编译支持的多变体融合执行设计与实现

李秉政[1], 张　铮[1], 马博林[2], 邢福康[1], 邬江兴[2]

[1]数学工程与先进计算国家重点实验室 郑州 中国 450001
[2]国家数字交换系统工程技术研究中心 郑州 中国 450002

摘要　多变体执行是由异构冗余变体并行执行来检测攻击的一种技术。作为一种主动防御技术, 多变体执行(multi-variant execution, MVX)通过并行运行的异构执行体之间一致性检查发现攻击行为。相较于补丁式的被动防御, MVX 可在不依赖攻击特征信息的情况下防御已知漏洞乃至未知漏洞威胁, 在网络安全领域具有广泛的应用前景。然而该技术在实际部署中, 由于多变体执行架构的边界不清晰, 将随机数、进程 PID 号等被动地纳入到了表决范围, 从而产生误报, 导致多变体执行无法兼容更多的软件系统。本文分析了多变体执行假阳问题产生的原因, 提出 I-MVX, 一种编译支持的多变体融合执行架构, 包括多变体同步编程框架和运行时同步模块。I-MVX 通过添加少量编译指示, 在编译阶段对程序内部引起假阳性问题的代码和变量进行插桩标识, 在运行时由监视器对变体进程内部和外部的变量及资源进行同步处理, 消除多变体执行中的误报。本文基于 LLVM/Clang 编译器和 Linux 内核加载模块设计实现了 I-MVX 的编译器和同步监视器。性能实验评估显示, I-MVX 在 SPEC 2006 基准测试集和 tinyhttpd 测试程序下引入的平均开销分别为 2.13%和 13.2%。多变体融合执行架构能够以少量的性能损耗为代价有效解决多变体执行中的假阳问题, 提升多变体执行的可用性。基于真实 CVE 漏洞的安全性测试表明, I-MVX 在保证多变体执行安全防御有效性基础上提升了多变体执行的兼容性。

关键词　多变体执行; 编译指示; 网络空间安全
中图法分类号　TP309　DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.07.09

Design and Implementation of Integrated Multi-Variant Execution Supported by Compiler

LI Bingzheng[1], ZHANG Zheng[1], MA Bolin[2], XING Fukang[1], WU Jiangxing[2]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China
[2] National Digital Switching System Engineering & Technological R&D Center, Zhengzhou 450002, China

Abstract　Multi-variant execution (MVX) is a technique in which heterogeneous redundant variants are executed in parallel to detect attacks. As an active defenses technique, MVX can detect attacks by monitoring the consistency of heterogeneous variants with parallel execution. Compared with patch-style passive defense, MVX can defend against known and even unknown vulnerabilities without relying on attack feature information, which has broad application prospects in the field of cyberspace security. However, in the actual deployment of the MVX framework, due to the unclear boundary of multi-variant execution, random numbers, process PID numbers, etc. are passively included in the voting range, resulting in false alarms, which make some software systems cannot be compatible with the MVX framework. We analyze the causes of the false positive problem of MVX and proposes I-MVX, a MVX framework supported by compilation, including MVX synchronization programming framework and a runtime synchronization module. I-MVX framework adds a small number of pragmas to instrument code and variables that cause false positives in the program during the compilation phase. At runtime, the monitor synchronizes the variables and resources from the inside and outside of variant processes to eliminate false alarms in the MVX framework. Based on the LLVM/Clang compiler and Linux kernel loading module, we design and implement the I-MVX compiler and synchronization monitor respectively. Performance experimental evaluation indicates that the average overhead introduced by I-MVX under the SPEC 2006 benchmark and Tinyhttpd program is 2.13% and 13.2%, respectively. At the cost of a small amount of performance loss, the integrated multi-variant execution framework can effectively solve the false positive problem in the MVX framework and improve the usability of the MVX framework. The security experiments based on real CVE vulnerabilities show that I-MVX improves the compatibility of multi-variant execution on the basis of ensuring the effectiveness of multi-variant execution security defense.

Key words　multi-variant execution; compiler pragma; cyberspace security

通讯作者: 张铮, 博士, 副教授, Email: ponyzhang@126.com。
本课题得到国家自然科学基金项目(No. 61521003)与国家重点研发项目(No. 2018YF0804003, No. 2017YFB0803204)资助。
收稿日期: 2021-06-15; 修改日期: 2021-09-22; 定稿日期: 2022-05-11

1 引言

系统级软件开发过程中, 由于编程语言(如 C 和 C++等)的局限性和软件开发者的安全知识缺乏等因素, 软件不可避免地出现诸如缓冲区溢出[1]、悬空指针[2]和内存泄漏[3]等漏洞。同时, 在软件开发过程中, 使用第三方库、借鉴开源代码等代码复用方式虽然能够显著提升开发效率, 但也带来了软件同质化的问题。OpenSSL 是一个开放源代码的软件库包, 被广泛应用在互联网程序和网站上, 其加密代码中一种名为 Heartbleed 的严重安全漏洞(CVE-2014-0160)曾引发重大影响。相同的漏洞经代码复用方式被广泛传播, 给攻击者提供了便利, 并给网络安全带来极大风险。

研究人员提出了多种措施来防御软件漏洞。针对控制流劫持攻击, 早期研究提出了栈保护技术(StackGuard) [4]、数据不可执行(Data Execution Prevention, DEP)[5]、地址空间布局随机化(Address Space Layout Randomization, ASLR)[6]等防御方法, 能够有效抵御代码注入攻击, 但容易被 return-to-libcReturn[7]、面向返回编程(Return Oriented Programming, ROP)[8-9]等代码重用攻击手段绕过。控制流完整性防御(Control Flow Integrity, CFI)[10-11]在运行时严格按照预先定义的控制流图限制程序控制转移, 能够抵御代码注入、代码重用攻击和内存信息泄露, 但难以在性能和安全性两者间找到平衡。面向数据编程(Data-oriented Programming, DOP)[12]采用非控制数据攻击方式实现图灵完备的攻击, 可绕过细粒度的控制流完整性防御。攻防对抗双方此消彼长, 被动式防御方法往往容易被新的攻击绕过, 无法应对未知漏洞的威胁。

Cox[13]于 2006 年首次提出将多变体执行(Multi-Variant Execution, MVX)用于解决软件安全问题。作为一种主动防御方式, 多变体执行使用软件多样性技术生成变体集合, 将程序输入分发至多个功能相同, 结构不同的变体并行执行, 并设置检查点, 通过比较发现变体执行状态的不一致来检测攻击。攻击者必须在不触发表决检测的情况下同时对多变体执行架构中所有变体实施攻击。

近年来, 对多变体执行的研究主要围绕多变体执行安全防御架构机制的实现以及安全性和性能平衡等方面展开[14-16]。Österlund S 等人[17]首次将软件多变体执行防御技术应用于操作系统内核, 将内核内存空间分区, 采用地址空间布局随机化来构造两个内核多变体。在执行系统调用时两个内核同时处理, 对执行结果进行同步检查以判断是否存在内核内存泄露。Voulimeneas A[18]首次提出了分布式 ISA 异构平台 DMON 的多变体执行架构, 实现指令集异构执行, 并利用 arm 和 x86 不同的硬件安全机制, 进一步提升多变体执行安全性, 但存在通信成本和性能损耗大的问题。与之类似的 HeterSec[19]在异构的 ISA 上实现多变体执行。不同于 DMON 基于 ptrace 在用户层对变体进程进行监控和裁决, HeterSec 基于 Linux 内核修改, 增加内核动态加载模块支持用户层分布式变体进程的同步和通信。姚东等人[20]提出 MVX-CFI, 一种基于多变体执行架构的 CFI, 提高了 MVX 的执行性能。拟态防御[21-22]是我国科研团队提出的针对未知漏洞后门的主动防御技术, 核心思想是动态异构冗余(Dynamic Heterogeneous Redundancy, DHR), 相比 MVX, 拟态防御的动态性和反馈机制使拟态防御系统具有内生的测不准效应, 在裁决机制上更加完善。

多变体执行在内存漏洞方面具有较强的无需先验知识的入侵检测能力。然而, 目前研究人员多变体执行设计实现中的缺陷讨论不足, 多变体执行中存在的假阳性错误问题常常导致误报率居高不下, 可用性下降。Pina 等人[23]提出一种特定领域的语言(Domain-Specific Language , DSL), 用以解决不同版本变体之间因系统调用号执行顺序的不同导致不一致产生假阳性错误的问题。该 DSL 语言及算法提高了多变体执行的兼容性, 但仅解决了系统调用级的假阳性问题。Stijn 等人[24]在多变体执行中加入线程时钟同步并行机制, 采用锁步的方式对多线程中部分系统调用排序执行, 解决了多变体执行中线程调度和资源争用问题, 首次在软件多变体执行架构中支持多线程并行, 该多变体执行能够支持多线程的本质是对线程共享资源进行加锁, 阻塞线程的执行, 因此必然会导致性能的降低。

针对多变体执行技术存在的假阳问题, 本文提出一种编译支持的多变体融合执行架构, 称之为 I-MVX(Integrated MVX)在编译阶段对程序内部引起假阳性问题的代码和变量进行插桩标识, 在运行时由监视器对来自多个变体进程内部和变体进程外部的变量及资源进行同步处理, 消除多变体执行中的误报, 提升多变体执行技术的可用性。本文主要贡献有:

(1) 本文提出了多变体执行同步编程模型, 通过添加少量的编译制导语句即可完成对多变体执行中假阳代码的同步处理。

(2) 本文提出了一种多变体执行架构下变体进

程间数据与资源同步的解决方案, 首次在软件多变体执行架构中解决了随机数假阳问题, 扩展了多变体执行的应用范围。

(3) 本文基于 Clang 编译器和 Linux 内核加载模块设计实现了一个多变体融合执行编译器和运行时监视器, 实验评估结果表明, 以少量的性能损耗为代价, 多变体融合执行方法能够有效解决多变体执行中的进程内部对象和外部对象引起的假阳问题, 提升多变体执行的可用性。

文章结构安排如下: 第 1 章介绍多变体执行研究背景; 第 2 章分析多变体执行假阳性问题产生的原因; 第 3 章提出多变体融合执行方案; 第 4 章阐述多变体融合执行实现; 第 5 章针对多变体融合执行架构做相应实验评估; 第 6 章进行全文总结。

2 多变体执行假阳问题分析

多变体执行是由异构冗余变体并行执行来检测攻击的一种技术。软件中存在的某些缺陷导致程序状态发生非正常转换(如缓冲区越界、内存泄漏等), 称之为软件漏洞。漏洞能被攻击者所非法利用, 进而控制系统和获取敏感信息。多变体执行技术在保证程序功能完备性的前提下, 通过异构程序的部分组件, 例如异构的虚拟内存空间等, 使得攻击者的漏洞利用过程在不同变体之间产生非一致的状态转换, 由监视器在程序检查点对多变体输出结果或中间状态进行比较, 即可检测攻击行为。而对于正常的用户输入, 程序状态转换是一致的, 能够产生一致的输出结果。

在不具备攻击输入的情况下, 变体程序并行执行时出现不一致的程序状态转换或输出结果, 监视器将程序正常执行识别为发生攻击, 称之为多变体执行的假阳问题。如图 1 所示, 变体进程 1 和变体进程 2 分别向操作系统请求进程号, 作为 libc 库随机数种子, 因为两个变体进程号不相同, 导致生成的随机数也不同, 在后续使用随机数, 如 HTTPS 协议交换随机数等应用场景时输出产生不一致, 监视器检测到该不一致, 被误认为发生攻击。

多变体执行当中变体程序异构组件的边界不清晰是导致假阳性问题产生的原因之一。现有的多变体执行技术以进程为冗余执行的主体, 以虚拟地址空间为异构对象来检测内存攻击, 其运行时的本质为一种执行相同任务且内存空间异构的进程级并行计算。多个变体进程并行执行过程中, 将部分进程内部和外部相关联的对象, 如进程内部随机数变量、进程外部由操作系统管理的进程 PID 号和文件描述符

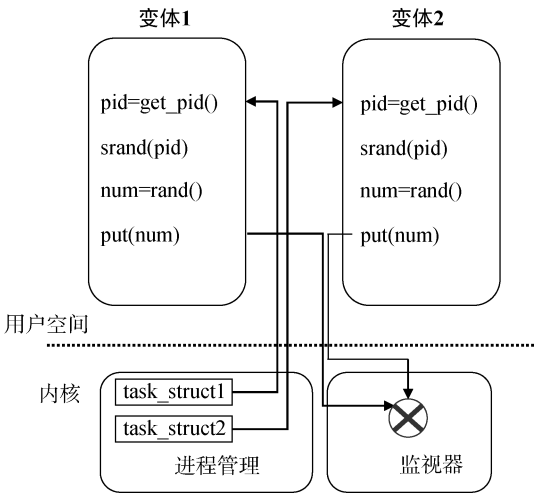

图 1 多变体执行假阳问题示意图

Figure 1 MVX false positive problem

等被动地纳入到了多变体执行的边界内, 参与监视器的表决过程, 从而被检测到不一致产生误报。解决假阳性问题的关键就在于确定多变体执行的边界。对于边界外的对象, 需要进行同步操作, 消除多变体执行中合法的不一致。

3 多变体融合执行方案设计

基于上述分析, 本文提出了一种多变体融合执行方法, 用以解决多变体执行中的假阳问题。多变体融合执行, 是指在多变体执行架构中, 对部分因进程级冗余执行产生假阳问题的变体进程内部和外部的对象进行同步处理, 减少误报, 使得多变体执行系统中多个变体进程对于外部正常输入能够协同一致, 表现为一致性的输出结果, 而对于攻击输入则出现不一致的执行状态, 通过监视器的比较发现攻击。假阳错误主要来源分为多变体进程外部对象和内部对象两种类型, 需要分别选择合适的同步点并设计相应的同步处理方法。

3.1 多变体进程外部对象

多变体执行是一种运行时的防御技术, 当变体程序被操作系统装载到内存并分配给它一定资源后, 此时可称为变体进程。变体进程的执行一方面受到监视器的控制, 另一方面也作为一个进程实体, 与操作系统进行交互来使用运行时的资源。变体进程与操作系统交互过程中的外部数据结构和相关系统调用, 如 Linux 内核中的 task_struct、文件描述符等, 是影响多变体执行的重要外部对象。

举例而言, 变体进程并行执行时打开文件并将返回的文件描述符 files_struct 打印至终端输出。由于进程的并行执行在微观层面的时间片上大概率是不

同步的, 变体进程执行 sys_open 系统调用的顺序不同, 则返回的 files_struct 中引用计数字段将产生不一致, 监视器输出表决过程中即产生误报。

对于多变体执行中进程外部对象引发的假阳问题, 设计融合执行方案如图 2 所示。在关键系统调用处设置同步点, 变体进程合路执行一次, 将执行结果返回给各个变体程序。针对不同的系统调用, 融合执行的实现细节不同。对于输入性系统调用, 例如打开文件、获取系统当前时间等操作由主进程执行, 主进程向从进程同步执行结果(文件描述符、系统当前时间等), 保证输入数据的一致性。对于输出型系统调用, 例如文件写操作, 在表决点完成表决后, 由主进程向对应文件(终端、网络接口等)执行写数据操作, 从进程执行空操作, 保持对外的一致输出。

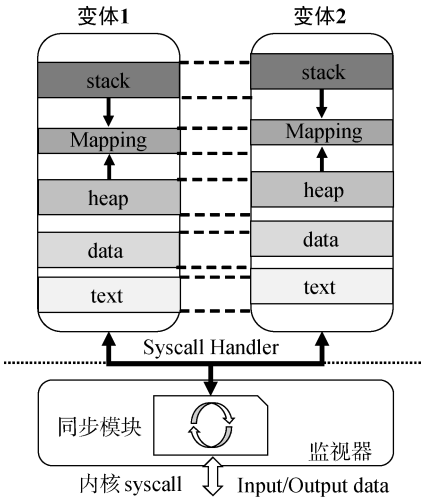

图 2 多变体融合执行示意图(进程外部对象)

Figure 2 Integrated MVX (for external objects of the processes)

3.2 多变体进程内部对象

进程内部对象, 包括函数、变量等, 在正常运行过程中由于多变体并行执行的时间序列不同或外部输入不同导致执行结果产生不一致, 也会引发假阳性错误。以随机数函数为例, 使用进程 PID 号作为随机种子生成随机数, 在两个变体进程上必然产生不一致, 而采用未赋值寄存器作为种子的随机数生成算法在不同变体中也会大概率导致随机数的不一致。对于进程内部产生不一致的对象, 需要进行同步操作。

变体进程拥有独立的地址空间(代码段、数据段等), 其中的二进制代码及数据由源代码经编译器转换生成, 是进程私有的, 在运行过程中, 操作系统无法解析程序内部的逻辑和所实现的功能, 因而无法确定同步点和同步对象。虽然编程人员在代码编写时掌握函数及相关变量的逻辑功能, 但由编程人员直接编写同步代码的方式, 需要修改源代码且难以推广应用。

本文设计多变体进程内部对象的融合执行方案如图 3 所示, 由编译器在编译阶段对关键代码及变量进行函数插桩的形式设置同步点, 使用动态链接的方式链接插桩函数, 由外部监视器完成同步操作, 并将同步结果返回变体进程。

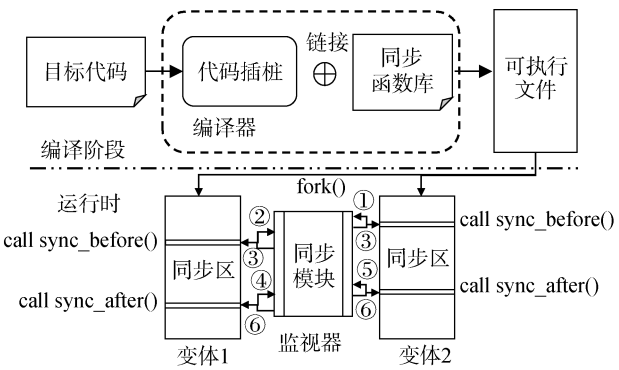

图 3 多变体融合执行示意图(进程内部对象)

Figure 3 Integrated MVX (for internal objects of the processes)

4 多变体融合执行架构实现

多变体执行假阳性问题是由于表决时将部分非一致性变量, 如随机数、端口号等纳入了表决范围, 并行执行的变体之间异步执行时, 监视器检测到非一致性变量而产生了误报。对随机数等关键变量及代码采用多变体变体融合执行方式, 降低多变体执行误报率, 可以提升多变体执行的可用性。多变体融合执行架构主要包含以下两个部分的实现。

编译器的主要工作: 在编译阶段根据编译制导语句的指示, 对产生误报的非一致性变量和关键代码地址进行分析提取, 并插桩同步处理函数。

运行时的主要工作: 基于 Linux 内核动态加载模块实现了一个运行监视器同步模块, 在运行时监视多变体进程的关键系统调用, 设置主从变体同步交换机制, 将执行结果拷贝至从变体进程。

4.1 多变体执行同步编程模型

对于多变体进程内部对象造成的假阳问题, 需要在分析理解程序代码的逻辑功能的基础上来确定产生不一致的变量及代码, 并在运行时进行同步处理。人工分析和修改程序源代码是一种实现方案, 以随机数函数为例, 编写随机数代理模块, 修改源代码从代理函数获取随机数, 由代理模块在运行时通过进程间共享内存通信等方式, 将生成的随机数分发给变体进程。该方案需要程序开发人员对多变体

执行有非常深刻的分析和理解, 不具有通用性, 且有可能在修改代码时引入新的 bug。

本文基于 LLVM 编译框架在 Clang 编译器前端实现基于编译制导的多变体同步编程模型,通过添加少量的编译制导和提供对应的编译器, 即可完成对多变体执行中假阳代码的同步处理。Clang/LLVM 版本为 10.0.0。其流程如图 4 所示。

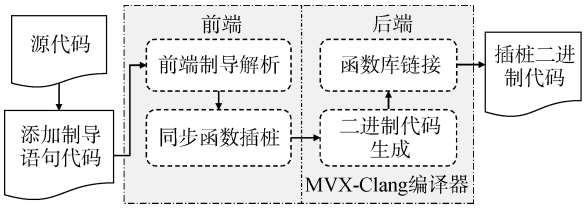

图 4 MVX-Clang 编译器编译流程

Figure 4 Workflow of MVX-Clang compiler

4.1.1 编译制导语句设计

#pragma 编译制导命令常用来给编译器传递额外的信息, 指示编译器做额外的操作。为了提高多变体执行的适配效率, 本文设计的多变体执行同步编程模型对代码的改动量较小, 具体的同步编译制导语句定义如表 1。

4.1.2 扩展编译器实现

MVX-Clang 编译器前端主要完成对编译制导语句的解析, 在语法分析树中添加随机数等非一致性变量相关的代码注解信息并对代码进行插桩处理。MVX-Clang 编译器后端将插桩 IR 代码与同步函数库链接编译成可执行代码。如图 5 所示, Clang 编译器输入为带编译制导语句的 C 语言程序源代码, 其编译步骤为:

表 1 多变体同步编译制导语句

Table 1 MVX synchronization compiler directives

编译制导语句类型	制导语句	同步操作含义
同步代码区	1. #pragma mvx sync before 2. #pragma mvx sync after	1.指定同步代码区域起始位置, 插桩前驱函数 2.指定同步代码区域结束位置, 插桩后驱函数
同步数据	#pragma mvx sync data(var_list,size,type)	指定同步变量名次, 大小, 类型等参数, 生成 API 函数参数

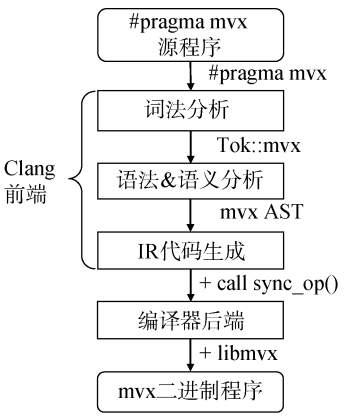

图 5 MVX-Clang 编译步骤

Figure 5 MVX-Clang compilation steps

(1) 词法分析器检测到对以#pragma mvx 字符开头的多变体同步制导语句, 生成词法单元 Token, 填入对应的语义信息, 并加入到当前编译单元 Token 流中, 传递给语法分析器。

(2) 语法分析器根据 Token 中的抽象符号所表示的不同语义, 生成抽象语法数 AST 对应节点并进行语法方面的检查, 语义分析器在生成 AST 节点阶段进行语义检查。

(3) IR 生成器遍历 AST 节点, 在同步代码位置进行插桩处理, 生成 IR 中间表示代码。

(4) 编译器后端将 IR 代码编译为可执行文件, 并链接同步函数库。

图 6 所示为编译制导语句编程和编译器插桩同步函数样例。编程人员在随机数生成代码前后添加编译指示, 由编译器完成对随机数代码前后插桩同步处理函数。之后由链接器链接同步函数库。

```
1.  static int RAND_bytes_loop(void*args)
2.  {
3.      loopargs_t*tempargs = *(loopargs_t**) args;
4.      unsigned char*buf = tempargs->buf;
5.      int count;
6.
7.      for (count = 0; COND(c[D_RAND][testnum]); count++)
8.          #pragma mvx sync before
9.          RAND_bytes(buf, lengths[testnum]);
10.         #pragma mvx sync after data(buf, lengths[testnum], char*)
11.     return count;
12. }
```

(a) MVX synchronization programming (before compiling)

```
1.  static int RAND_bytes_loop(void*args)
2.  {
3.      loopargs_t*tempargs = *(loopargs_t**) args;
4.      unsigned char*buf = tempargs->buf;
5.      int count;
6.
7.      for (count = 0; COND(c[D_RAND][testnum]); count++)
8.          sync_op_before();
9.          RAND_bytes(buf, lengths[testnum]);
10.         sync_op_after(buf, lengths[testnum], char*);
11.     return count;
12. }
```

(b) Instrumented sync_op (after compiling)

图 6 #pragma 编译指示和插桩函数示意图

Figure 6 #pragma directives and instrumented functions

4.2 运行时监视器同步模块

多变体执行中监视组件在整个架构中起着举足轻重的作用, 在运行时除了要对变体进程的输出进行表决之外, 还要执行变体进程创建、跟踪、退出等一系列管理功能。根据实现方式的不同, 监视器主要分为基于 Ptrace API 实现的监视器和基于内核模块实现监视器。Ptrace 实现的监视器与变体位于不同的进程空间, 它们的交互通过 Ptrace 调试接口进行。基于可加载的内核模块(Loadable Kernel Module, LKM)实现的监视器通过拦截变体进程的系统调用来监视多变体执行过程, 部署在内核中的监视器, 与变体共享相同的地址空间, 并且具有特权级的权限。本文基于内核动态加载模块实现的监视器来实现运行时同步模块, 其实现方案稍加改造, 也可应用于基于 Ptrace 调试接口的监视器中。

如图 7 所示, 采用环形缓冲区实现变体进程外部的数据同步, 在监视器模块中设置系统调用参数缓冲区和数据缓冲区两个环形缓冲区, 用于进程外部数据的同步。以 read 系统调用为例, 监视器拦截变体进程 read 系统调用后, 随机选择其中一个变体作为主变体, 执行 read 系统调用, 将输入数据放入环形缓冲区, 系统调用返回值放入环形缓冲区, 从变体读取环形缓冲区数据和系统调用返回值, 完成同步。对于输出型系统调用的同步点设置, 如 write 系统调用等, 在表决点之后进行。表决通过后, 由监视器选择一个变体进程向目标地址(文件)执行 write 系统调用。

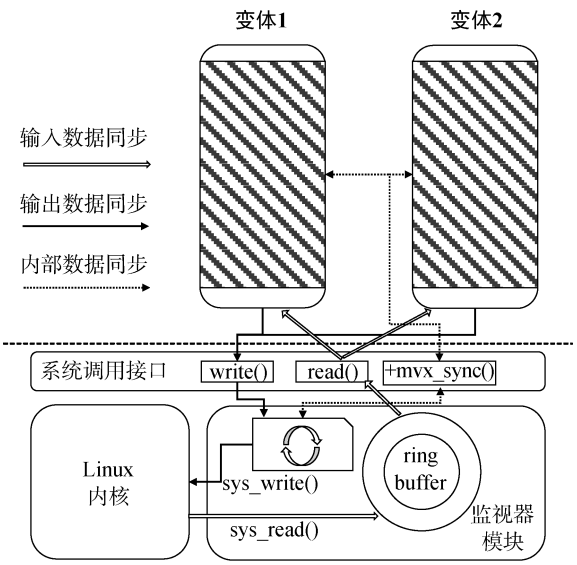

图 7 运行时模块架构图

Figure 7 Runtime module architecture

对于变体进程内部数据的同步处理, 本文新增一个系统调用作为专用的同步处理函数, 用于监视器模块和变体进程内插桩的同步函数进行同步交互。新增系统调用采用 Linux 内核动态加载模块覆盖系统调用表的方式实现, 不需要修改内核源代码。其同步流程为:

(1) 变体进程执行到前驱函数时, 发起系统调用;

(2) 监视器等待所有变体程序均运行到前驱函数并发起该系统调用后返回变体进程, 继续执行同步区代码;

(3) 变体进程执行到后驱函数时, 发起系统调用, 将需要同步的变量地址和数据类型作为系统调用的参数传递给监视器;

(4) 监视器根据编译制导指定的同步类型, 随机选择其中一个变体或使用主变体的执行结果, 拷贝到另外一个变体进程, 之后返回变体进程继续执行。

5 实验评估

本章节从四个方面对本文提出的多变体融合执行架构进行评估。首先, 本文采用 SPEC2006 基准测试集和 tinyhttpd 测试程序, 对多变体融合执行架构进行 CPU 密集型与 I/O 密集型两种类型程序的性能测试。其次, 为了衡量多变体融合执行代码插桩和运行时同步所引入的性能开销, 本文设计了微基准测试程序并对多变体融合执行架构模块进行深入的测试分析。之后, 本文基于真实 CVE 漏洞, 对多变体融合执行安全性进行评估。最后, 本文基于真实网关应用和服务器程序对 I-MVX 进行有效性实验评估。

5.1 正确性和性能评估

为了验证程序多变体融合执行架构的正确性, 评估多变体融合执行架构的性能, 本文采用 SPEC CPU Benchmark 2006[25] 中的 CINT 程序集和 tinyhttpd 测试程序, 运行 3 次基准程序和多变体执行程序来测量平均性能损耗。实验在 Ubuntu18.04 系统中运行, 内核版本为 Linux 5.4.0-42-generic, CPU 型号为 Intel Silver 4210, 内存容量为 32GB。

如图 8 所示, SPEC2006 基准测试中, 多变体融合执行架构的平均时间开销为 2.13%。与 GHUMVEE 等其他多变体执行相比性能损耗较低的原因主要有两点。一方面, 多变体融合执行架构采用内核加载模块实现对变体进程的运行时监视和同步, 与 Ptrace 实现方式相比, 减少了进程间通信和用户态与内核态频繁切换带来的性能开销; 另一方面, SPEC 基准性能测试中的程序为 CPU 密

集型程序, 其中的系统调用次数较少, 变体间同步的频率较低, 因而性能损耗较小。单个测试样例中, 存在比平均时间开销高的样例 429.mcf, 分析该测试样例为访存密集型程序, 在运行时需要最大约 1.7GB 的内存, 其访存次数频繁, 触发系统调用次数远大于其他程序。

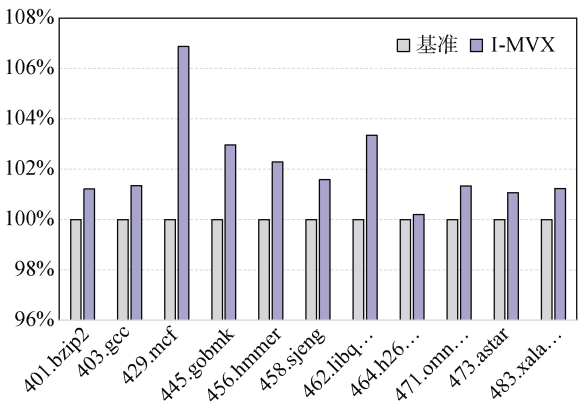

图 8 系统调用微基准测试结果

Figure 8 Syscall microbenchmarks result

如表 2 所示, tinyhttpd 测试中, 多变体融合执行架构对 tinyhttpd 导致的平均响应时间损耗为 13.2%, 对 tinyhttpd 导致的吞吐量损耗为 18.3%。tinyhttpd 测试程序为 I/O 密集型程序, 其系统调用次数远大于 SPEC2006 基准测试集中的 CPU 密集型程序, 在系统调用处进行表决和同步的概率增加, 因而时间损耗和 I/O 吞吐量损耗较大。

5.2 微基准测试评估

为了评估多变体融合执行架构中编译插桩同步及运行时同步模块对程序性能的具体影响, 本文设计了两组微基准测试来分析多变体融合执行架构引入的性能损耗。

首先, 本文采用 OpenSSL 开源加密库中的随机数测试代码来测试编译插桩同步对程序性能带来的损耗, 版本号为 openssl-1.1.1f。openssl speed 为 OpenSSL 用于测试加密算法性能的应用程序, openssl speed rand 命令可测试随机数生成算法的性能, 其测试过程为循环执行 RAND_bytes()函数并统计每秒处理的字节数。使用本文提供的 MVX-Clang 编译器编译 OpenSSL 代码, 对 speed.c 代码中 RAND_bytes()函数进行插桩同步处理, 作为微基准测试的测试程序。测试结果如表 3 所示, 平均性能损耗为 1.42%, 表明插桩的同步函数及监视器运行时同步模块对于程序性能的影响较小。

表 2 tinyhttpd 测试结果

Table 2 Tinyhttpd experiment result

测试程序	测试样例		响应时间(ms)	吞吐量
	Samples	Error %	Average	Transactions/s
基准	1000	0.00	157.6	641.85
I-MVX	1000	0.00	164.17	524.11

表 3 OpenSSL Rand 算法性能测试

Table 3 OpenSSL Rand performance

随机字节数 (bytes)	测试程序(性能为每秒处理的字节数, 单位 kbytes/s)	
	基准	I–MVX (%)
16	12353.61	12343.30 (–0.89%)
64	47902.84	46486.92 (–2.96%)
256	154309.54	152463.27 (–1.20%)
1024	353336.89	350433.91 (–0.91%)
8196	560871.60	559946.49 (–1.78%)
16384	584765.85	586552.70 (–0.80%)

其次, 为了评估运行时同步模块在系统调用级的影响, 本文本文选取 3 个代表性的系统调用, 设计微基准测试程序, 分别在基准环境和多变体融合执行架构下执行系统调用 1 万次, 对比其性能损耗。选取的系统调用为:

(1) sys_read(*fd*, *buf*, 512) 为需要输入同步的系统调用, 监视器将读取的数据同步至变体。

(2) sys_write(*fd*, *buf*, 512) 为需要输出表决和同步的系统调用, 这类系统调用在多变体融合执行架构中作为敏感系统调用, 监视器将不同变体的输出数据进行表决, 表决一致后进行同步输出。

(3) sys_sched_yield() 该系统调用不需要同步和表决。

测试结果如图 9 所示。在微基准测试中, sys_read 和 sys_write 系统调用受到两个变体进程同步操作的延迟影响, 达到了 1.51 倍和 1.27 倍的性能损耗, 而不需同步的 sys_sched_yield 系统调用则没有性能损耗。在实际的程序中, 触发多变体融合执行架构进行同步的频率由程序的类型决定, 对于计算密集型程序, 其 I/O 系统调用耗时占比较小, 因而程序性能受到的影响较小; 反之, 对于 I/O 密集型程序则性能影

响较大, 这也与本文 5.1 节中性能测试结果相符。

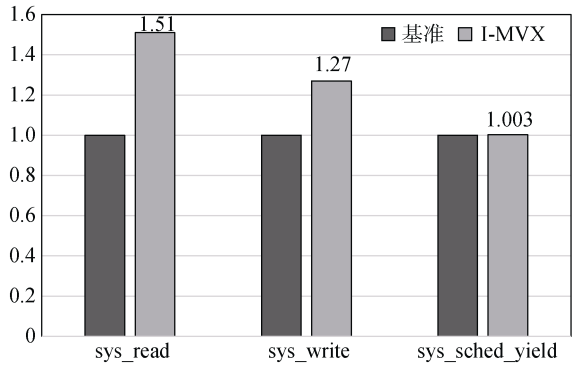

图 9　系统调用微基准测试结果

Figure 9　Syscall microbenchmarks result

5.3　安全性评估

为了验证本文提出的多变体融合执行架构防御攻击的有效性, 本文选取开源程序中 CVE 漏洞对多变体融合执行架构进行测试, 评估其防御效果。

wget 网络文件下载工具存在栈溢出漏洞, 编号为 CVE-2017-13089。由于源码中 fd_read 函数将 HTTP 响应数据(用户输入)写入 *dlbuf* 时并未检查长度参数 *contlen* 是否为负数, 攻击者可构造特定 long 长整型负数值, 覆盖写入缓冲区, 引发栈溢出漏洞。该漏洞可用于实施控制流劫持攻击。多变体融合执行架构中采用 ASLR 生成的两变体进程能够有效防御该漏洞。攻击者在覆盖写入缓冲区时, 由于两个变体的函数返回地址不同, 导致攻击者覆盖的地址在两个变体中必然指向的是不同的内容, 从而执行了不同的控制流, 最终在 I/O 输出系统调用中检测输出不一致, 发现攻击行为, 进而阻断攻击。

CVE-2014-0160 漏洞存在于 OpenSSL 程序中。该漏洞存在于 ssl/dl_both.c 实现 TLS 和 DTLS 的心跳处理逻辑时没有检测心跳包中的长度字段是否和后续的数据字段相符合, 导致内存区域后续的隐私数据被拷贝泄露给攻击者。多变体融合执行架构能够有效防御该漏洞。在泄露内存数据时, 由于异构变体进程间的地址空间不同, 泄露数据出现不一致, 最终在输出系统调用中被多变体融合执行监视器检测到并阻断信息泄露。

5.4　有效性评估

安全套接字协议 SSL(Secure Sockets Layer)及传输层安全协议 TSL(Transport Layer Security)是目前互联网应用最为广泛的加密通信协议, 为网络通信双方提供身份认证及加密数据传输功能。OpenSSL 是 SSL/TSL 协议实现的开放源码软件包, 采用 C 语言开发。OpenVPN 是一个基于 OpenSSL 加密库实现的应用层虚拟专用通道(VPN)软件, 其中使用了 OpenSSL 中的 SSLv3/TLSv1 协议函数库来实现身份认证、访问控制等功能。

某电网网关服务端程序基于开源 OpenVPN 和 OpenSSL 定制开发, 其中 OpenSSL 软件包通过调用函数 RAND_set_rand_method() 替换 OpenSSL 所提供的随机数生成方式, 实现了基于硬件方式(随机数发生器)来生成随机数, 具有更高的安全性。将网关服务端程序部署于 MVX 系统时, 由于两个网关变体进程在 TLS 双向认证过程中分别调用随机数生成函数 RAND__bytes()生成不同的随机数, 进而加密生成不同的预主密钥输出, 在输出表决过程中被 MVX 监视器判断为不一致, 产生误报。而在 I-MVX 中, 对 OpenSSL 随机数生成函数添加编译指示。使用本文的 MVX-Clang 编译器进行编译, 并链接同步函数库, 在运行时由监视器对两个变体进程随机数进行同步处理, 能够消除因随机数产生的假阳问题。该 I-MVX 防御系统同样可以兼容其他基于 OpenSSL 中 SSL/TLS 协议的应用, 亦可在 SSL/TLS 协议的其他代码实现中通过添加少量编译指示的方式进行适配。

由于本文所提出的多变体同步编程模型需要在 C 语言源代码相应位置添加编译指示并经过定制编译器生成可执行文件, 故无法适用于无源码的应用程序。对于无源码的应用程序, 其内部函数及变量的同步关系不透明, 需要通过反编译手段作进一步的分析处理。同时, 本文所实现的编译器及同步库函数、监视器组件不依赖于指定的指令集, 故本文所述基于 x86 平台下的 I-MVX 实现方法同样可以延伸借鉴到其他诸如 arm、MIPS 等指令集架构的系统中。

6　结论

MVX 作为一种主动防御技术, 在网络安全领域具有广泛的应用前景。本文在分析多变体执行假阳问题产生原因的基础上, 设计实现了多变体融合执行架构 I-MVX, 分为多变体同步编程框架和运行时同步模块。编程人员通过添加少量编译指示对多变体进程内部对象插桩同步函数, 在运行时完成对多变体进程内部和外部对象的同步处理, 消除多变体执行中的误报, 有效提升了多变体执行技术的兼容性。通过实验验证了 I-MVX 的防御有效性, 并且只对 CPU 密集型和 I/O 密集型程序分别增加 2.13%和 13.2%的性能损耗。

本文提出的 I-MVX 架构, 其设计方案具有通用性, 不局限于本文所使用的 Clang 编译器和 Linux 内

核模块监视器, 同样可以基于 gcc 等开源编译器实现多变体同步编程模型, 基于 ptrace 的 MVX 监视器实现运行时同步模块。

存在不足的是, 在 I-MVX 架构中, 我们对主要功能进行了实现和验证, 还有诸如多变体执行中的多线程支持、无源码程序的分析、同步算法的改进等细节需要在未来的研究工作中进一步完善。同时, 多变体执行与拟态防御思想相结合来实现软件安全防御, 将拟态防御的 DHR 构造引入到进程执行过程中, 在异构冗余的基础之上进行表决, 称之为拟态 2.0, 可为拟态防御研究提供新的思路。

致　谢　在此, 向评阅本文的审稿专家表示衷心的感谢, 向对本文工作给与支持和指导的同行, 尤其是课题组的老师表示感谢。

参考文献

[1] Larochelle D, Evans D. Statically detecting likely buffer overflow vulnerabilities[C].*Proceedings of the 10th conference on USENIX Security Symposium*, 2001: 14-14.

[2] Pincus J, Baker B. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns[J]. *IEEE Security & Privacy*, 2004, 2(4): 20-27.

[3] Xie Y C, Aiken A. Context- and Path-Sensitive Memory Leak Detection[C]. *The 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, 2005: 115-125.

[4] Cowan C, Pu C, Maier D, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks[C].*USENIX security symposium*, 1998, 98: 63-78.

[5] Molnar I. Method and Apparatus for Creating an Execution Shield: US20040250105[P]. 2004-12-09.

[6] Bhatkar S, DuVarney D C, Sekar R. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits[C].*USENIX Security Symposium*, 2003, 12(2): 291-301.

[7] Tran M, Etheridge M, Bletsch T, et al. On the Expressiveness of Return-into-Libc Attacks[M]. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 121-141.

[8] Shacham H. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)[C]. *The 14th ACM conference on Computer and communications security*, 2007: 552-561.

[9] Roemer R, Buchanan E, Shacham H, et al. Return-Oriented Programming: Systems, Languages, and Applications[J]. *ACM Transactions on Information and System Security*, 2012, 15(1): 2.

[10] Abadi M, Budiu M H, Erlingsson Ú, et al. Control-Flow Integrity Principles, Implementations, and Applications[J]. *ACM Transactions on Information and System Security*, 2009, 13(1): 4.

[11] Zhang C, Wei T, Chen Z F, et al. Practical Control Flow Integrity and Randomization for Binary Executables[C]. *2013 IEEE Symposium on Security and Privacy*, 2013: 559-573.

[12] Hu H, Shinde S, Adrian S, et al. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 969-986.

[13] Cox B, Evans D, Filipi A, et al. N-Variant Systems: A Secretless Framework for Security through Diversity[C]. *The 15th conference on USENIX Security Symposium - Volume 15*, 2006: 105-120.

[14] Volckaert S, Coppens B, Voulimeneas A, et al. Secure and efficient application monitoring and replication[C].*2016 {USENIX} Annual Technical Conference*, 2016: 167-179.

[15] Hosek P, Cadar C. VARAN the Unbelievable[J]. *ACM SIGARCH Computer Architecture News*, 2015, 43(1): 339-353.

[16] Koning K, Bos H, Giuffrida C. Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization[C]. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016: 431-442.

[17] Österlund S, Koning K, Olivier P, et al. KMVX: Detecting Kernel Information Leaks with Multi-Variant Execution[C]. *The Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019: 559-572.

[18] Voulimeneas A, Song D, Parzefall F, et al. DMON: A Distributed Heterogeneous N-Variant System[J]. *arXiv preprint* arXiv: 1903.03643, 2019.

[19] Kim S H, Ravindran B. A Framework for Software Diversification with ISA Heterogeneity[C]. *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020: 427-442.

[20] Yao D, Zhang Z, Zhang G F, et al. MVX-CFI: A Practical Active Defense Framework for Software Security[J]. *Journal of Cyber Security*, 2020, 5(4): 44-54.
(姚东, 张铮, 张高斐, 等. MVX-CFI: 一种实用的软件安全主动防御架构[J]. *信息安全学报*, 2020, 5(4): 44-54.)

[21] Wu J X. Research on Cyber Mimic Defense[J]. *Journal of Cyber Security*, 2016, 1(4): 1-10.
(邬江兴. 网络空间拟态防御研究[J]. *信息安全学报*, 2016, 1(4): 1-10.)

[22] Tong Q, Zhang Z, Zhang W H, et al. Design and Implementation of Mimic Defense Web Server[J]. *Journal of Software*, 2017, 28(4): 883-897.
(仝青, 张铮, 张为华, 等. 拟态防御 Web 服务器设计与实现[J]. *软件学报*, 2017, 28(4): 883-897.)

[23] Pina L, Grumberg D, Andronidis A, et al. A {DSL} Approach to Reconcile Equivalent Divergent Program Executions[C].*2017 {USENIX} Annual Technical Conference*, 2017: 417-429.

[24] Volckaert S, Coppens B, de Sutter B, et al. Taming Parallelism in a Multi-Variant Execution Environment[C]. *The Twelfth European Conference on Computer Systems*, 2017: 270-285.

[25] Henning J L. SPEC CPU2006 Benchmark Descriptions[J]. *ACM SIGARCH Computer Architecture News*, 2006, 34(4): 1-17.

李秉政 于 2019 年在信息工程大学计算机科学与技术专业获得学士学位。现在信息工程大学网络空间安全专业攻读博士学位。研究领域为网络安全。研究兴趣包括:主动防御技术、软件安全。Email: francisleeha@163.com

张铮 于 2006 年在信息工程大学计算机科学与技术专业获得博士学位。现任数学工程与先进计算国家重点实验室副教授。研究领域为网络安全、先进计算。研究兴趣包括:主动防御技术、高性能计算。Email: ponyzhang@126.com

马博林 于 2015 年在哈尔滨工业大学信息安全专业获得学士学位。现在信息工程大学网络空间安全专业攻读博士学位。研究领域为主动防御。研究兴趣包括: 拟态防御、移动目标防御。Email: msd_mbl@qq.com

邢福康 于 2019 年在东华大学大学软件工程专业获得学士学位。现在信息工程大学计算机技术专业攻读博士学位。研究领域为软件安全和主动防御。Email: k36666@163.com

邬江兴 现任国家数字交换系统工程技术研究中心主任, 教授, 博导, 中国工程院院士。研究领域为信息通信网络、网络安全。研究兴趣包括: 主动防御、交换技术与宽带信息网络、高效能计算。Email: 17034203@qq.com