基于软件多样化的拟态安全防御策略

张宇嘉[1] 庞建民[1] 张 铮[1] 邬江兴[2]
(数学工程与先进计算国家重点实验室 郑州 450001)[1]
(国家数字交换系统工程技术研究中心 郑州 450001)[2]

摘 要 随着逆向工程的不断发展,软件产业的利益在很长一段时间内受到了来自盗版产业和恶意行为的破坏。为了找到一种低成本的高效抗逆向方法,人们引入了原本为恶意软件用于隐藏自身恶意行为的混淆技术来提高逆向工程的门槛。但是,现存的大多数混淆方法都是语言相关或者依赖目标平台的,对逆向的作用往往极其有限。鉴于此,提出一种基于LLVM的新的编译时的混淆实现方法,并结合拟态防御思想提出一种新的软件防御策略,其能有效防御针对软件的多种恶意攻击。

关键词 混淆,软件多样化,拟态防御

中图法分类号 TP309 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.02.037

Mimic Security Defence Strategy Based on Software Diversity

ZHANG Yu-jia[1] PANG Jian-min[1] ZHANG Zheng[1] WU Jiang-xing[2]
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)[1]
(National Digital Switching System Engineering & Technological R&D Center, Zhengzhou 450001, China)[2]

Abstract With the development of reverse engineering, the software industry has suffered a great loss from the software piracy and malicious attack for a long time. Code obfuscation techniques which can hide specific function of a program from malicious analysis for malware is thus frequently employed to mitigate this risk. However, most of the existing obfuscation methods are language embedded and depend on the target architecture, this paper proposed a method of compile-time obfuscation, and further presented a prototype implementedation based on the LLVM compiler infrastructure. Furthermore, this paper implemented a mimic security defence system which is free from malicious attack with the software diversity method.

Keywords Obfuscation, Software diversity, Mimic defence

1 引言

随着逆向工程的兴起,软件保护技术近年来受到了极大的重视。IT安全产业单就软件安全这一领域的投入就不可估计,即便如此,现存的软件保护技术在应对软件篡改、恶意逆向和破解等方面仍极其薄弱。通常,针对知识产权的保护有法律和技术两种途径,法律上通过保护版权和专利来禁止专属软件的复制和恶意利用,而技术上的保护是指软件开发者通过一些技术手段使得软件难以复制或被恶意利用。近年来对恶意代码的研究比较热门,其通常被描述为怎样保护主机不被潜在的不可信外来代码破坏;然而与其对应的针对恶意主机问题的研究相对较少,恶意主机的问题可被归纳为怎样保护可信的程序免受潜在的恶意主机的攻击。由于在不可信的主机环境中,攻击者对软件拥有无限的控制权,可以借助一系列调试工具、反汇编工具和反编译工具对软件进行分析和破解。从现存的恶意代码的防御方法(如防火墙、安全网关和可信通信终端等)中可以明显看出,这些针对恶意代码的防御方法因只是保护数据免受外部攻击,而对恶意主机的内部攻击和软件破解无能为力。为了抵抗来自内部的针对软件的恶意行为,一种普遍使用的方法是利用软件的内生机制(即在编程时提前)对一系列恶意行为进行防御,其通过在程序中加入混淆代码和机制来增加攻击者的逆向难度。

近期,Schrittwieser等人[1]发表了关于现存的混淆技术是否能够与最新的代码分析技术抗衡的报告,报告指出混淆

到稿日期:2016-12-27 返修日期:2017-03-21 本文受国家重点基础研究发展计划(2016YFB0800104),上海市科学技术委员会科研计划(14DZ1105300),自然科学基金(61472447)资助。

张宇嘉(1992-),男,硕士生,助理工程师,主要研究方向为软件逆向、信息安全,E-mail:365663308@qq.com;**庞建民**(1964-),男,博士,教授,博士生导师,CCF高级会员,主要研究方向为逻辑与推理、信息安全,E-mail:jianmin_pang@126.com(通信作者);**张 铮**(1976-),男,博士,副教授,主要研究方向为操作系统;**邬江兴**(1953-),男,中国工程院院士,主要研究方向为通信系统、拟态安全。

技术和代码分析技术作为两种相互博弈的技术，混淆强度高低的评估很大程度上依赖于分析者的目标和可获得的资源。

混淆的目的是防止在自动化工具或者人力的帮助下获得程序中的重要信息，从而通过混淆技术将其隐藏。混淆技术最早起源于恶意软件[2]，其编写者通过多态引擎将恶意代码生成不同的变体从而躲避基于特征的检测程序，并在一定程度上降低了代码的可读性。此时期混淆技术的兴起，就标志着代码开发者和分析者之间博弈的开始。越来越多的复杂混淆技术被开发者用来隐藏自己行为的代码，与此同时，各种先进的分析工具和技术也应运而生。

文献[3]指出，多样化技术的发展在一定程度上增加了攻击者发现和利用软件漏洞的难度，内存重排[4-5]、随机化系统调用[6-7]以及指令随机化等一系列多样化技术都具有该功效。混淆技术，主要是通过编译时混淆和二进制重写来实现多样化。本文提出的混淆方法以编译时混淆为基础，通过修改编译器并在其上增加一个 Pass 来实现。将程序作为输入，经过修改后的编译器编译后，可以生成多个异构变体，因此该编译器为多样化编译器。

本文提出的基于拟态防御思想的攻击防御检测策略，不依赖于自身策略的保密性，通过混淆技术对待防护程序构造了一个含有多个异构变体的集合。图 1 展示了依赖于多样化编译器的功能等价多变体的生成过程。程序的所有输入将会被复制并分发给所有异构变体。通常，多变体的异构性特征与其采用的混淆技术相关，当攻击特征与变体异构性特征重合时，攻击产生的输出在不同变体上将会不同，通过比较变体的所有输出并对其进行表决，可以防御针对该异构特征的攻击。详细设计框架如图 2 所示，假设变体 A，B，C 在内存中拥有不同的地址空间，对于变体 A 有效的绝对地址无法有效地访问到变体 B 和 C，反之则亦。变体 A，B，C 由于具有相同的语义功能，因此对于正常的输入一定具有相同的输出。然而，当攻击者利用程序的绝对地址进行相关攻击时，该绝对地址在变体 B，C 中肯定无法同时有效，在防御策略的表决阶段可以通过对比发现攻击者针对非法内存地址空间的访问。

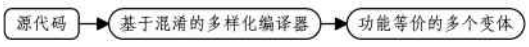

图 1 功能等价多变体的生成过程

Fig. 1 Generation process of equivalent variants

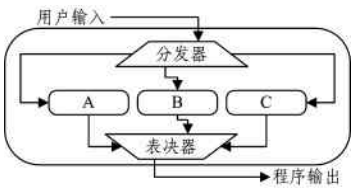

图 2 基于拟态防御的设计框架

Fig. 2 Framework based on mimic defence

从拟态防御的设计框架可以很容易看出，为了实现拟态防御机制带来的以上好处，原本只需要一个程序执行的工作在拟态中同时交给了 3 个变体，大大增加了其性能的损耗。然而，实际上在一些对安全比较敏感的行业（如政府和银行部门）中，这些性能损耗在一定程度上是可接受的。

2 相关研究

随着计算机的商业化，同构软件在经济上的可行性、行为上的一致性和分发逻辑上的简洁性等优点使其不断普及；与此同时，同构软件也容易被攻击者恶意拷贝并进行各种各样的分析。当攻击者通过分析发现同构软件中的漏洞和弱点并加以利用[8-9]时，其可以很轻松地在另一台装有该软件的机器上复现该过程。例如，当一些攻击者对一些商业软件进行破解并绕过其各种鉴权，使得用户不再愿意为这些软件买单时，无疑会给企业造成大量的经济损失。为了解决同构软件带来的这一系列问题，本文引入了混淆技术，其不仅增强了软件的抗逆向能力，而且带来了软件的多样化。其次，将多样化技术与拟态防御思想相结合，通过不同的多样化手段，提出了针对特定的攻击防御的检测策略。

混淆技术最早于 1998 年 Collberg[10-11] 在关于混淆技术不同种类分类的综述中正式被提出。该综述几乎涵盖了当时所有的常见混淆方法，同时包含一套衡量混淆方法有效性的指标，但因该指标过于主观而无法进行量化，针对混淆方法验有效性证的问题一直未能得到解决。Barak[12] 在之后的关于混淆技术的理论证明中推导出满足虚拟黑盒的混淆方法是不存在的（在虚拟黑盒中，混淆器被视作编译器，将输入的一段程序输出为具有相同功能的程序，但通过对比输入输出的程序很难找到其间的联系），除了一些在特定环境下针对某种特定目标进行的混淆。在 Barak 关于存在针对特定目标混淆的证明的基础上，Wee[13] 提出了对点函数的混淆可能性分析；另一方面，Bendersky[14] 指出相较于严格的图灵机混淆方式难于实现的现状，人工混淆的实现是有可能的，这也是“International obfuscated C code contest”自 1984 年举行至今的原因之一。

2.1 混淆的分类

根据混淆方法的不同，Collberg 将混淆技术分为布局混淆、控制流混淆和数据混淆；与该分类不同，本文以混淆的粒度为标准将其分为三大类。

1）细粒度（指令层）：混淆方法包括指令替换[15]、指令重排和花指令插入等。

2）中等粒度（基本块层）：该层主要通过基本块的变换来改变控制流，主要包括不透明谓词插入[11]、分支函数[16]、代码克隆[17]和控制流扁平化[18]等。

3）粗粒度（函数层）：将混淆作用于函数层，其基本变换包括随机化栈布局[19]、参数布局随机化和函数的拆分合并[10]。

根据以上的分类标准，本文提出的混淆方法属于指令层和基本块层，整合了指令替换、控制流扁平化、代码克隆、动态不透明谓词等。

2.2 混淆的周期

通常，混淆过程的生命周期是在编译时或者二进制阶段，

也就是常说的编译时混淆和二进制重写技术[20]。本文提出了一种基于 LLVM[21]编译器的新型编译时的混淆实现方法，其作为 LLVM 编译器的一个内部 Pass，在编译器的中间代码生成阶段发挥作用。

与二进制重写相比，选择编译时的混淆实现方式至少有两方面优点。

1）编译器针对多种硬件平台。如今的主流编译器（如 GCC 和 LLVM 等）主要由 3 个部分组成（前端、优化器和后端），这样的结构使得可以很容易地针对不同的平台设计相应的前端和后端；并且由于优化器部分通常采用编译器所特有的中间代码，其在优化阶段添加的特性也可以继承到所有平台。

2）编译时的混淆不用对目标程序进行反编译。以二进制重写方式来实现混淆，首先需要对目标程序进行逆向，而编译过程的损耗通常使得无法恢复原始的程序，这无疑提高了二进制重写混淆的难度。

混淆技术作为一种能够实现软件多样化的方式[22]，将其与拟态防御思想相结合，在保证拟态防御有效性的前提下能够降低拟态防御的实现难度。经典的拟态防御思想是指判断对多个软件执行体的相同输入是否有相同的输出，这里的多个软件执行体通常指分别由多个独立开发团队开发的具有相同功能但存在不同错误的软件。多版本的程序设计方法最早由 Joseph[23]提出，并应用于软件可靠性和容错系统领域；由于程序设计的独立性，其指出多版本的程序设计方法将会在某种程度上屏蔽软件中相同的设计错误或漏洞。但实验表明，相同的错误在不同的实现方式下仍然可能出现[24]。Bitansky[25]提出利用一种多线性等级编码的混淆方法对函数进行加密，其实际上是一种代数上特定的假设。之后，针对函数分支的信息泄漏，Wang[26]提出利用分支 hash 混淆来阻止逆向人员的符号执行。

2.3　拟态安全防御

文献[27]指出，拟态安全防御的基本思想是，在功能等价的条件下，以提供目标环境的动态性、非确定性、异构性和非持续性为目的，通过网络、平台、环境、软件、数据等结构的主动跳变或快速迁移来实现拟态环境，以防御者可控的方式进行动态变化，对攻击者则表现为难以观察和预测目标变化，从而大幅度增加包括未知的可利用的漏洞和后门在内的攻击难度和成本。

本文提出了一种新的基于编译器的混淆方法，并将其用于软件多样化。通过动态不透明谓词的引入和对控制流的混淆，其可以在原程序的基础上构造多个功能相同的异构变体，并将生成的多个异构多变体应用在拟态安全防御系统中，从而避免了多版本程序开发过程中的费用开销，同时也在一定程度上扩大了多样化变体库。

3　方法的实现

本文提出的新型编译时的混淆方法是在不透明谓词、控制流混淆和代码克隆的基础上实现的，虽然其继承于以上提到的 3 种混淆方法，但又存在区别之处，首先其通过将程序中的变量与构造的混淆谓词进行绑定，实现了不透明谓词的动态性。程序在每次执行中变量值的不确定性，造成相应的绑定谓词值也无法确定，从而使得控制流的选择对攻击者变得模糊。静态分析是攻击者在不执行程序的状态下通过反编译和调试工具对程序进行的分析，可以很容易暴露程序内在的一系列问题。通过静态分析，攻击者可以重建程序的控制流和处理逻辑，而动态不透明谓词的引入使得重建后的控制流杂乱且难以分析，从而干扰了对程序的静态分析。

另一方面，动态分析以准确性作为实现的基础，相较于以覆盖性为特点的静态分析，不透明谓词对动态分析的干扰无法实现。由动态调试工具的利用和之前针对恶意主机问题的分析可知，攻击者总是可以通过系统函数调用、指令执行和访问内存等方法来记录程序的实时状态，并通过不断分析迭代获得的数据，来得到较为精确的程序的控制流图。为了增大动态分析所获得的控制流的分析难度，本文通过整合控制流扁平化技术和代码克隆技术，来扩展程序控制流分支数量，使得攻击者在相同功能输入下每次获得的路径信息存在较大不同。

3.1　抗静态分析

不透明谓词通常指，谓词值在混淆阶段对混淆器是已知的，而对分析者不可知。然而，由于不透明谓词的值是恒定的，因此攻击者可以通过数次执行进行识别。为了弥补以上方法的缺陷，Palsberg[28]于 2000 年提出了动态不透明谓词的概念，并给出了一种实现方法，其通过使用一组在不同执行环境中值不同的谓词来实现动态变化。在此之后，Majumdar[29]描述了一种不稳定且不透明谓词的实现方法，即对于每一个谓词，其值是通过多个节点的协同计算获得的。以上两种方式虽然在一定程度上增大了不透明谓词被发现的难度，但同时也需要对程序源代码进行修改，且针对不同语言的修改方式也不同。本文提出的绑定程序变量的动态不透明谓词实现方法，在 LLVM 编译器工具链原型的基础上进行改造，通过生成一个 Pass，在中间代码生成阶段对程序进行混淆，从而不需要修改源代码。首先，该 Pass 随机选择程序中特定数据类型的变量并将其值赋给动态不透明谓词，此时，随着变量值的改变，动态不透明谓词的值也将改变，从而导致攻击者无法确定程序控制流的走向。

动态不透明谓词混淆的对象主要是函数中的部分基本块，下面对其混淆过程进行详细的描述。在一次混淆过程中，随机选择一个非末端基本块（记为基本块 A）并对其指令进行分析。根据分析结果判断基本块 A 的后继个数，以此进行后续操作。当基本块 A 只有一个后继时，构造基本块“obfBlock”并将其插入到 A，之后保存 A 的后继用于后续混淆（见 3.2 节）。基本块“obfBlock”用于动态不透明谓词的构造以及路径的选择。首先，对动态不透明谓词（记为 jmpVar）进行初始化；其次，从基本块 A 中随机选择一个变量与 jmpVar 绑

定;最后,对 jmpVar 进行一系列复杂运算(本文以简单的求余运算为例),并将获得的结果用于后续路径的选择。当基本块 A 含有两个后继时,混淆的唯一区别在于基本块"obfBlock"插入位置的选择,本文通过对比两个后继基本块的长度将基本块"obfBlock"插入到较长基本块之前。表 1 使用 LLVM 的中间代码展示了只有一个后继基本块时混淆前后的差别,其中动态不透明谓词%jmpVar 随机选择绑定基本块中的变量值,在程序的函数中,基本块之间的依赖关系非常明显,但动态不透明谓词的引入将会模糊这种关系。

表 1 动态不透明谓词的构造

Table 1 Construction of dynamic opaque predicate

原代码	混淆代码
%1=alloca i32	%1=alloca i32
%2=alloca i32	%2=alloca i32
%3=alloca i32	%3=alloca i32
store i32 %a,i32 * %1	store i32 %a,i32 * %1
store i32 %b,i32 * %2	store i32 %b,i32 * %2
store i32 %c,i32 * %3	store i32 %c,i32 * %3
br label %4	br label %obfBlock
;⟨label⟩:4	obfBlock:
…	%jmpVar=alloca i32
	;动态不透明谓词的声明
	store i32 %b,i32 * %jmpVar
	;随机选择变量值赋予谓词
	%5=urem i32 %jmpVar,2
	;为选择路径对谓词进行求余运算
	%6=icmp eq i32 %5,0
	br i1 %6,label %4,label
	%multiBlcok
	;⟨label⟩:4
	…
	multiBlock:
	…

注:表中展示了混淆前后的 LLVM 中间代码,与汇编代码类似,其中"br"与汇编中的"jmp"类似

3.2 抗动态分析

动态不透明谓词因其值的不确定性,能够有效地抵抗静态分析;然而,动态分析通过对程序状态和行为的监视,可以有效地获得运行时的控制流。由前面的恶意主机问题可知,攻击者可以通过动态调试工具得到程序在任意时刻的状态信息,通过多次的动态分析,攻击者可以通过掌握的规律发现不透明谓词的存在并将其移除[30]。

为了保护程序特别是敏感部位信息的安全,本文在动态不透明谓词的基础上又提出通过多样化程序控制流来模糊基本块之间的依赖关系。通过对函数中某些基本块的克隆体进行改造,建立了改造后的基本块与原基本块前驱(利用动态不透明谓词混淆后的基本块)的联系。其中,本文对克隆体的改造使用了两种混淆方法(基本块扁平化和等价指令替换)。对于控制流扁平化,首先打乱程序中基本块的顺序,然后将其放入 switch 语句中(又称调度器),并按照一定的规则构造不透明谓词,最后通过不透明谓词判断下一个跳转的目的基本块。本文使用的基本块扁平化区别于上述控制流扁平化,混淆的目标从基本块转换为了指令,前者模糊了指令间的逻辑关系,后者模糊了基本块间的逻辑关系。详细的转换流程如图 3 所示。混淆前后的指令见下划线部分。

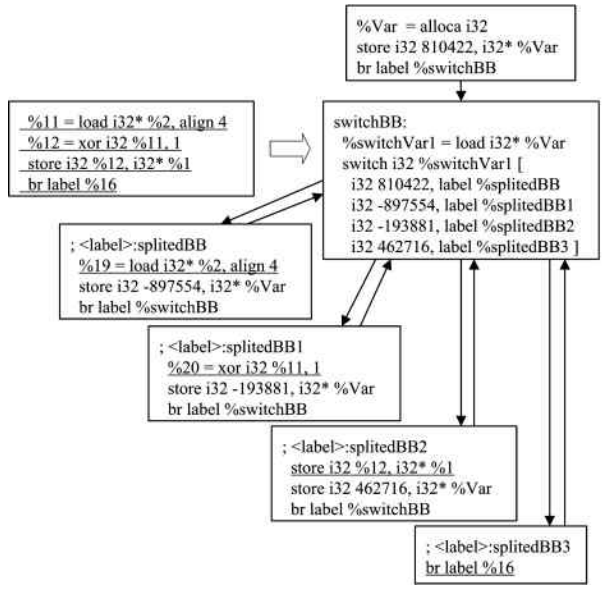

图 3 基本块的扁平化

Fig. 3 Flattening the basic-block

等价指令替换通常是指使用语义相同但句法不同的指令来替换原指令。本文对程序中出现次数较多的指令进行替换,如异或指令 XOR,表 2 详细描述了其实现过程。异或运算的结果实际上与无进位的加法相同,因此可以用加法运算替换程序中所有的异或运算。通过改变操作数的基数,使其在加法运算中不会产生进位,并在加法运算后将结果进行模数运算以恢复到原来的基数,这里的求余运算实际上与异或运算等价,且此时获得的目的操作数即为异或运算后的结果。以异或运算转换十进制加法运算为例,假设 $X=11B$,$Y=10B$,通过 X 异或 Y 可以得到 $Z=01B$。利用上述的替换方法,将 X 和 Y 转换为 $X'=11D$,$Y'=10D$,通过将 X' 与 Y' 相加可以得到 $Z'=X'+Y'=21D$。最后对 Z' 进行求余运算以将其转换回 Z。

表 2 等价指令替换

Table 2 Substitution of equivalent instruction

操作符	等价指令操作
$Y=D$ XOR X	$f:x=\sum x_i\cdot 2^i \mapsto x'=\sum x_i\cdot n^i$
	$h:x=\sum x_i\cdot n^i \mapsto x'=\sum(x_i \bmod 2)\cdot 2^i$
	$D'=f(D)$ and $X'=f(X)$
	$A=D'+X'=\sum(d_i+x_i)\cdot n^i$
	$A'=h(A)=h(D'+X')$
	$h(D'+X')=D$ XOR X

注:其中 D 和 X 是实数;函数 f 将以 2 为基数的变量 X 转换为 $n(n\geqslant 3)$ 为基数的对应值;$\sum x_i\cdot 2^i$ 是以 2 为基数的变量 X 的一种表示方式

4 方法评估

利用混淆技术保护软件在其自身的空间复杂度和时间复杂度的变化上总有一个折衷。为了评估本文提出的混淆方法,下面将通过以采用 C 语言实现的 OpenSSL 加密库为基准来对比混淆前后时间复杂度的变化。这些加密算法通常拥有复杂而庞大的控制流,能够对算法进行正确且高效的混淆同样是一次检验。表 3 列出了本文提出的混淆算法对 OpenSSL 加密库中 MD5,RC4,AES128-CBC,SHA256 和 DES-CBS 算

法的效率影响。由于混淆每次只对函数中随机的一个基本块进行多样化，以上加密算法的执行时间并没有过大增加。然而，增加每次混淆函数中基本块的个数将会大大提高攻击者的逆向难度，但同时也会增加执行时间。总之，选择一个合适的基本块混淆数量需要在安全和效率之间进行权衡。同时，为了检测本文混淆算法对程序空间复杂度的影响，引入编译器测试工具 Csmith，通过生产大量随机 C 程序计算混淆算法，增加了程序平均空间。通过 Csmith 产生的 1000 个 C 程序来对比其混淆前后空间大小的变化，结果空间约提升9.73%。同样地，该测试结果是对程序进行一次混淆所得出的。

表 3　混淆对不同加密算法性能的影响

Table 3　Impact of obfuscation on performance for various cryptographic algorithms

算法	未混淆	混淆后	比例/%
MD5	448247k	437840k	2.32
RC4	672466k	661727k	1.59
AES128-CBC	91072k	88672k	2.63
SHA256	178547k	175045k	1.96
DES-CBC	69281k	66669k	3.37

注：表中的数字代表了每秒能够加密 1024 字节数据块的个数

为了进一步展示本文的混淆算法对控制的改变，图 4 对比了针对函数 function()进行混淆前后的控制流。函数 function 的具体代码如下：

```
int function(int a, int b){
  int cmp = 4;
  if(a > cmp)
    return a^1;
  else
    return b^2;
}
```

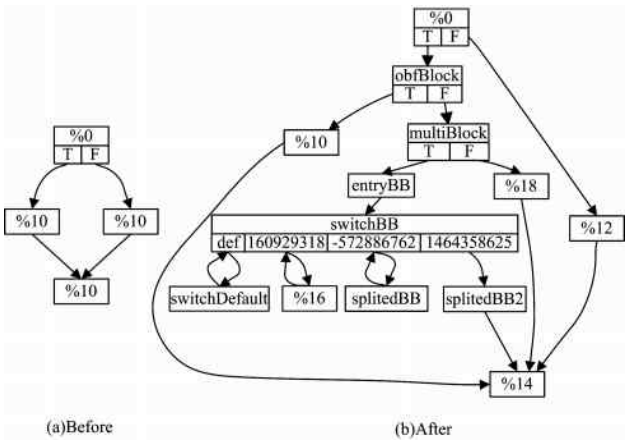

图 4　在函数 function()混淆前后控制流的变化

Fig. 4　Changing of control flow before and after function() obfuscation

5　结合拟态思想的防御策略

拟态防御技术通过执行多个程序实例并对比实例之间行为的差别，来阻止针对程序的恶意行为。这些实例又被称为变体。理想的变体通常指针对正常的输入具有相同的语义和行为；对于异常的输入或者恶意行为表现不同，但在实际实现中很难达到的实例。因此，变体的构造通常以最大化其异构性为目标，使其能够在受到攻击时拥有最大的输出差异性。变体间行为的不一致往往意味着程序的异常，通过这一特征可以有效地在攻击者对系统产生损害之前检测出程序的漏洞和后门。在拟态防御技术中，程序的输入被同时发送给多个变体，这种设计方式使得攻击者无法对每个变体构造特定的恶意输入以损害每个变体。可以看出，变体异构性能够使得针对一个变体的恶意输入在另一个变体上产生分歧，当变体异构性特征越多时，其能够抵抗的攻击种类也越多。

5.1　表决器的粒度

从根本上讲，拟态防御的核心是表决器机制，其监视着系统中所有变体的执行状态，并确保变体按照预定的规则运行。表决器又被称为表决器代理，通过对不同变体输出进行比较来保证没有任何程序实例被破坏。表决器的设计可以有不同的粒度，不同粗细的粒度虽然无法增加所检测到的攻击的种类，但其决定了攻击行为暴露的时间点。粗粒度的表决器模式针对变体程序的输出结果进行表决，阻止了通过利用变体缺陷进行信息窃取等需要进行通信的攻击方式。中粒度的表决器模式针对系统调用进行表决，借鉴了恶意软件分析中基于行为特征的分析方法，阻止了对系统本身的攻击，如文件破坏、隐蔽网络通信等。细粒度的表决器模式针对变体指令序列进行表决，通过对比同一时刻变体执行的两段指令和指令操作数是否语义等价，来阻止针对变体注入的恶意代码的执行。可以看出，表决器粒度的粗细直接影响了攻击能够到达的阶段。以攻击者利用远程代码注入进行系统文件窃取为例，其攻击大概可分为远程代码执行、读取文件信息、传输文件信息 3 个阶段。基于指令的细粒度表决器模式可以在攻击的远程代码执行阶段通过指令比较发现变体的异常。基于系统调用的中粒度表决器模式可以在攻击读取文件系统调用时发现变体异常。基于变体输出的粗粒度表决器模式可以在变体输出结果时通过对比发现异常。从上述分析可以看出，表决器模式的粒度越细，就能在越早的阶段发现攻击行为，阻止攻击带来的伤害，并对其进行处理；但是细粒度的表决模式引入的多核多线程调度、变体指令同步和随机数等问题，增加了实现的难度。

5.2　表决器的误报

从以上针对表决器的讨论中可以很容易发现其在表决过程中可能发生误报。对于粗粒度的基于变体输出的表决器，虽然不同变体接收到正常的输入会有相同的正常输出，但实际中的系统通常无法保证变体输出的同时性。例如，当某个变体在执行过程中受到第三方程序的影响而中断时，该变体的输出时间将被延长，从而干扰表决器的表决结果并有可能产生误报。同样地，中粒度的基于系统调用的表决器产生误报的原因通常在于对文件的操作，当变体中含有对文件读的操作时，假设其中一部分变体完成了读操作，此时若出现第三方程序需要对文件进行写操作，则由于读写文件操作的互斥性，剩下的未完成读文件变体的执行时间必然增加。最后，细粒度的基于指令的表决器产生误报的原因不仅包含以上的同步互斥，还包含指令序列等价性的定义。指令序列的识别常

用于基于语义的恶意代码研究(通常将恶意代码库与程序中的指令序列按照事先定义的语义等价规则进行对比)。由于定义语义等价规则具有一定难度且指令中存在副作用[31-32](除了对操作数参数有影响,还对其他寄存器产生附加影响),因此其规则的准确性与完备性在一定程度上决定了细粒度表决器的误报率。表 4 列出了指令副作用的一个代码实例。综上,表决器的误报在一定程度上与每次表决的等待时间有关,因此需要在误报率和等待时间之间做出折衷。设定表决等待时间阈值,能在一定程度上降低表决器的误报率。

表 4 指令副作用实例

Table 4 Example of side effect of code

原指令	等价指令段
SUBEAX,100	MOVECX,100 XCHGEAX,ECX LOOP -1

注:等价指令段中通过对 ECX 和 EAX 进行循环-1 操作,最终实现 EAX 减去 100

5.3 表决器的漏报

漏报在本文中是指对所有变体的恶意输入产生相同的输出从而绕过表决器的一致性判决。这种情况通常发生于所有变体都包含某一特殊种类的漏洞。通过上文提到的变体异构性与攻击特性相关可知,当变体异构的特征与攻击所利用漏洞的特性不一致时,表决器无法发现恶意行为。例如,针对软件指令地址空间的异构能够有效地防御代码重用攻击[33],但却不能抵抗远程代码注入攻击。

为了降低表决器的漏报率,可以通过尽量增大变体间的异构性特征来减少攻击者的攻击表面。第 2.3 节列举的大量软件多样化方法,通过利用上述不同的软件多样化方法,可以在一定程度上增大变体异构性特征,使其能够抵御更多类型的攻击。

本文提出的结合拟态思想的防御策略与多版本程序的区别主要在于变体的生成方式不同。本文中的程序的变体是由同一份源代码生成的,而不用对每个变体进行手工重写,这在一定程度上降低了变体开发的成本。与本文的变体生成方式不同,其他变体生成技术着眼于代码的多样化和内存分配的随机化,如指令集随机化、地址空间随机化等。本文通过动态不透明谓词的引入以及基本块扁平化实现对变体的多样化,动态不透明谓词选择的随机性与路径的多样性,使得本文提出的混淆方法可以生成多个具有不同控制流的变体。对于不同的两个变体,针对其基本块的混淆是随机选择的,并且由于引入了指令替换,可以肯定其在相同地址空间上的指令一定不同,因此通过利用程序地址空间进行的攻击可以被表决器检测到。

6 策略效果的评估

本文提出的结合拟态思想的防御策略以拟态 Web 服务器为效果验证平台。拟态 Web 服务器采用多版本服务器的设计方式实现了粗粒度的响应表决,通过对比不同服务器实例的输出结果对攻击行为进行判断。为了验证本文提出的多样化方法的有效性,利用其对一种 Web 服务器进行编译时混淆产生多个 Web 服务器变体,并用其替换原多版本 Web 服务器的设计方式。测试中分别采用 Nginx 和 Apache 服务器源码作为测试服务器。由于本文提出的多样化方法只针对变体的地址空间和控制流进行异构,因此选用针对以上两个特征进行攻击的样例进行测试。从 exploit-db 漏洞库中获取相关服务器软件所有已暴露漏洞的总数,并选择与程序地址空间相关的攻击样例进行测试,其中包含针对特定地址的缓冲区溢出攻击、ROP 攻击和远程代码注入攻击等 7 个样例。表 5 列出了利用本文提出的多样化方法生成的变体能够抵御上述攻击的数量,其中 Apache 中有一例测试没有通过(CVE-2014-7810)。通过分析可知,该攻击方法利用了程序函数的逻辑漏洞,因此在所有变体中均有该漏洞存在。

表 5 基于混淆的多样化实现的拟态防御系统能够抵御的漏洞数

Table 5 The number of bugs defended by mimic system based on obfuscation

	漏洞总数	特定漏洞	有效数
Nginx	20	3	3(100%)
Apache	33	4	3(75%)

注:其中特定漏洞是指利用多样化方法的异构特征的漏洞

结束语 本文提出了一种新型的利用混淆实现程序多样化的方法,其对程序中的基本块进行克隆,并将克隆的基本块转换为语义相同但句法不同的基本块,通过复制程序的控制流使得攻击者通过动态分析无法获得程序的整体结构,并且随着混淆的不断迭代,攻击者破解程序的难度也将不断增大。

将基于混淆技术实现的程序多样化与拟态防御思想相结合,同时运行多个变体并对其输出进行表决,不一致的输出将被视为攻击。与其他以消除漏洞为手段的安全防护措施不同,拟态防御思想承认漏洞是不可避免的,并通过阻止程序漏洞功能的生效起到保护系统的作用。这种方法的主要优点是能够有效地发现漏洞并且阻碍漏洞的传播,包括一些 0day 漏洞,为抵抗大型类别的攻击提供了更加安全的保障;如果能够证明通过混淆产生的变体具有完全相同的功能,并且能够用于防御特殊攻击,则利用该方法可以肯定针对上述特殊攻击所属类的攻击均能够被检测到。

本文提出的多样化方法异构了变体的地址空间和控制流特性,利用以上两种特征进行的攻击都将被检测到。未来的工作包括研究其他变体异构性技术,以及变体异构性与攻击类型之间的对应关系。

参 考 文 献

[1] SCHRITTWIESER S, KATZENBEISSER S, KINDER J, et al. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? [J]. ACM Computing Surveys (CSUR), 2016, 49(1): 1-31.

[2] BORELLO J M, MÉ L. Code obfuscation techniques for metamorphic viruses[J]. Journal in Computer Virology, 2008, 4(3): 211-220.

[3] SASIREKHA N, HEMALATHA M. A Thorough Investigation on Software Protection Techniques against Various Attacks[J]. Bonfring International Journal of Software Engineering and Soft Computing, 2012, 2(3): 10-15.

[4] BHATKAR S, DUVARNEY D C, SEKAR R. Address Obfusca-

tion：An Efficient Approach to Combat a Broad Range of Memory Error Exploits[C]// Usenix Security. 2003：105-120.

[5] XU J，KALBARCZYK Z，IYER R K. Transparent runtime randomization for security[C]// International Symposium on Reliable Distributed Systems. 2003：260-269.

[6] BARRANTES E G，ACKLEY D H，PALMER T S，et al. Randomized instruction set emulation to disrupt binary code injection attacks[C]// Proceedings of the 10th ACM Conference on Computer and Communications Security. 2003：281-289.

[7] KC G S，KEROMYTIS A D，PREVELAKIS V. Countering codeinjection attacks with instruction-set randomization[C]// Proceedings of the 10th ACM Conference on Computer and Communications Security. 2003：272-280.

[8] ANCKAERT B，DE SUTTER B，DE BOSSCHERE K. Software piracy prevention through diversity[C]// Proceedings of the 4th ACM Workshop on Digital Rights Management. 2004：63-71.

[9] LARSEN P，HOMESCU A，BRUNTHALER S，et al. SoK：Automated software diversity[C]// 2014 IEEE Symposium on Security and Privacy. 2014：276-291.

[10] COLLBERG C，THOMBORSON C，LOW D. A taxonomy of obfuscating transformations[C]// Department of Computer Science. New Zealand，1997：1173-3500.

[11] COLLBERG C，THOMBORSON C，LOW D. Manufacturing cheap，resilient，and stealthy opaque constructs[C]// Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1998：184-196.

[12] BARAK B，GOLDREICH O，IMPAGLIAZZO R，et al. On the (im) possibility of obfuscating programs[C]// Annual International Cryptology Conference. 2001：1-18.

[13] WEE H. On obfuscating point functions[C]// Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing. 2005：523-532.

[14] BENDERSKY D，FUTORANSKY A，NOTARFRANCESCO L，et al. Advanced software protection now[OL]. http://www.zentralblatt-math.org/ioport/en/?q=an%3A05767133.

[15] LIN D，STAMP M. Hunting for undetectable metamorphic viruses[J]. Journal in Computer Virology，2011，7(3)：201-214.

[16] LINN C，DEBRAY S. Obfuscation of executable code to improve resistance to static disassembly[C]// Proceedings of the 10th ACM Conference on Computer and Communications Security. 2003：290-299.

[17] KULKARNI A，METTA R. A New Code Obfuscation Scheme for Software Protection[C]// IEEE International Symposium on Service Oriented System Engineering. 2014：409-414.

[18] WANG C，DAVIDSON J，HILL J，et al. Protection of software-based survivability mechanisms[C]// International Conference on Dependable Systems and Networks，2001(DSN 2001). 2001：193-202.

[19] SHACHAM H，PAGE M，PFAFF B，et al. On the effectiveness of address-space randomization[C]// ACM Conference on Computer and Communications Security. 2004：298-307.

[20] LARSEN P，BRUNTHALER S，FRANZ M. Security through diversity：Are we there yet? [J]. IEEE Security & Privacy，2014，12(2)：28-35.

[21] JUNOD P，RINALDINI J，WEHRLI J，et al. Obfuscator-LLVM：software protection for the masses[C]// Proceedings of the 1st International Workshop on Software Protection. 2015：3-9.

[22] SCHAEFER I，RABISER R，CLARKE D，et al. Software diversity：state of the art and perspectives[J]. International Journal on Software Tools for Technology Transfer，2012，14(5)：477-495.

[23] JOSEPH M K. Architectural issues in fault-tolerant，Secure Computing Systems[D]. Los Angeles：University of California at Los Angeles，1988.

[24] KNIGHT J C，LEVESON N G. An experimental evaluation of the assumption of independence in multiversion programming [J]. IEEE Transactions on software engineering，1986，SE-12 (1)：96-109.

[25] BITANSKY N，VAIKUNTANATHAN V. Indistinguishability Obfuscation from Functional Encryption[C]// IEEE 56th Annual Symposium on Foundations of Computer Science. 2015：171-190.

[26] ZHI W，JIA C F，LIU W J，et al. Branch Obfuscation to Combat Symbolic Execution[J]. Acta Electronica Sinica，2015，43(5)：870-878.

[27] WU J X. Mimic Security Defense in Cyber Space [J]. Secrecy Science and Technology，2014，10(1)：4-9. (in Chinese)
邬江兴. 网络空间拟态安全防御[J]. 保密科学技术，2014，10(1)：4-9.

[28] PALSBERG J，KRISHNASWAMY S，KWON M，et al. Experience with software watermarking[C]// 16th Annual Conference Computer Security Applications，2000(ACSAC'00). 2000：308-316.

[29] MAJUMDAR A，MONSIFROT A，THOMBORSON C. On Evaluating Obfuscatory Strength of Alias-based Transforms using Static Analysis[C]// International Conference on Advanced Computing and Communications. 2006：605-610.

[30] YADEGARI B，JOHANNESMEYER B，WHITELY B，et al. A generic approach to automatic deobfuscation of executable code [C]// 2015 IEEE Symposium on Security and Privacy. 2015：674-691.

[31] SCHRITTWIESER S，KATZENBEISSER S，KIESEBERG P，et al. Covert Computation—Hiding code in code through compile-time obfuscation[J]. Computers & Security，2014，42(4)：13-26.

[32] SEBASTIAN S，KATZENBEISSER S，KIESEBERG P，et al. Covert computation：hiding code in code for obfuscation purposes[C]// Acm Sigsac Symposium on Information. 2013.

[33] SNOW K Z，MONROSE F，DAVI L，et al. Just-in-time code reuse：On the effectiveness of fine-grained address space layout randomization[C]// 2013 IEEE Symposium on Security and Privacy (SP). 2013：574-588.