编译支持的程序栈空间布局运行时随机化方法

朱鹏喆, 姚远, 刘子敬, 席睿成

引用本文

朱鹏喆, 姚远, 刘子敬, 席睿成.编译支持的程序栈空间布局运行时随机化方法[J]. 计算机科学, 2023, 50(8): 314-320.

ZHU Pengzhe, YAO Yuan, LIU Zijing, XI Ruicheng. Compiler-supported Program Stack Space Layout Runtime Randomization Method [J]. Computer Science, 2023, 50(8): 314-320.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220800098

编译支持的程序栈空间布局运行时随机化方法

朱鹏喆[1]　姚　远[2]　刘子敬[1]　席睿成[1]
1 信息工程大学网络空间安全学院　郑州 450001
2 网络通信与安全紫金山实验室　南京 211100
(1696207482@qq.com)

摘　要　多变体执行(Multi-Variant Execution,MVX)是目前最流行的主动防御技术之一,其通过并行运行一组功能等价的异构变体,检测不同变体之间不一致的状态转换实现对攻击行为的识别。多变体执行的防御效果在很大程度上依赖于程序变体之间的异构性,程序变体之间的异构性越高多变体执行的防御效果就越好。为了提高程序变体之间的异构性,文中提出了一种编译支持动静态相结合的程序栈空间布局随机化方法,该方法基于 LLVM 12.0 编译框架,首先在静态编译阶段根据外部输入获取函数识别程序中的关键变量,定位其栈空间分配指令,并在这些分配指令前添加额外的调用和分配指令,其次在程序运行阶段,利用静态编译时添加的指令在栈空间中的关键变量前进行内存块的随机化填充,从而实现程序运行时内存空间布局随机化。仿真实验结果表明,所提动静结合程序栈空间布局随机化方法可有效提高多变体执行程序间的异构性,对于基于程序内存地址溢出类攻击,不仅提升了其本身的攻击难度,也使得其不能通过不断试探程序地址来进行攻击,有效提高了程序的防御能力。

关键词:网络安全;主动防御;编译器;多变体执行;随机化
中图法分类号　TP311

Compiler-supported Program Stack Space Layout Runtime Randomization Method

ZHU Pengzhe[1],YAO Yuan[2],LIU Zijing[1] and XI Ruicheng[1]
1 College of Cyber and Space Security College,Information Engineering University,Zhengzhou 450001,China
2 Purple Mountain Lab of Network Communications and Security,Nanjing 211100,China

Abstract　Multi-variant execution is one of the most popular active defense technologies. MVX identifies attack behavior by running a set of functionally equivalent heterogeneous variants parallelly and detecting inconsistent state transitions between different variants. The defense effect of MVX depends on the heterogeneity between program variants in a large extent. Generally,the higher the heterogeneity between program variants,the better the defense effect of MVX. To improve the heterogeneity between program variants,this paper proposes a compiler-supported,dynamic and static program stack space layout randomization method. The method is based on LLVM 12.0 compilation framework. At static compile stage,the method identifies the key variables in program based on external input acquisition functions,locates their stack space allocation instructions,and adds additional call and allocation instructions before these allocation instructions. At program runtime,the method uses the instructions added during static compilation to randomly fill memory blocks before the key variables in stack space,realizing program memory space layout runtime randomization. Simulation experiment results indicate that the dynamic and static program stack space layout randomization method proposed in this paper can effectively improve the heterogeneity between MVX programs. For attacks based on program memory address overflow,the method not only increases their own attack difficulty,but also makes it impossible to conduct attacks by constantly testing program addresses,improving the defense ability of program effectively.

Keywords　Cyber security,Active defense,Compiler,Multi-variant execution,Randomization

1　引言

现代操作系统应用了许多防御措施用于抵御网络攻击,例如 Web 防火墙[1-2](Web Application Firewall,WAF)、地址空间布局随机化[3](Address Space Layout Randomization,ASLR)、控制流完整性防御[4-5](Control Flow Integrity,CFI)等。但这些防御手段是被动的,容易被攻击者针对性地绕过[6-7],且难以对未知的漏洞进行防护。

Cox 等[8]于 2006 年提出了多变体执行技术,用于防御软件漏洞,开创性地将防御手段由被动转为主动。多变体执行并行地运行一组功能相同但结构各异的程序变体,当接收到输入时,多变体执行会将输入分发到各个程序变体,这些程序变体各自独立地处理输入,在检查点处多变体执行会对所有程序变体的输出或执行状态进行检查,通过检测程序变体

到稿日期:2022-08-10　返修日期:2023-02-13
通信作者:姚远(1196207482@qq.com)

输出或执行状态的不一致来实现对攻击行为的检测。在理想情况下,多变体执行架构运行的一组变体功能相同,但结构完全不同,即任意一个程序变体中存在的漏洞只存在于当前变体中,任意两个程序变体中不存在相同的漏洞。当攻击者针对某一程序变体中的某一漏洞进行攻击时,攻击只能在该程序变体成功,在其他程序变体将攻击失败,因此程序变体将会产生不一致的输出或状态转换,这种不一致会在检查点被检测到,从而实现攻击行为的检测。

多变体执行实现了从被动防御到主动防御的进步,通过在检查点检测程序变体的输出或状态转换来实现对攻击行为的主动检测,相比传统的被动防御技术,多变体执行对攻击行为的防御具有更强的主动性,改变了被动防御技术原有的"攻击行为已生效-发现攻击行为-检查发现漏洞-修复漏洞"的防御模式[9],在攻击行为发生时即可发现并防御攻击行为,阻止攻击行为生效。同时,不同于被动防御技术只能对已知漏洞进行防御,多变体执行对未知漏洞同样具有较好的防御效果。当未知漏洞只存在于少数程序变体中时,针对未知漏洞进行的攻击行为将导致变体程序产生不一致的输出或状态转换,这种不一致被检测到即可实现攻击行为的检测和防御[10]。因此,多变体执行作为一种新型的主动防御技术,对于抵御网络攻击、保护网络空间安全具有十分重要的意义和价值。

由多变体执行的防御原理可知,多变体执行的防御效果依赖于程序变体之间的异构性:如果相同漏洞存在于多数程序变体中,那么针对此漏洞的攻击就会对多数程序变体生效,从而绕过多变体执行的防御机制;相应地,如果攻击针对的漏洞只存在于极少数程序变体中,那么多变体执行通过检测程序变体之间不一致的输出或状态转换,即可实现对攻击的检测和防御。因此,提升多变体执行防御效果的一项重要手段就是提高程序变体之间的异构性。程序内存空间布局异构性是程序异构性的重要部分[11-12],研究者们针对程序内存空间布局提出了许多程序异构方法。multicompiler[13]实现了静态的内存空间布局随机化,通过在编译阶段对程序进行处理来实现程序栈空间的随机填充,multicompiler 实现的内存空间布局随机化提高了程序变体内存空间布局的异构性,但这种随机化是静态的,程序栈空间中填充的大小在编译阶段已经被固定。TASR[14]实现了程序内存空间布局运行时随机化,将运行时程序产生输出和接收下一次输入之间的间隔作为进行内存空间布局随机化的时机。TASR 的随机化针对程序的代码段布局,本质上是在随机化时将程序代码段移动到随机选择的新地址空间,TASR 实现的程序代码段布局运行时随机化提高了程序变体内存空间布局的异构性,但此方法需要修改操作系统内核以提供运行时随机化的环境支持。Chameleon[15]也实现了程序内存空间布局运行时随机化,本质上是在运行阶段以固定的时间间隔随机打乱程序的代码段和栈空间布局,Chameleon 实现的程序内存空间布局运行时随机化提高了程序变体内存空间布局的异构性,但连续的运行时随机化会对程序运行进行较多干涉,且此方法需要基于 Ptrace 实现额外的运行环境支持。

栈空间是程序内存空间的重要部分,与位置无关的面向返回编程(PIROP)、面向数据编程(DOP)等攻击都依赖于对栈空间布局信息的窃取[15]。因此,提高程序栈空间布局的异构性是增强多变体执行防御效果的一项重要途径。本文提出了一种编译支持的程序栈空间布局运行时随机化方法,在编译阶段对程序进行特定处理,在运行阶段无需额外运行环境支持来实现程序栈空间布局随机化,从而有效提高程序栈空间布局的异构性。本文的主要贡献如下:

(1)针对程序栈空间布局提出了一种异构性量化模型,以数值反映程序变体的异构性,用于评估本文提出的程序栈空间布局运行时随机化方法为程序带来的异构性提高程度。

(2)提出了一种编译支持的程序栈空间布局运行时随机化方法,并基于 LLVM 12.0 编译框架实现方法原型。利用原型在编译阶段处理 C/C++程序,经过处理的程序在运行阶段能够实现栈空间布局的动态随机化。

(3)本文提出的随机化方法,能够有效提高程序在栈空间布局层面的异构性,增强程序对内存地址溢出类攻击的防御能力。

2 异构性量化模型

为了评估本文提出的编译支持的程序栈空间布局运行时随机化方法为程序带来的异构性提高程度,本文提出了一种异构性量化模型。该模型针对程序的栈空间布局,定量分析了程序异构性。

首先,模型基于程序栈空间布局的以下事实。

事实 1 程序栈空间在程序执行栈空间分配指令时进行分配,因此程序中栈空间分配指令与栈空间分配具有对应关系,栈空间分配指令的执行顺序即为栈空间的实际分配顺序。

事实 2 在程序栈空间生长方向确定时,程序栈空间布局可由栈空间内存块数量、内存块起始地址和内存块大小确定。

本文方法实现程序栈空间布局的随机化,随机化会改变程序栈空间布局,使程序的栈空间布局产生异构。由事实 2 可知,栈空间布局的改变会导致栈空间内存块数量、内存块起始地址和内存块大小中一个或多个属性的差异。因此,栈空间内存块数量、内存块起始地址和内存块大小是量化程序栈空间布局层面异构性的理想指标。

为了利用上述指标建立程序异构性量化模型,本文给出了如下定义。

定义 1(程序栈空间布局) 程序执行至某一指令时的栈空间布局。

定义 2(等价内存块) 在同一程序变体下,等价内存块为程序变体任意两次执行至同一指令时存储同一变量的内存块;在不同程序变体下,任意两个程序变体执行至同一指令时存储同一变量的内存块。

定义 3(内存块数量) 程序栈空间布局中包含内存块的数量。

定义 4(内存块起始地址) 程序栈空间布局中内存块起始位置的地址。

定义 5(内存块大小) 程序栈空间布局中内存块的大小。

定义 1—定义 5 对异构性量化模型的范围、对象和指标进行规定。内存块数量、内存块起始地址和内存块大小均为

异构性量化模型的指标，但这些指标具有不同的单位和量级，显然不能直接使用原始数据计算异构性数值。归一化是一种广泛使用的变量处理方法，用于消除变量量纲、统一变量数值范围[16]。本文的异构性量化模型基于归一化思想对量化指标进行处理。

对于内存块数量指标，内存块数量差异带来的异构性与内存块数量相关，且内存块数量差值相同时，内存块数量越多，异构性越小。因此，对于内存块数量指标，以内存块数量差值与内存块数量的比值为指标取值。

对于内存块起始地址指标，等价内存块起始地址差异带来的异构性与内存块大小相关，且等价内存块起始地址差值相同时，内存块大小越大，异构性越小。因此，对于内存块起始地址指标，计算每对等价内存块起始地址差值与内存块大小的比值并求和作为指标取值。

对于内存块大小指标，等价内存块大小差异带来的异构性与内存块大小相关，且等价内存块大小差值相同时，内存块大小越大，异构性越小。因此，对于内存块大小指标，计算每对等价内存块大小差值与内存块大小的比值并求和作为指标取值。

基于上述指标取值，程序异构性数值计算方法如式(1)所示：

$$D_{xy}=\rho_m\times\frac{|M_x-M_y|}{(M_x+M_y)}+\rho_a\times\sum_j\frac{|A_{xj}-A_{yj}|}{(S_{xj}+S_{yj})}+\rho_s\times\sum_j\frac{|S_{xj}-S_{yj}|}{(S_{xj}+S_{yj})}\tag{1}$$

其中，V_i 为程序变体；M_i 为 V_i 栈空间中的内存块数量；B_{ij} 为 V_i 栈空间中的内存块；A_{ij} 为 B_{ij} 的起始地址；S_{ij} 为 B_{ij} 的大小；B_{xj} 和 B_{yj} $(x\neq y)$ 为等价内存块；D_{xy} 为 V_x 和 V_y 的异构性数值；ρ_m 为内存块数量指标的异构性系数；ρ_a 为内存块起始地址指标的异构性系数；ρ_s 为内存块大小指标的异构性系数。

程序的异构性数值为内存块数量指标、内存块起始地址指标和内存块大小指标的取值乘上对应的异构性系数并求和。ρ_m，ρ_a 和 ρ_s 为异构性量化指标对应的异构性系数，可以根据程序类型等因素调整。本文中，异构性系数取值均为 1。

3 程序栈空间布局运行时随机化设计

为了提高程序异构性，本文提出了一种编译支持的程序栈空间布局运行时随机化方法。该方法通过扩展编译器来实现随机化功能：在编译阶段识别程序中的关键变量并对程序关键变量进行特定处理，从而在运行阶段实现程序栈空间的布局随机化，提高程序异构性。自然地，本文方法主要分为程序关键变量识别和程序关键变量处理两个部分。

3.1 程序关键变量识别

程序关键变量识别是实现程序栈空间布局运行时随机化的基础。内核信息泄露、控制流劫持、代码重用等攻击手段无论是执行实际的攻击行为还是执行窃取程序指令信息、嗅探程序地址空间布局信息等攻击前置行为，都依赖于程序输入，本质上是向程序输入恶意数据。如果一个程序不接收外部输入，那么实际的攻击行为和攻击前置行为都几乎不可能执行[17]。因此，程序内部接收外部输入的变量就是攻击的“门户”。基于上述内容对程序关键变量的定义如下。

定义 6 程序关键变量是程序内部接收外部输入(终端、套接字、文件等)的变量。

程序关键变量识别设计基于编译支持：由程序关键变量的定义可知，程序关键变量存放了程序的外部输入，因此程序关键变量必然是外部输入获取函数的参数或者接收了外部输入获取函数的返回值。在识别程序中外部输入获取函数的基础上获取函数的参数列表和返回值，结合函数类型即可实现程序关键变量的识别。在编译优化阶段，编译器能够识别特定函数、获取函数类型信息以及函数的参数列表和返回值。因此基于编译支持的程序关键变量识别设计如图 1 所示。

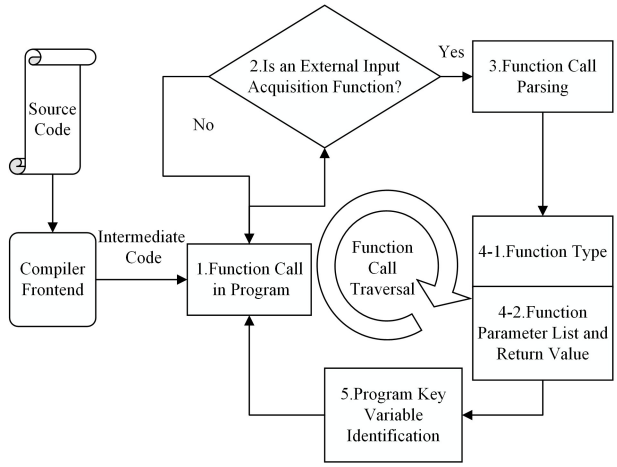

图 1 程序关键变量识别流程

Fig. 1 Identification process of program key variables

本文提出的程序关键变量识别设计基于编译支持，在编译阶段对程序中的函数调用进行遍历，识别程序中的外部输入以获取函数，在此基础上获取函数的参数列表和返回值，并结合外部输入获取函数的类型以完成程序关键变量的识别。

3.2 程序关键变量处理

在识别程序关键变量的基础上，为实现程序栈空间布局运行时随机化，需要对程序关键变量进行特定处理。本文提出的程序栈空间布局运行时随机化方法结合了编译阶段与运行阶段，在编译阶段对程序关键变量进行识别和特定处理；在运行阶段实现程序栈空间中无意义内存块随机填充。如前文所述，程序关键变量是攻击行为的“门户”，因此本文方法选择程序关键变量作为程序栈空间中无意义内存块的填充点，在编译阶段对程序关键变量进行特定处理，使得在运行阶段在程序栈空间中的程序关键变量前随机填充无意义内存块。

程序关键变量处理设计基于编译支持：在编译阶段，在程序关键变量识别的基础上进一步定位到程序中为程序关键变量在栈空间中分配内存的操作位置，在分配操作前添加内存块大小生成操作，使得程序在运行阶段动态地生成内存块大小；然后，以该大小作为参数添加栈空间内存分配操作，使得程序在运行阶段为程序关键变量实际分配栈空间前，先在栈空间中填充无意义内存块。运行时程序栈空间随机填充无意义内存块的效果如图 2 所示。

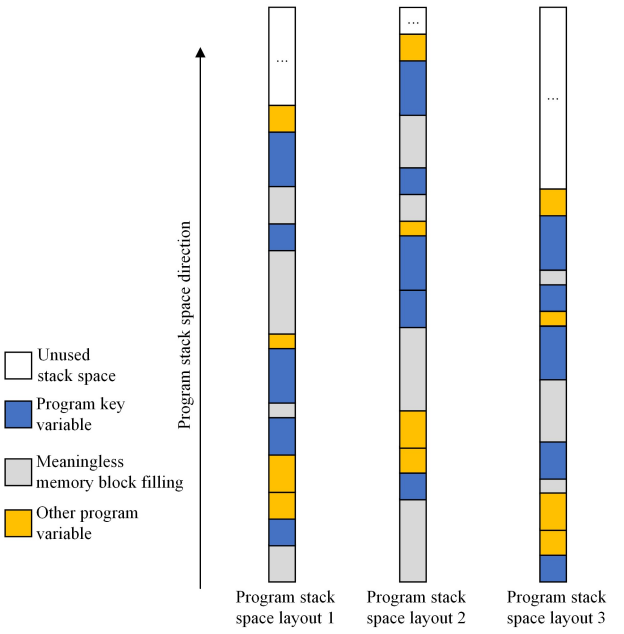

图 2 运行时程序栈空间随机填充效果图

Fig. 2 Schematic of random filling program stack space at runtime

由于程序栈空间中填充无意义内存块的大小是程序自身在运行时随机生成的，且程序栈空间填充是在运行时实际完成的，因此本文提出的程序栈空间填充方法是一种程序栈空间布局运行时随机化方法。与 multicomiler[13] 相比，multicomiler 也实现了程序栈空间填充，但填充大小由编译器在编译阶段确定，即编译完成后程序栈空间填充大小是固定不变的，这就使得攻击者通过足够次数的尝试能够感知栈空间填充并确定填充大小，导致填充失效。与 Chameleon[15] 相比，Chameleon 实现了程序栈空间布局运行时随机化，但它打乱程序栈空间布局，不仅破坏了程序栈空间的特性，且需要额外的运行时环境支持。本文提出的程序栈空间布局运行时随机化方法不仅保证了随机化的动态性，且程序栈空间随机填充无意义内存块完全由程序自身完成，不需要额外的运行环境支持。

攻击者执行攻击行为依赖充足的程序内部信息，因此需要反复向程序构造恶意输入以获取包含程序内部信息的输出，故会重复运行某一程序或者重复执行程序中的某段特定语句[18]。本文提出的程序栈空间布局运行时随机化方法在每次实际为程序关键变量分配栈空间之前都会填充随机大小的无意义内存块，在运行阶段改变了程序的栈空间布局，这不仅在一定程度上使程序原本的栈空间布局信息失效，而且有效地提高了程序的异构性。

4 程序栈空间布局运行时随机化实现

本节基于第 3 节中的程序栈空间布局运行时随机化方法，提出了一种方法的具体实现。本文方法实现了基于 LLVM 12.0 编译框架，针对 C/C++程序实现了栈空间布局运行时随机化。与方法设计对应，该方法的实现过程分为程序关键变量识别定位和程序关键变量前填充两个部分。

4.1 程序关键变量识别定位

程序的外部输入主要包括终端输入、文件读取和套接字传输，对于 C/C++程序，无论是自定义函数实现还是直接调用，程序中的外部输入获取函数都依赖于对 C/C++底层外部输入获取函数的调用。因此，程序关键变量必然是这些底层外部输入获取函数的参数或接收了这些底层外部输入获取函数的返回值。基于这一事实，本文的程序关键变量识别定位实现首先确定了 C/C++中获取终端输入、文件读取和套接字传输 3 类外部输入的底层函数。部分底层函数如表 1 所列。

表 1 C/C++部分底层外部输入获取函数

Table 1 Some low-level external input acquisition function in C/C++

Function Name	External Input Type	Acquisition Method
stdio. h scanf	terminal input	parameter
stdio. h getchar	terminal input	return value
stdio. h gets	terminal input	parameter
iostream cin	terminal input	parameter
stdio. h fread	file read	parameter
stdio. h fscanf	file read	parameter
stdio. h fgets	file read	parameter
sys/socket. h recv	socket transport	parameter
sys/socket. h recvfrom	socket transport	parameter

识别程序中外部输入获取函数调用即可识别程序关键变量。在 IR 指令级别，函数调用的形式为“‘函数返回值标识（若函数返回值类型为 void 则无此项）’‘call’‘函数返回值类型’‘函数名’‘函数参数列表（包括参数类型和参数标识）’”，函数调用的指令类型为“call”类型。程序关键变量的栈空间分配指令为变量分配栈空间，在识别程序关键变量的基础上，定位程序关键变量的栈空间分配指令。栈空间分配指令的形式为“‘分配栈空间标识’‘ alloca’‘分配栈空间类型’‘分配栈空间大小’”，栈空间分配的指令类型为“alloca”类型。在编译阶段，LLVM 编译框架能够遍历程序中的 IR 指令、获取 IR 指令的类型、获取 IR 指令中的各项参数。因此，基于 LLVM 编译框架的程序关键变量识别定位实现如算法 1 所示。

算法 1 程序关键变量识别定位算法

输入：IR 指令级别程序

输出：程序关键变量栈空间分配指令集合 S

1. 初始化：程序关键变量栈空间分配指令集合 S 置空，变量与对应栈空间分配指令的映射集合 M 置空，按照预先确定的底层外部输入获取函数集合，构造底层外部输入获取函数与对应程序关键变量的映射集合 F；
2. while（程序中所有 IR 指令）do
3. if（当前指令为“alloca”指令）then
4. 将被分配变量与当前指令的映射加入映射集合 M；
5. end if；
6. if（当前指令为“call”指令）then
7. 获取被调用函数的函数名并利用映射集合 F 判断；
8. if（被调用函数为底层外部输入获取函数）then
9. 利用映射集合 F 获取当前函数对应的程序关键变量；
10. 利用映射集合 M 获取当前程序关键变量对应的栈空间分配指令并加入集合 S；
11. end if；
12. end if；
13. end while.

在 C/C++程序中，程序关键变量一定是先分配栈空间

再使用，即程序关键变量的栈空间分配指令一定在使用程序关键变量的指令之前。因此，算法使用 LLVM 遍历程序中的 IR 指令。如果 IR 指令是“alloca”指令，则将被分配变量和“alloca”指令的映射加入映射集合 M。如果 IR 指令是“call”指令，则获取被调用函数的函数名并利用映射集合 F 判断被调用函数是否是底层外部输入获取函数。如果是，则利用映射集合 F 获取当前函数对应的程序关键变量。当前程序关键变量的栈空间分配指令必然存在于映射集合 M 中，获取当前程序关键变量的栈空间分配指令并加入集合 S。以算法 1 遍历程序中的 IR 指令，即可完成程序关键变量的识别和对应栈空间分配指令的定位。

4.2 程序关键变量前填充

在识别和定位程序关键变量栈空间分配指令的基础上，使用 LLVM 处理程序关键变量的栈空间分配指令，使得运行阶段在程序关键变量的栈空间前填充随机大小的无意义内存块。无意义内存块填充本质上是在栈中分配额外的空间，因此必然需要额外的栈空间分配“alloca”指令。为了在运行阶段随机生成无意义内存块填充的大小，需要程序在运行阶段执行额外的随机大小生成操作。本文使用 LLVM 编译框架在程序中添加额外的函数调用“call”指令，程序通过执行额外的随机大小生成函数实现在运行阶段随机生成栈空间分配大小。显然，“call”指令在“alloca”指令之前，“alloca”指令以“call”指令生成的大小作为分配栈空间的大小。为了使无意义内存块填充在程序关键变量的栈空间前，额外的“call”指令和“alloca”指令应被添加到程序关键变量的栈空间分配指令之前。使用 LLVM 编译框架处理程序关键变量的栈空间分配指令，实现运行阶段在程序关键变量的栈空间前填充随机大小无意义内存块的流程如图 3 所示。

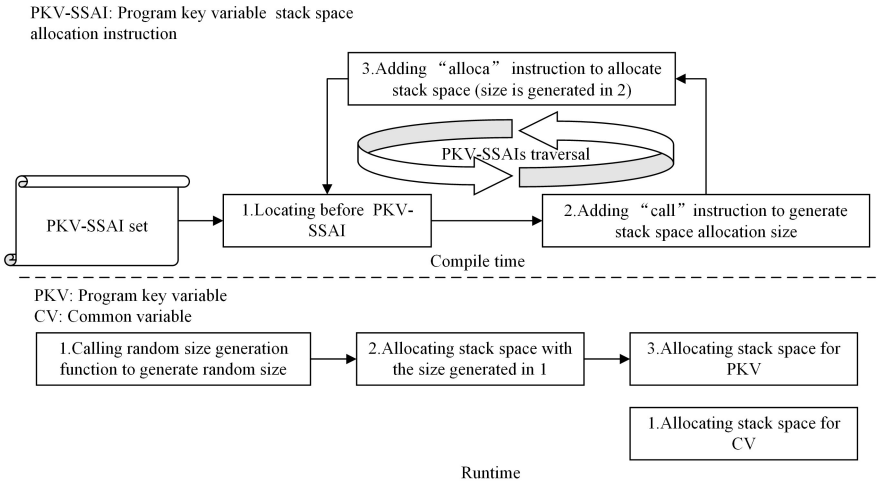

图 3 程序关键变量栈空间前的填充处理流程

Fig. 3 Filling before program key variable stack space processing flow

在编译阶段使用 LLVM 遍历 4.1 节中定位到的程序关键变量栈空间分配指令，将处理位置设置在栈空间分配指令之前。首先添加额外的“call”指令，调用随机大小生成函数，用于在运行阶段随机生成无意义内存块填充的大小。然后以“call”指令生成的大小作为参数添加额外的“alloca”指令，用于在运行阶段分配对应大小的栈空间，实现在程序关键变量的栈空间前填充无意义内存块。在运行阶段，经过处理的程序为程序关键变量分配栈空间前，调用随机大小生成函数生成无意义内存块填充的大小，并以此大小分配栈空间，然后程序实际为程序关键变量分配栈空间，从而实现在程序关键变量的栈空间前填充随机大小的无意义内存块。而对于其他普通变量，程序正常分配栈空间，不执行额外操作。通过在编译阶段使用 LLVM，在程序中程序关键变量的栈空间分配指令前添加额外的“call”指令和“alloca”指令，程序实现了在运行阶段在程序关键变量的栈空间前填充随机大小的无意义内存块。

本文提出的程序栈空间布局运行时随机化方法实现了基于编译支持，在编译阶段直接对原始 C/C++程序进行处理，不需要对原始 C/C++程序进行任何前置修改，即本文方法实现了以透明方式处理程序。同时，在本文方法的实现中，无意义内存块的大小生成和实际的栈空间填充均由程序自身在运行阶段完成，不需要额外的运行环境支持。

5 实验评估

本节对本文提出的程序栈空间布局运行时随机化方法进行了实验评估。为了评估方法为程序带来的异构性提高效果，实施量化分析实验，我们利用第 2 节提出的异构性量化模型对程序的异构性进行量化分析。为了评估方法为程序带来的防御能力提升，实施仿真实验，我们以内存地址溢出类攻击为例，从破坏程序栈空间的难度和破坏程序栈空间的可重复性这两方面来评估所提方法带来的防御能力的提升。

本文提出的程序栈空间布局运行时随机化方法对程序关键变量进行处理，因此测试程序的选取主要考虑程序关键变量。程序关键变量是根据变量用途产生的定义。在测试程序中，程序关键变量具有图 4 所示的伪代码形式。测试程序中，一个从外部输入获取函数中接收了外部输入的变量被视为一个程序关键变量。

本文的实验环境为：Ubuntu 20.04 系统、Linux 5.4.0-42-generic 内核版本、Intel Silver 4210 型号 CPU、32 GB 内存容量。

5.1 量化分析

量化分析实验用于评估本文方法提高程序异构性的效果。实验选取的测试程序包含图 4 所示形式的程序关键变量，测试程序包含的程序关键变量数量分别为 1，5，10，20，50，100。对于每个测试程序，将由本文方法处理过的测试程序执行 1 000 次，利用异构性量化模型计算与原始测试程序的异构性数值，计算 1 000 次异构性数值的算数平均值作为测试程序最终的异构性数值。实验结果如图 5 所示。

```
{
  ……
  variable V definition;
  ……
  V=External input acquisition function (parameter list);
  or External input acquisition function (…,V,…);
  ……
}
```

图 4 程序关键变量示意图

Fig. 4 Schematic of program key variables

图 5 显示，利用本文提出的程序栈空间布局运行时随机化方法处理程序能够大幅提高程序的异构性；随着程序中包含程序关键变量数量的增加，程序的异构性数值提高。这是因为本文方法以程序关键变量为单位进行随机化处理，程序关键变量数量越多，引入的随机化处理就越多，程序的异构性数值相应提高。

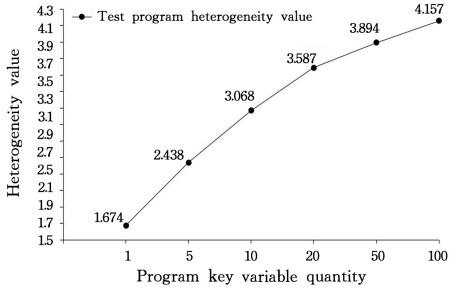

图 5 异构性量化分析实验结果

Fig. 5 Heterogeneity quantitative analysis results

5.2 仿真实验

仿真实验用于评估本文方法带来的防御能力提升，利用程序内存地址溢出类攻击的白盒测试，从破坏程序栈空间的难度和破坏程序栈空间的可重复性两个方面评估防御能力：白盒测试已知程序源代码，因此实验选取的测试程序只考虑包含的程序关键变量类型，包含的程序关键变量数量统一为 1。实验选取的测试程序分别包含接收终端输入的程序关键变量、接收文件读取的程序关键变量和接收套接字传输的程序关键变量，程序关键变量的形式符合图 4。实验利用本文方法处理测试程序，以正常编译的测试程序为对照。对于每组测试程序分别在单程序运行和多程序变体运行环境下实施对照实验，尝试在程序栈空间的指定位置覆盖指定数据。限制覆盖尝试次数不超过 1 000，在覆盖成功后验证覆盖是否可重复。仿真实验结果如表 2、表 3 所列。

表 2 单程序运行仿真实验结果

Table 2 Single program running simulation results

Program key variable type		Terminal input	File read	Socket transport
Randomized test program	Attempts before success	137	197	231
	Can repeat successfully?	No	No	No
Normal test program	Attempts before success	3	4	6
	Can repeat successfully?	Yes	Yes	Yes

表 3 多程序变体运行仿真实验结果

Table 3 Multiple program variants running simulation results

Program key variable type		Terminal input	File read	Socket transport
Randomized test program	Attempts before success	Failure	Failure	Failure
	Can repeat successfully?	No	No	No
Normal test program	Attempts before success	97	117	133
	Can repeat successfully?	Yes	Yes	Yes

仿真实验采用白盒测试，已知程序源代码，覆盖成功的尝试次数反映了破坏程序栈空间的难度；覆盖成功后是否可重复反映了破坏程序栈空间的可重复性。实验结果表明，对于程序内存地址溢出类攻击，本文方法为程序内存空间引入了动态随机性，不仅大幅提高了攻击自身的难度，且阻止了攻击输入的可重复性，使通过不断试探程序地址进行攻击成为不可能，有效提升了程序的防御能力。

结束语 作为一种先进的主动防御技术，多变体执行对于抵御网络攻击、保护网络空间安全具有重要意义。多变体执行的防御效果在很大程度上依赖于程序变体之间的异构性。本文针对程序的栈空间布局，提出了一种编译支持的程序栈空间布局运行时随机化方法，基于编译支持在编译阶段对程序中的程序关键变量进行识别和特定处理，实现在运行阶段为程序的栈空间布局引入随机化，有效提高了程序变体的异构性。

本文提出的随机化方法针对程序的栈空间布局进行随机化，有效提高了程序在栈空间布局层面的异构性，增强了程序对栈空间溢出类攻击的防御能力。但本文方法并未处理程序栈空间以外的其他内存空间（如堆区、数据区等），因此并未增强程序对其他内存空间攻击（如堆空间溢出攻击、内存泄漏攻击等）的防御能力。如何提高程序在其他内存空间的异构性需要在未来工作中进行进一步的研究。

参 考 文 献

[1] KRUEGER T，GEHL C，RIECK K，et al. TokDoc：A self-healing web application firewall[C]// Proceedings of the 2010 ACM Symposium on Applied Computing. 2010：1846-1853.

[2] CLINCY V，SHAHRIAR H. Web application firewall：Network

security models and configuration[C]// 2018 IEEE 42nd Annual Computer Software and Applications Conference(COMPSAC). IEEE, 2018: 835-836.
[3] LU K, SONG C, LEE B, et al. ASLR-Guard: Stopping address space leakage for code reuse attacks[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 280-291.
[4] ABADI M, BUDIU M, ERLINGSSON U, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Transactions on Information and System Security (TISSEC), 2009, 13(1): 1-40.
[5] BUROW N, CARR S A, NASH J, et al. Control-flow integrity: Precision, security, and performance[J]. ACM Computing Surveys(CSUR), 2017, 50(1): 1-33.
[6] HUND R, WILLEMS C, HOLZ T. Practical timing side channel attacks against kernel space ASLR[C]// 2013 IEEE Symposium on Security and Privacy. IEEE, 2013: 191-205.
[7] 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 (SP). IEEE, 2016: 969-986.
[8] COX B, EVANS D, FILIPI A, et al. N-Variant Systems: A Secretless Framework for Security through Diversity[C]// USENIX Security Symposium. 2006: 105-120.
[9] JIANG W, FANG B X, TIAN Z H, et al. Evaluating network security and optimal active defense based on attack-defense game model[J]. Chinese Journal of Computers, 2009, 32(4): 817-827.
[10] VOLCKAERT S, COPPENS B, DE SUTTER B. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution[J]. IEEE Transactions on Dependable and Secure Computing, 2015, 13(4): 437-450.
[11] ÖSTERLUND S, KONING K, OLIVIER P, et al. kMVX: Detecting kernel information leaks with multi-variant execution[C]// Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 559-572.
[12] VOULIMENEAS A, SONG D, LARSEN P, et al. dMVX: secure and efficient multi-variant execution in a distributed setting [C]// Proceedings of the 14th European Workshop on Systems Security. 2021: 41-47.
[13] HOMESCU A, JACKSON T, CRANE S, et al. Large-Scale Automated Software Diversity — Program Evolution Redux[J]. IEEE Transactions on Dependable and Secure Computing, 2015, 14(2): 158-171.
[14] BIGELOW D, HOBSON T, RUDD R, et al. Timely Rerandomization for Mitigating Memory Disclosures[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 268-279.
[15] LYERLY R, WANG X, RAVINDRAN B. Dynamic and Secure Memory Transformation in Userspace[C]// European Symposium on Research in Computer Security. Cham: Springer, 2020: 237-256.
[16] SINGH S, KRISHNAN S. Filter Response Normalization Layer: Eliminating Batch Dependence in the Training of Deep Neural Networks[C]// 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2020: 11237-11246.
[17] SONG D, LETTNER J, RAJASEKARAN P, et al. SoK: Sanitizing for Security[C]// 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 1275-1295.
[18] WANG Z, WU C, ZHANG Y, et al. Safehidden: an efficient and secure information hiding technique using re-randomization [C]// USENIX Security Symposium. USENIX Association, 2019: 1239-1256.

ZHU Pengzhe, born in 2000, postgraduate. His main research interests include compiler technology and multi-variant execution.

YAO Yuan, born in 1972, Ph.D, professor. His main research interests include parallel compilation and mimic defense.

(责任编辑:喻藜)