术的测试,它们分别以日期和 IP 地址作为路径分支条件.如图 6(a)所示,*time_check*()有 2 个 if 语句.*time_check*()的输入为月份(*month*)和日期(*day*),不同的输入会触发 3 种不同的行为:*behavior_a*,*behavior_b*,*behavior_c*.如图 6(b)所示,*ip_filter*()有 3 个 if 分支语句.*ip_filter*()的输入为 ip 地址,不同的 ip 地址,会触发 2 种不同的行为:*valid_ip*()和 *invalid_ip*().

```
void time_check(int month
                int day)
① if (month > 6){
②   if (day > 15){
③     behavior_a();
    }
④   else{
⑤     behavior_b();
    }
  }
⑥ else{
⑦   behavior_c().
  }
}
```

(a) Branch conditions based on date

```
void ip_filter(int i1, int i2, int
               i3, int i4){
①  if ((i1== 192){
②    if (i2== 168){
③      if (i3< 5){
④        valid_ip();
⑤        return.
       }
     }
   }
⑥: invalid_ip();
⑦: return.
}
```

(b) Branch conditions based on IP

Fig. 6 Branch conditions based on date and IP.

图 6 基于日期和基于 IP 地址的路径分支

在实验中,我们使用 Crest[29] 和 BitBlaze[30] 对测试样本进行符号执行分析,它们分别使用 Yices[5] 和 STP[3] 作为路径约束求解工具.

Crest 支持 4 种启发式的路径搜索策略:深度优先(DFS)、基于控制流图的(CFG)、一致随机(uniform random)和随机分支(random branch).

首先,用 Crest 的 4 种搜索策略对样本的源代码进行符号执行分析,结果如表 2 所示,除了随机搜索策略在 300 s 内没有遍历所有路径以外,其他的 3 种搜索策略都在 1 s 以内遍历了所有路径.

Table 2 Testing Result of Crest on Original Code

表 2 原代码的 Crest 分析结果

Strategies	date_check	ip_filter
DFS	4	6
CFG	4	6
uniform_random	4	6
random_branch	1	1

然后,对源代码进行路径模糊处理,过程为:首先,用 gcc 编译出可执行文件;其次,定位可执行文件中与输入相关的路径分支点并提取其路径分支条件;再次,构造条件异常代码替换条件跳转指令,并插入一段不透明谓词代码;最后,根据异常代码的位置以及分支路径入口地址生成映射表,并构造异常处理函数.

由于 Crest 是基于源代码的符号执行分析工具,它需要在源代码中插入分析代码.我们使用基于 IDA 的 Hey-Rays 反编译工具把路径模糊技术处理过的二进制代码反编译成 C 语言代码用于 Crest 分析.分析结果如表 3 所示:

Table 3 Testing Result of Crest on Branch Obfuscated Code

表 3 路径模糊后的可执行代码的 Crest 分析结果

Strategies	date_check	ip_filter
DFS	2	3
CFG	2	3
uniform_random	2	3
random_branch	4	2

Crest 分别在 2 段代码中找到了 2 和 3 个路径分支,并非原来的 4 和 6.对这个结果进行简要的分析:代码在经过条件异常处理之后,由于条件跳转指令被非条件跳转代码替代,所以不存在基于条件跳转指令的路径分支,因此 Crest 分析得到的路径分支数为 0;为了平衡指令分布上的差异,我们往代码中加入了与原代码条件跳转指令数量相同的不透明谓词跳转指令,这些跳转指令是恒为真或恒为假的,有一半的路径分支是不可达的,所以使得路径分支数变为 2 和 3.date_check 代码基于随机分支搜索策略的测试结果为 4,是因为在测试过程中 Crest 出现了整数溢出,使不可达路径分支被触发,导致程序崩溃.

接着我们使用 BitBlaze 平台对样本的二进制代码进行了分析.首先,BitBlaze 平台的 TEMU 通过插装 QEMU 虚拟机,追踪样本在虚拟机中执行的每条指令并将其记录在文件中.接着,BitBlaze 平台的 VINE 从记录下来的指令执行过程中提取出与输入相关的分支条件.然后,使用定理证明工具 STP 推理是否存在其他输入能够触发未被执行的分支路径.

我们记录了样本在路径模糊前后不同行为所对应的路径信息,如表 4 和表 5 所示.表 4 和表 5 的 Original 和 Obfuscated 行分别记录了样本在路径模糊前后分别执行的指令数,Added Instructions 行是模糊后增加的指令数,第 4 行是执行过程中所触发的异常条件跳转数.在路径模糊处理后的 obf_time_check 程序中,触发行为 *behavior_a*,*behavior_b*,*behavior_c* 的路径分别比模糊之前多执行了 291,158,18 条指令.在 *behavior_a* 的路径中触发了 2 次条件异常,在 *behavior_b* 的路径中触发了 1 次条件异常,而在 *behavior_c* 的路径中没有触发条件异常.由此可见,每次条件异常处理过程会使程序多执行 100 条左右的指令,只增加了程序很少的运行开销.表 5 的数据也表明条件异常处理过程只增加了少量的程序运行开销.

Table 4 time_check Path Information of Original and Obfuscated

表 4 time_check 路径模糊前后不同行为对应的路径信息

Behavior	*behavior_a*	*behavior_b*	*behavior_c*
Original	9 426	9 425	9 422
Obfuscated	9 717	9 583	9 440
Added Instructions	291	158	18
Conditional Exceptions	2	1	0

Table 5 ip_filter Path Information of Original and Obfuscated

表 5 ip_filter 路径模糊前后不同行为对应的路径信息

Behavior	*valid*()	*invalid*()
Original	11 631	11 838
Obfuseated	12 063	12 137
Added Instructions	432	299
Conditional Exceptions	3	2

路径模糊之前,BitBlaze 可以根据条件跳转指令的信息提取出路径分支条件并推理出触发不同路径分支所需的输入,进而遍历样本的路径空间;路径模糊之后,由于与输入相关的条件跳转指令被条件异常代码替换,BitBlaze 不能从指令流中识别条件异常代码并提取路径分支条件,因此无法有效地遍历路径空间.

上述实验结果表明本文提出的路径模糊技术可以有效地减少路径分支信息泄露,抵抗基于符号执行的逆向分析,而且没有过多增加程序的运行开销.

4 结 论

当前的软件保护技术没有重视对路径分支信息的保护,攻击者可以轻易地通过符号执行从软件的执行轨迹中收集到泄露的路径分支信息,并推理出软件的内部逻辑.针对路径分支信息泄露的问题,本文提出了路径模糊技术.路径模糊是一种轻量级的二进制代码的混淆技术,它使用条件异常来隐藏路径分支信息,然后通过不透明谓词技术引入伪造的路径分支来弥补程序在统计属性上的差异.同时,还对路径模糊技术的强度、弹性和开销进行了定性的评价.实验证明路径模糊技术能够有效保护软件的路径分支信息,阻止基于符号执行的逆向分析.

我们在下一步工作中,将从路径分支条件的混淆入手,增加路径约束求解的难度,以更好地保护软件知识产权,抵抗逆向分析攻击.

参 考 文 献

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

[2] Zhang Jian. Sharp static analysis of programs [J]. Chinese Journal of Computers, 2008, 31(9): 1549-1553 (in Chinese) (张健. 精确的程序静态分析[J]. 计算机学报, 2008, 31(9): 1549-1553)

[3] Ganesh V, Dill D. STP: A decision procedure for bitvectors and arrays [OL]. [2010-06-10]. http://verify. stanford. edu/PAPERS/STP-ganesh-07. pdf

[4] Cadar C, Ganesh V, Pawlowski P, et al. EXE: Automatically generating inputs of death [C]//Proc of the 13th ACM CCS. New York: ACM, 2006: 322-335

[5] Dutertre B, Moura L M. A fast linear-arithmetic solver for DPLL(T) [C] //Proc of CAV'06. Berlin: Springer, 2006: 81-94

[6] Godefroid P, Levin M Y, Molnar D. Automated whitebox fuzz testing [C] //Proc of Network and Distributed System Security Symp. Reston, VA: ISOC, 2008: 1-11

[7] Cadar C, Engler D. Execution generated test cases: How to make systems code crash itself [C] //Proc of Int SPIN Workshop. Berlin: Springer, 2005: 2-23

[8] Cadar C, Engler D, Engler D. Klee: Unassisted and automatic generation of highcoverage tests for complex systems programs [C] //Proc of USENIX OSDI'08. Berkeley, CA: USENIX, 2008: 209-224

[9] Lee G, Morris J, et al. Using symbolic execution to guide test generation [J]. Software Testing, Verification & Reliability, 2005, 15(1): 41-61

[10] Brumley D, Newsome J, et al. Towards automatic generation of vulnerability-based signatures [C] //Proc of IEEE Symp on S&P. Piscataway, NJ: IEEE, 2006: 2-16

[11] Brumley D, Poosankam P, et al. Automatic patch-based exploit generation is possible: Techniques and Implications [C] //Proc of IEEE Symp on S&P. Piscataway, NJ: IEEE, 2008:143-157

[12] Molnar D, Li X C, Wagner D A. Dynamic test generation to find integer bugs in x86 binary linux programs [C] //Proc of USENIX Security Symp. Berkeley, CA: USENIX, 2009: 67-82

[13] Felmetsger V, Cavedon L, et al. Toward automated detection of logic vulnerabilities in Web applications [C] //Proc of the USENIX Security Symp. Berkeley, CA: USENIX, 2010: 143-160

[14] Brumley D, Hartwig C, et al. Bitscope: Automatically dissecting malicious binaries, CMU-CS-07-133 [R]. Pittsburgh, PA: Carnegie Melleon University, 2007

[15] Brumley D, Hartwig C, et al. Automatically identifying trigger-based behavior in malware, CMU-CS-07-105 [R]. Pittsburgh, PA: Carnegie Melleon University, 2007

[16] Moser A, Kruegel C, Kirda E. Exploring multiple execution paths for malware analysis [C] //Proc of the IEEE S&P. Piscataway, NJ: IEEE, 2007: 231-245

[17] Newsome J, Brumley D, et al. Replayer: Automatic protocol replay by binary analysis [C] //Proc of the 13th ACM CCS. New York: ACM, 2006: 311-321

[18] Caballero J, Johnson N, et al. Binary code extraction and interface identification for security applications [C] //Proc of the 17th Annual Network and Distributed System Security Symp. Reston, VA: ISOC, 2010: 275-290

[19] Popov I V, Debray S K, Andrews G R. Binary obfuscation using signals [C] //Proc of the USENIX Security Symp. Berkeley, CA: USENIX, 2007

[20] Sharif M, Lanzi A, et al. Impeding malware analysis using conditional code obfuscation [C] //Proc of Network and Distributed System Security Symp. Reston, VA: ISOC, 2008: 321-333

[21] Kolter J Z, Maloof M A. Learning to detect malicious executables in the wild [C] //Proc of the 10th ACM SIGKDD Int Conf on Knowledge discovery and data mining. New York: ACM, 2004: 470-478

[22] Masud M, Khan L, Thuraisingham B. A hybrid model to detect malicious executables [C] //Proc of IEEE ICC'07. Piscataway, NJ: IEEE, 2007: 1443-1448

[23] Moskovitch R, Feher C, Tzachar N, et al. Unknown malcode detection using OPCODE representation [C] //Proc of the 1st European Conf on Intelligence and Security Informatics. Berlin: Springer, 2008: 204-215

[24] Collberg C, Thomborson C, Low D. A taxonomy of obfuscation transformations, TR148 [R]. Auckland: The University of Auckland, 1997

[25] Collberg C, Thomborson C, Low D. Manufacturing cheap, resilient, and stealthy opaque constructs [C] //Proc of the 25th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 1998: 184-196

[26] Bird D, Munoz C. Automatic generation of random self-checking test cases [J]. IBM Systems Journal, 1983, 22(3): 229-245

[27] Forrester J E, Miller B P. An empirical study of the robustness of Windows NT applications using random testing [C] //Proc of the 4th USENIX Windows System Symp. Berkely, CA: USENIX, 2000: 59-68

[28] Offutt J, Hayes J. A semantic model of program faults [C] // Proc of ISSTA'96. New York: ACM, 1996: 195-200

[29] Burnim J, Sen K. Heuristics for scalable dynamic test generation [C] //Proc of ASE'08. Piscataway, NJ: IEEE, 2008: 443-446

[30] Song D, Brumley D, Yin H, et al. BitBlaze: A new approach to computer security via binary analysis [C] //Proc of ICISS'08. Berlin: Springer, 2008: 1-25

Jia Chunfu, born in 1967. PhD, professor and PhD supervisor. His main research interests include information security and trusted computing, malware analysis and prevention.

Wang Zhi, born in 1981. PhD candidate. His main research interests include code obfuscation, malware analysis and prevention.

Liu Xin, born in 1974. PhD candidate. Her main research interests include malware analysis and prevention.

Liu Xinhai, born in 1986. MSc candidate. His main research interests include code obfuscation.