

基于二进制重写的软件多样化方法

何本伟, 郭云飞, 王亚文, 王庆丰, 扈红超

(信息工程大学, 河南 郑州 450001)

摘 要: 软件多样化是应对代码重用攻击的有效方法, 但现有软件多样化技术大多基于源代码实现, 相比二进制文件, 程序源代码并不容易获得。二进制文件难以做到精准拆卸、区分代码指针和数据常量, 使得对二进制文件的多样化转换有限, 难以产生足够高的随机化熵, 容易被攻击者暴力破解。针对此问题, 提出一种面向二进制文件的软件多样化方法, 指令偏移随机化, 该方法基于静态二进制重写技术在程序指令前以一定概率插入不同字节长度的无操作(NOP)指令, 不仅能够减少程序中非预期的 gadget 数量, 还使原指令地址发生随机偏移, 打乱程序原有的内存布局, 增加了代码重用攻击的成本。同时, 针对所提方法设计了基于“热”代码的优化策略, 通过动态插桩获得二进制文件中基本块的执行次数, 以此调整每个基本块中 NOP 指令的插入概率, 在执行频率更高的基本块中插入更少的 NOP 指令, 可以保证较低性能开销的同时产生更高的随机化熵。实验部分使用 SPEC 基准测试程序, 从性能开销、gadget 存活率、文件大小等角度对优化后的方法进行实例测试, 结果表明: 当插入概率为 15%时效果最好, 程序中 gadget 平均存活率趋于稳定且小于 1.49%, 增加攻击者重复利用相同 gadget 攻击链攻击难度的同时, 该安全性下仅额外增加了 4.1%的运行开销和 7.7%的文件膨胀率。

关键词: 软件多样化; 二进制重写; NOP 指令插入; 代码重用攻击

中图分类号: TP311.5

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2023024

Software diversification method based on binary rewriting

HE Benwei, GUO Yunfei, WANG Yawen, WANG Qingfeng, HU Hongchao

Information Engineering University, Zhengzhou 450001, China

Abstract: Software diversity is an effective defense against code-reuse attacks, but most existing software diversification technologies are based on source code. Obtaining program source code may be difficult, while binary files are challenging to disassemble accurately and distinguish between code pointers and data constants. This makes binary file diversification difficult to generate high levels of randomization entropy, and easily compromised by attackers. To overcome these challenges, a binary file oriented software diversification method was proposed based

收稿日期: 2022-05-31; 修回日期: 2023-03-02

通信作者: 何本伟, he_benwei@163.com

基金项目: 国家重点研发计划(2021YFB1006200, 2021YFB1006201); 国家自然科学基金(62072467)

Foundation Items: The National Key R&D Program of China (2021YFB1006200, 2021YFB1006201), The National Natural Science Foundation of China (62072467)

引用格式: 何本伟, 郭云飞, 王亚文, 等. 基于二进制重写的软件多样化方法[J]. 网络与信息安全学报, 2023, 9(2): 94-103.

Citation Format: HE B W, GUO Y F, WANG Y W, et al. Software diversification method based on binary rewriting[J]. Chinese Journal of Network and Information Security, 2023, 9(2): 94-103.

on static binary rewriting technology, namely instruction offset randomization. This method inserted NOP instructions of varying byte lengths before program instructions with a certain probability, reducing the number of unintended gadgets in the program and randomly offsetting the original instruction address. This disrupts the program's original memory layout and increases the cost of code-reuse attacks. At the same time, an optimization strategy based on hot code was designed for this method. The execution times of basic blocks in binary files were obtained by dynamic pile insertion, so as to adjust the NOP instruction insertion probability in each basic block. The higher the execution frequency, the fewer NOP instructions were inserted into the basic block, which can ensure lower performance overhead and produce higher randomization entropy. In the experimental part, the SPEC benchmark program was used to test the optimized method from the aspects of performance overhead, gadget survival rate and file size. The results show that a 15% insertion probability achieves the best effect, with an average gadget survival rate of less than 1.49%, increasing attackers' difficulty in reusing the same gadget attack chain. Furthermore, only a 4.1% operation overhead and 7.7% space overhead are added, maintaining high levels of security.

Keywords: software diversity, binary rewriting, NOP insertion, code-reuse attack

0 引言

当今信息技术飞速发展，计算机软件已渗透到人们生活的各个领域，不断地推动着社会的进步发展。然而，由于软件开发与分发机制中的“单一文化主义”^[1-2]，相同的程序运行在数百万甚至数亿台计算机上，软件中的一个漏洞就可能导致运行相同漏洞软件的所有计算机受到攻击，导致大规模的攻击发生，给系统安全、数据安全和隐私保护等带来了极大的威胁，甚至造成严重的损失。

为了解决同质软件带来的安全性问题，Cohen^[3]在 1993 年提出了软件多样化概念，将一个软件程序多样化为多个不同的实例，每个实例与源程序的功能相同但内在布局和结构不同，其目的是迫使攻击者为每个软件实例构造特定的攻击矢量，提高其攻击成本。随后，研究人员提出地址空间布局随机化（ASLR）^[4]技术，在每次执行时随机化堆、栈和共享库等的基地址，广泛部署到 Windows、Linux 和 iOS 等现代主流的操作系统中，以防御代码重用攻击。

近年来，研究人员基于编译器在源代码级别上实现了多种软件多样化方法^[5]，可以在不同生命周期以及不同粒度上对软件进行多样化处理。文献[6]重新分配 32 位 x86 ELF 文件中每个指令在地址空间的位置，并生成一组重写规则，这些规则描述了如何继续执行新位置的指令，每条指

令在地址空间的位置随机，具有很高的随机化熵，然而该方法的平均运行开销为 13%，难以运用到实际。文献[7]基于编译器实现了函数重排序、指令替换和控制流平坦等 6 种软件多样化技术，并通过内存中可跳转指令片段（gadget）的存活率来比较这些多样化技术对代码重用攻击的防御效果。文献[8]实现了一种在加载时对函数进行重排序的方法，源代码经过编译和链接后，在执行前会由新的加载器加载到内存，同时打乱函数的排列顺序，使程序在每次运行时有不同的内存映象。相关的开源集成工具很成熟，如 Obfuscator-LLVM^[9]、Tigress^[10]。然而，现阶段的软件多样化技术大多针对源代码实现，在实际软件项目中，软件的源代码并不总是容易获得，且二进制文件难以做到精准拆卸、区分代码指针和数据常量等反汇编^[11]，因此，如何在不依赖源代码情况下实现二进制级别的软件多样化，是现阶段研究的重点难题。

本文实现了一种面向二进制文件的细粒度多样化方法，指令偏移随机化，旨在为无法获得源代码的二进制文件提供随机化保护。具体方法是先通过二进制插桩技术获得基本块的执行次数，以此区分出程序中的“冷”“热”代码块^[12]；再利用二进制重写器对软件进行静态解析，获得函数、基本块和指令等要素的关键信息，如地址和相互关系；然后在“冷”“热”代码块^[12-14]中的指令之间以不同概率随机插入一些垃圾代码，如

NOP 等非语义指令,不影响程序的最终执行结果,从而随机化偏移程序原指令地址;最后调整程序内存布局,生成新的二进制文件。本文的主要贡献如下。

1) 提出了一种面向二进制文件的指令偏移随机化方法,在指令级别实现了细粒度的软件多样化。

2) 设计了基于基本块执行频率的指令偏移策略,不仅降低了转换的性能开销,还增加了随机化熵。

3) 对所提方法在安全性和运行开销方面进行了评估,实验结果表明,当插入概率约为 15% 时,效果最好, `gadget` 平均存活率趋于稳定且小于 1.49%,仅增加了 4.1% 的运行开销和 7.7% 的文件膨胀率。

1 相关技术

1.1 代码重用攻击

随着 DEP、NX (no-execute)、不可执行栈、等内存不可执行技术的广泛部署,攻击者以缓冲区溢出等形式向程序中注入的恶意代码不再被允许执行,很大程度上抑制了代码注入攻击。因此,攻击者提出了新的攻击方式——代码重用攻击,与注入式攻击不同,代码重用攻击不需要引入恶意代码,而是利用程序或库中已有的指令片段构造攻击,以实现恶意代码的功能,从而绕过了现有的安全机制。

代码重用攻击的演变主要分为 3 个阶段。第一阶段是函数级别的 Ret2libc 攻击^[15],通过重写指向程序重要的库函数的指针,如 `libc` 库中的 `system()` 函数,可以重构之前注入的恶意代码的操作,而无须注入任何代码。第二阶段利用以 `ret` 和 `jmp` 结尾的具有控制流转移功能的代码块,即 `gadget`,可以进行算术运算,进行从内存加载数据、向寄存器赋值等操作,将多条这样的配件串联起来可以形成 `gadget` 攻击链,这一阶段的代表攻击主要有 ROP 和 JOP 等^[16-18]。第三阶段攻击者提出 JIT-ROP 攻击^[19-20],由于软件多样化后程序中的 `gadget` 位置发生改变,ROP 和 JOP 等重用攻击很难发挥作用,而 JIT-ROP 通过程序在运行时泄露的任一内存地址获得所在页的起始地址,反汇编该页找到直接跳转指令,通过直接跳转地

址找到其他页地址,递归下去,可动态获得新的内存页并搜索其上的 `gadget` 构造攻击链。

总体来说,代码重用攻击获得内存中足够多 `gadget` 的位置,构造劫持控制流的攻击链,才能通过溢出漏洞将程序的控制流转移到攻击链上,达到攻击目的。

1.2 二进制重写

二进制重写技术是在没有源代码的情况下,在二进制文件中添加、删除和替换指令,对其进行转换,并且这种转换可以使二进制文件保持可执行。二进制重写广泛应用于程序性能优化、安全加固和漏洞修复^[11]等。本文通过动态二进制插桩和二进制提升两种重写技术对软件进行分析和转换,具体解释如下。

(1) 二进制提升

二进制提升是将二进制文件转换成更高级的中间语言表示,如 LLVM IR 和 VEX IR 等,然后通过其对应的开发平台进行分析和转换,最后将修改过的 IR 重编译成新的二进制文件。现有的二进制提升方法依赖于静态反汇编,而静态反汇编从根本上无法准确地建模间接控制流目标,区分代码指针和数据常量,或识别数据字节和指令字节之间的边界,难以对文件进行精确拆卸。本文利用二进制重写工具 `Egalito`^[21]对位置无关可执行程序 (PIE) 进行解析,提升到其中间语言表示 (EIR),EIR 是低级的 IR,提升结果可预测,与原始输入更接近,避免了反汇编的错误。

(2) 动态二进制插桩

此方法在运行时对二进制程序进行分析,在保证被测程序原有正常执行的情况下,将程序监控代码插入被测程序中,能在运行过程中获得程序的控制流和函数或基本块的执行信息等。转换过程发生在运行时,与程序运行会频繁切换,造成极大开销。动态二进制重写技术已相当成熟,如广泛应用的动态二进制重写工具 `Pin`^[22],具有丰富易用的 API,用户可自定义 `Pintool`^[23]对程序进行插桩分析,获得程序的运行信息。

1.3 软件多样化

软件多样化的主要思想是增加软件的不可知性,使软件在每个计算机系统或每次执行之间实现不同,以此抵御大规模代码重用攻击的发生。

在实施软件多样化的过程中，一个重要原则是不能改变软件的语义信息，即相同的输入产生相同的输出^[24]。Cohen^[3]提出程序“进化”这一新的防御策略，程序“进化”成与原始程序不同但语义相同的版本，随后演示了几种不同的多样化技术，如等效指令替换、指令重新排序和垃圾插入。通过添加新指令或改变现有指令来转换程序代码，对程序中指令的位置和顺序造成了巨大影响，增加了程序的随机性。

由于代码重用攻击需要知道程序中 gadget 的具体位置，程序在多样化技术转换后，攻击者很难执行相同的 gadget 攻击链重复进行攻击。如 ASLR 技术可以在程序每次加载时，使堆、栈和共享库等基地址随机分布，提高攻击成功难度，攻击者只能不断进行试错攻击。然而，现部署到系统的 ASLR 技术是程序级别粗粒度的多样化^[25]，攻击者可以通过地址泄露来推断程序各个段的布局信息。

2 指令偏移随机化

2.1 NOP 指令插入策略

本文选用不同字节长度的 NOP 指令及其等价指令作为垃圾代码，插入程序指令之间，使指令位置发生随机化偏移。NOP 是汇编语言的一个指令，除 PC 加 1 外，不执行任何操作，一般用来控制 CPU 的时间周期，达到时钟时延的效果。如表 1 所示，为了增加软件的异构程度，本文构建了一个 NOP 指令池，共有 10 条不同的 NOP 指令，分别来自 Inter 架构开发手册^[26]和文献[10]。插入的 NOP 指令除了需要程序的内存空间外，不会干扰程序的工作集，且具有较小的性能开销。

NOP 指令插入会改变程序中原有 gadget 地址，以此防御代码重用攻击。如图 1 所示，插入的 NOP 指令导致后续程序指令移位，随机化原

件中的 gadget 地址。NOP 指令会产生累积效应，造成后续的指令与原指令的偏移量不断增大。另外，由于 x86 架构指令长度是可变的，会产生非预期的 gadget，图 1 中指令 {call 609} 的最后一个字节编码 FF 和指令 {mov %rax, %rbx} 前两个字节编码 48 89 会组成新的汇编指令 FF 48 89 C3，由此产生非预期 gadgets {dec %rax, %rbx, ret}。NOP 指令插入通过在指令之间插入一条 3 byte NOP 指令 {nopl (%rax)}，隔绝了前后指令的联系，来消除这种小配件。

通过重写器 Egalito^[21]将函数、基本块和指令等要素的地址抽象为“链接 (link)”，这些 link 信息存储在 EIR 中。这样指令之间通过 link 相互关联而不是地址，可以在原程序指令之间任意添加和删除指令，而不用担心地址布局引起的错误。在实际插入中，设置 P_{nop} 表示在每个指令前插入 NOP 指令的概率，插入的 NOP 指令从 NOP 池中随机选择一条，然后生成该 NOP 指令的 link 信息存储到 EIR 中，当遍历完程序中的所有指令时，将转换后的 EIR 生成新的二进制文件。本文的 NOP 指令插入算法具体如下。

算法 NOP 指令插入

输入 待转换的 ELF 程序, NOP 指令池, P_{nop}

输出 转换后的 ELF 程序

Begin

1) EIR = EgalitoParse(ELF); //解析输入的 ELF 文件

2) FunctionList = Get(EIR); //找出文件中所有函数

3) foreach Fun in FunctionList; //遍历所有函数

4) foreach BB in Fun; //遍历所有基本块

5) foreach Ins in BB; //遍历当前基本

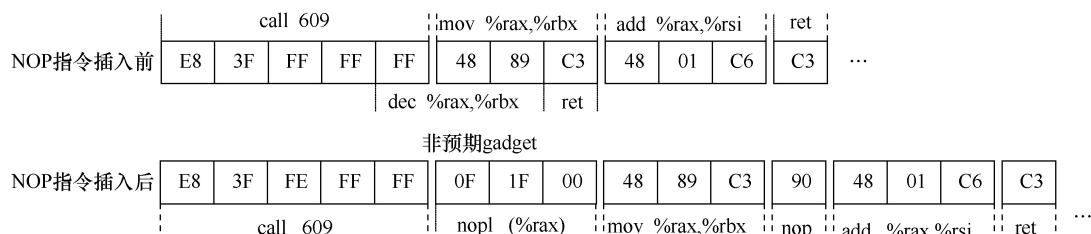


图 1 NOP 指令插入对 gadget 的影响

Figure 1 the influence of NOP instrumentation on gadgets

块中的指令

```

6)      if random(0,1) <  $P_{nop}$ ; //random
随机函数生成随机数如果小于预设  $P_{nop}$ 
7)      Insert a NOP  $\in$  NOPList; //从
NOP 池中随机选择一条插入
8)      end if
9)      end foreach
10)     end foreach
11) end foreach //NOP 指令插入完成
12) NewEIR = InsertNOP(EIR); //重新生成
EIR
13) NewELF = Compile(NewEIR); //生成新的
ELF 文件
End

```

表 1 NOP 指令池
Table 1 NOP instruction pool

字节长度/byte	编码	指令
1	90	nop
2	66 90	xchg %ax,%ax
3	0F 1F 00	nopl (%rax)
4	0F 1F 40 00	nopl 0x0(%rax)
5	0F 1F 44 00 00	nopl 0x0(%rax,%rax,1)
6	66 0F 1F 44 00 00	nopw 0x0(%rax,%rax,1)
3	48 89 ed	mov %rbp,%rbp
3	48 89 e4	mov %rsp,%rsp
3	48 8d 36	lea (%rsi),%rsi
3	48 8d 3f	lea (%rdi),%rdi

2.2 “热”代码优化

上述指令插入算法是以相同概率 P_{nop} 在每个指令前插入 NOP 指令,攻击者通过获得足够多的指令就可以计算出 NOP 的插入概率,以此获得每个指令的大致偏移量,并且,Homescu 等^[12]研究表明,在每个指令前以 50%的概率插入 NOP 指令,造成的开销已不适用于实际生产环境。为此,本文采用与文献[12]类似的优化方式,根据当前基本块的执行热度来更改 NOP 插入的概率,即在基本块执行次数多(“热代码”)处适当降低 P_{nop} 的值,在基本块执行次数少(“冷代码”)处适当提高 P_{nop} 的值,这样插入的指令虽然数量变化不大,但降低了 NOP 指令插入后的软件性能开销,而且

由于每个基本块插入的频率不尽相同,增加了每个软件变体的异构度。

在实际运行中,程序的大部分时间集中在少量的代码区域,根据程序的 2/8 分布原则,程序 20%的指令占据了 80%的执行时间。因此,首要任务是分析出二进制文件中“冷”“热”代码位置,本文以基本块的执行次数来区分“冷”“热”代码。基本块是程序中最大限度顺序执行的语句序列,且只有一个入口和出口,入口是其第一个语句,出口是其最后一个语句,控制流一旦从入口进入基本块,不会停止或分支,只要基本块入口指令被执行,基本块内所有指令都会按照顺序执行一次。

通过 Pin 动态插桩工具在每个基本块入口处插入计数器,统计基本块的执行次数来调整 P_{nop} 。对于不同输入或测试用例,程序每次运行时基本块的覆盖率不尽相同,为了减小计算误差,尽可能提高基本块的覆盖率,本文用多个测试用例保证基本块的覆盖率达到 60%,然后加权每个测试用例中基本块的执行次数,作为程序最终的基本块执行次数结果。

NOP 指令插入流程如图 2 所示。

1) 用 Egalito 解析 ELF 文件成其自定义的中间语言 EIR,遍历出程序的函数、基本块和指令并保留这些要素的地址和相互关系。

2) 用 Pin 插桩工具统计基本块的执行次数,这一步骤是对原 ELF 文件进行处理。

3) 利用初始的 P_{nop} 和 2)得到的统计信息计算当前基本块的插入概率,依次处理 1)分析出的基本块,直至完成。

4) 将插入的 NOP 指令重新与源程序指令通过“link”相互关联,重新分配在地址空间的位置,生成新的 EIR,再由重写器后端生成新的 ELF 文件。

由于程序中基本块的执行次数并不是简单的线性分布,对于某些嵌套在多个循环体内的基本块,执行次数能达到上千万次,而多数基本块仅执行一次。因此,引入对数来缩小这种差距,再依据初始 P_{nop} 随机化对每个基本块的插入概率。设程序中共有 n 个基本块,记作 $B = \langle B_1, B_2, \dots, B_n \rangle$, 其中第 i 个基本块 B_i 执行次

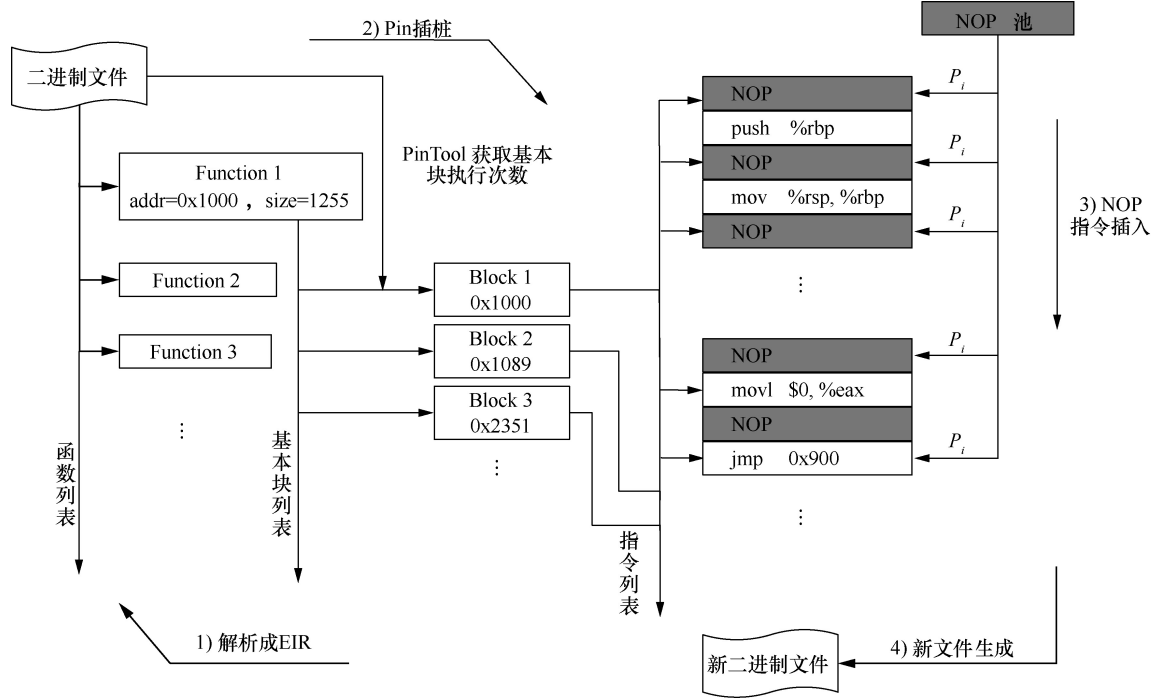


图 2 NOP 指令插入流程
Figure2 The process of NOP instruction insertion

数为 T_i ，则基本块执行的平均次数 T 为

$$T = \frac{1}{n} \sum_{i=1}^n T_i \quad (1)$$

设第 i 个基本块的插入概率 P_i 为

$$P_i = \frac{\text{lb}^T}{\text{lb}^{1+T_i} + 1} \cdot P_{\text{nop}}, P_{\text{nop}} \in [0, 1] \quad (2)$$

插入概率 P_i 是关于基本块执行次数 T_i 的减函数，基本块执行次数越多，该基本块插入的 NOP 指令数量越少，性能损耗越低。

3 随机化熵

文献[27]提出用熵 (entropy) 描述多样化的效果，熵值越高，其对代码重用攻击的防御效果越好，可表示为

$$H(X) = -\sum_{n=1}^N p_n \text{lb}^{p_n} \quad (3)$$

其中， N 表示多样化后可生成不同变体的数量， p_n 表示每种变体生成的概率。如果一个多样化技术生成每个变体的概率都是 $1/N$ ，那么该多样化技术的随机化熵为 lb^N 。本节通过对比“热”代码优化前后的随机化熵来评估优化的效果。优化

前，设程序中共有 m 条指令，每条指令前插入 NOP 指令的概率相同，然而指令偏移量大小的概率不同，指令偏移量为 3 byte 的概率为 $1/2P_{\text{nop}}$ ，偏移量为 1、2、4、5 和 6 byte 的概率为 $1/10P_{\text{nop}}$ ，偏移量为 0 的概率为 $1-P_{\text{nop}}$ ，由于每条指令是否选择插入 NOP 指令相互独立，则此时的随机化熵为

$$H_f = -m \cdot \left[\frac{1}{2} P_{\text{nop}} \cdot \text{lb}^{\frac{1}{20} P_{\text{nop}}^2} + (1 - P_{\text{nop}}) \text{lb}^{(1 - P_{\text{nop}})} \right] \quad (4)$$

由于优化后每个基本块的插入概率不尽相同，设 k_i 表示第 i 个基本块中指令的数量，则优化后的随机化熵为

$$H_b = -\sum_{i=1}^n k_i \left[\frac{1}{2} P_i \cdot \text{lb}^{\frac{1}{20} P_i^2} + (1 - P_i) \text{lb}^{(1 - P_i)} \right] \quad (5)$$

其中， P_i 由式(2)计算可得。

为了更加清楚地比较优化前后熵的变化，本节选用 SPEC CPU2006 基准测试集中的 403.gcc 进行实验，比较优化前后随机化熵的变化。“热”代码优化对随机化熵的影响如图 3 所示，程序 403.gcc 反汇编后约有 50 万条指令，可以产生足够高的熵，攻击者通过暴力破解几乎不可能，并

且优化后熵值大约提高了6倍,熵值的提高主要是由于每个基本块NOP指令插入概率不同。

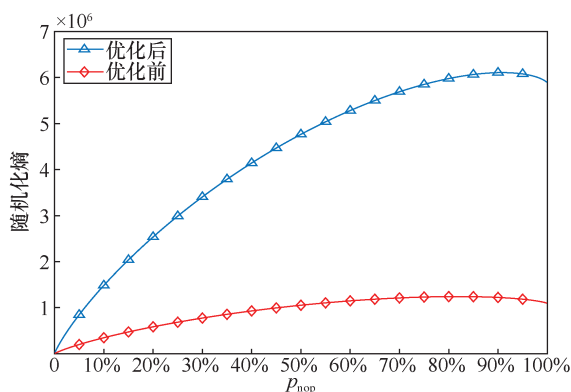


图3 “热”代码优化对随机化熵的影响
Figure 3 Impact of hot code optimization on randomization entropy

4 实验评估

为了验证对二进制文件插入空指令后程序的整体性能和实际效果,本文借鉴了软件多样化技术的主流评估方法,从转换前后程序的性能、安全性以及大小变化等方面进行了实例测试。实验使用SPEC CPU2006基准测试集中14个C/C++程序进行多样化转换,源程序生成用的是O2级别优化,实验环境如下:操作系统为64位Ubuntu 18.04,处理器为Inter Core i7-8700,内核版本为Linux 5.4.0-42-generic,内存为8 GB。

4.1 性能测试

为了评估本文方法对二进制文件性能的影响,取 $P_{nop}=10\%$ 、 $P_{nop}=30\%$ 、 $P_{nop}=50\%$ 和 $P_{nop}=100\%$ 为实验组,评估不同概率下的性能开销。NOP不同插入概率对软件性能的影响如图4所示。

即使在相同概率下,NOP指令的插入数量和位置不尽相同,且这些NOP指令对性能的影响也不完全相同,如交换指令{xchg %ax, %ax}比{mov %rbp, %rbp}会带来更大的性能开销,因为xchg指令在执行时会声明LOCK信号,然后锁定总线,独占共享内存,阻碍其他的CPU使用总线。为了减小误差,实验将每个测试程序运行3次,选取平均值作为最终结果。可以看到,随着 P_{nop} 的增加,软件整体的性能开销增加,当 $P_{nop}=10\%$ 和 $P_{nop}=30\%$ 时,平均性能降低约为3.7%和6.8%,

而当 $P_{nop}=50\%$ 和 $P_{nop}=100\%$ 时,平均性能降低约为11.5%和19.2%。

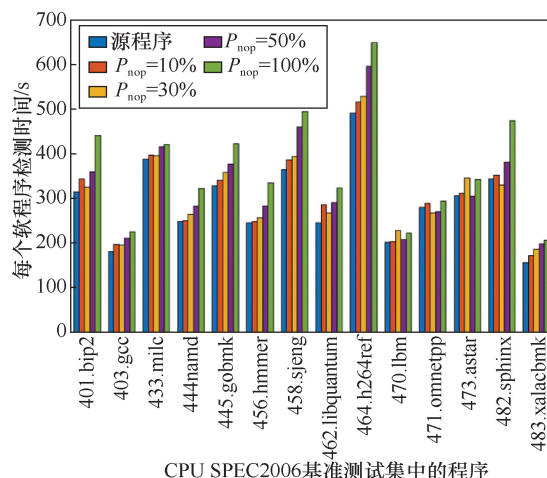


图4 NOP不同插入概率对软件性能的影响
Figure 4 Impact of different insertion probabilities of NOP on software performance

4.2 安全评估

安全性度量主要指从NOP指令插入后对代码重用攻击的防御能力,用gadget存活率这一指标来衡量。

4.2.1 gadget存活率

gadget存活率指的是转换前后文件中相同的gadget数占总数的比率,存活率越低表明相同的gadget越少,攻击者想用相同攻击链在其他软件复现攻击的难度越大。本文先用ROPgadget^[28]工具获得二进制文件中gadget,再寻找两个二进制文件中相同的gadget,最后计算存活率。由于重写器会把.text代码段的地址移到0x40000000处,为了计算存活率,本文用重写器重写源程序生成对照组,对照组不做任何转换,仅将源程序代码段和转换后程序代码段移到相同位置。另外,由于gadget中的空指令不会改变其语义,要将扫描出的gadget中的NOP删除后再做比较。实验结果如表3所示。

分析表2数据,通过横向比较,对同一软件进行转换,随着NOP指令插入概率的增加,插入数量增多,程序存活率总体趋势降低;通过纵向比较,同一插入概率下,不同程序的gadget存活率差异较大,如 $P_{nop}=10\%$ 和 $P_{nop}=30\%$ 时,470.lbm gadget存活率分别为10.74%和6.62%,

而 403.gcc gadget 存活率仅为 0.11% 和 0.09%。简单分析可知, 程序越大, 程序中的指令越多, 程序中插入的 NOP 指令数量也越多, 导致地址空间中靠后的 gadget 偏移量越大, 存活的 gadget 越少, 其存活率越低。

当 gadget 存活率足够低时, 攻击者必须在变体中重新搜寻 gadget 的位置才能发起攻击。同时, 即使在 P_{nop} 值相同的情况下, 插入 NOP 指令的数量和字节长度不全部相同, 因此生成的每个变体中, 原指令的偏移量都是随机的, 内存布局不可能全部相同。所以, 攻击者难以用同一攻击负载进行大规模的软件攻击, 增加了攻击成本, 达到了软件多样化的防御效果。

表2 NOP 不同插入概率下 gadget 的存活率
Table 2 Survival rate of gadgets under different insertion probabilities of NOP

软件名称	Gadger 数量	gadget 的存活率				
		10%	30%	50%	75%	100%
401.bzip2	3 111	2.16%	1.39%	1.41%	1.34%	1.49%
403.gcc	142 130	0.11%	0.09%	0.08%	0.09%	0.08%
433.milc	4 557	1.79%	1.84%	1.87%	1.58%	1.53%
444.namd	7 331	0.95%	0.88%	0.81%	0.89%	0.74%
445.gobmk	28 457	0.37%	0.33%	0.37%	0.35%	0.32%
456.hmmmer	12 749	1.00%	0.99%	0.88%	0.88%	0.90%
458.sjeng	6 397	1.19%	1.20%	1.00%	1.21%	1.05%
462.libquantum	2 097	2.59%	2.58%	2.09%	2.65%	2.27%
464.h264ref	19 242	0.45%	0.45%	0.43%	0.39%	0.40%
470.lbm	502	10.74%	6.62%	7.91%	6.78%	6.93%
471.omnetpp	22 182	0.82%	0.68%	0.69%	0.64%	0.62%
473.astar	1 838	2.45%	2.99%	3.02%	2.46%	2.56%
482.sphinx3	6 844	1.50%	1.69%	1.72%	1.44%	1.51%
483.xalancbmk	155 397	0.29%	0.12%	0.12%	0.12%	0.11%

4.2.2 最优 P_{nop}

为了求出实际生产中符合要求的 P_{nop} 值, 本文以 SPEC CPU2006 基准测试集中代表性文件 433.milc、444.namd 和 482.sphinx3 为例, P_{nop} 从 1% 开始, 增量为 1% 进行 NOP 指令插入, 每个 P_{nop} 值生成 3 个变体, 求其 gadget 的平均存活率和文件膨胀率。NOP 不同插入概率对 gadget 存活率的影响如图 5 所示, 当 P_{nop} 值较小时, gadget 存活

率波动比较大, 因为插入的 NOP 指令少, 不足以对地址靠前的 gadget 全部产生位移, 幸存的 gadget 数量较多。随着 P_{nop} 值增大, gadget 存活率保持稳定。当 $P_{nop} \approx 15\%$ 时, gadget 平均存活率为 1.49% 且趋近于 0, 同时对 $P_{nop} = 15\%$ 的 3 个变体进行性能测试, 平均结果表明, 该概率下运行开销增加约为 4.1%, 是合理的性能开销。

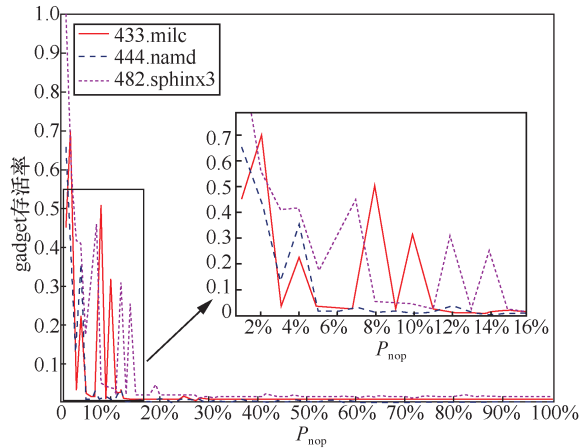


图5 NOP 不同插入概率对 gadgets 存活率的影响
Figure 5 Impact of different insertion probabilities of NOP on the survival rate of gadgets

4.3 文件膨胀率

插入的 NOP 指令会导致文件膨胀, 为具体评估不同插入概率对文件大小的影响, 选用软件 433.milc 在不同 P_{nop} 值下生成的变体, 同样取平均值作为最终的文件大小。NOP 不同插入概率对文件大小的影响如图 6 所示, 平滑曲线是根据平均大小拟合而成, 当 $P_{nop} = 15\%$ 时, 文件膨胀率约为 7.7%。

5 讨论

较现有的软件多样化实现方法, 指令偏移随机化能够显著降低 gadget 的存活率, 增加代码重用攻击难度。随着 NOP 指令插入概率的增加, gadget 的存活率收敛到 1% 左右, 而函数排列随机化、指令调度随机化和指令替换^[7]中 gadget 的存活率分别为 26%、9% 和 5%。同时, 指令偏移随机化具有较低的运行开销和较小的空间开销, 文献[29]将更新后的程序进行多种多样化方法转换, 以此阻止攻击者比较更新前后程序的差异,

增加了 5%~30% 运行开销和 15%~20% 文件大小, 文献[30]通过重写器将程序重组为一个新的自随机化二进制代码, 该二进制代码在每次执行时对其自身的基本块进行重新排序, 平均性能增加了 4.6%, 然而由于重写时对代码段进行了大量复制, 文件大小平均增加了 73%。在下一步工作中, 将结合代码覆盖率来更准确地获得软件的基本块热度, 以此优化该方法的运行开销和文件膨胀率。

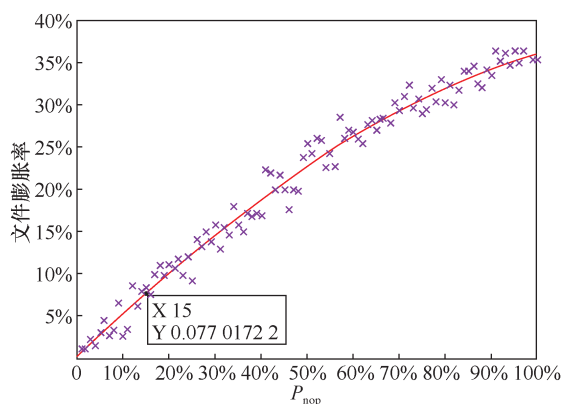


图6 NOP不同插入概率对文件大小的影响
Figure 6 Impact of NOP different insertion probabilities on file size

另外, 本文方法存在一定的局限性, 即二进制重写器 Egalito 不支持对非 PIE 和含有 C++ 异常的程序进行重写。目前 Linux 中的二进制文件基本默认为 PIE^[21], 难以对含有 C++ 异常的程序进行正确的重写, 是大多数重写器面临的共同问题, 在下一步工作中将对含有 C++ 异常的程序重写进行系统研究。

6 结束语

面向二进制程序的软件多样化方法, 由于不依赖于程序的源代码和内存信息, 现有的解决方案大多采用粗粒度的多样化方法, 在函数级别进行转换, 转换后熵值低且 gadget 存活率较高, 容易被暴力破解, 对代码重用攻击防御能力不足。

本文提出了一种在指令层细粒度的软件多样化方法, 指令偏移随机化, 该方法通过二进制重写解析二进制文件, 在每条指令前以一定概率插入不同的 NOP 指令。为了降低性能开销, 通过动态二进制插桩获得基本块的执行次数, 随机化 NOP 指令在每个基本块中的插入概率, 同时增加了该方法的随机化熵, 可生成足够多的内存布局

不同的变体以防御代码重用攻击。对该方法的实验表明: 当插入概率为 15% 时效果最好, 生成的程序变体中 gadgets 的平均存活率小于 1.49%, 约增加了 4.1% 的运行开销和 7.7% 的文件膨胀率, 具有良好的实用性。

参考文献:

- [1] FRANZ M. E unibus pluram: massive-scale software diversity as a defense mechanism[C]//Proceedings of the 2010 New Security Paradigms Workshop. 2010: 7-16.
- [2] LARSEN P, HOMESCU A, Brunthaler S, et al. SoK: automated software diversity[C]//Proceedings of 2014 IEEE Symposium on Security and Privacy. 2014: 276-291.
- [3] COHEN F B. Operating system protection through program evolution[J]. Comput Secur, 1993, 12(6): 565-584.
- [4] MARCO-GISBERT H, RIPOLL I. On the effectiveness of full-ASLR on 64-bit Linux[C]//Proceedings of the In-Depth Security Conference. 2014.
- [5] JACKSON T, SALAMAT B, HOMESCU A, et al. Compiler-generated software diversity[M]//Moving Target Defense. New York: Springer. 2011: 77-98.
- [6] HISER J, NGUYEN-TUONG A, CO M, et al. ILR: where'd my gadgets go[C]//Proceedings of 2012 IEEE Symposium on Security and Privacy. 2012: 571-585.
- [7] COFFMAN J, KELLY D M, WELLONS C C, et al. ROP gadget prevalence and survival under compiler-based binary diversification schemes[C]//Proceedings of the 2016 ACM Workshop on Software PROtection. 2016: 15-26.
- [8] CONTI M. Selfrando: securing the tor browser against de-anonymization exploits[C]//Proceedings of Priv Enhancing Technol. 2016: 454-469.
- [9] JUNOD P, RINALDINI J, WEHRLI J, et al. Obfuscator-LLVM--software protection for the masses[C]//Proceedings of 2015 IEEE/ACM 1st International Workshop on Software Protection. 2015: 3-9.
- [10] BANESCU S, COLLBERG C, PRETSCHNER A. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning[C]//Proceedings of 26th USENIX Security Symposium (USENIX Security 17). 2017: 661-678.
- [11] WENZL M, MERZDOVNIK G, ULLRICH J, et al. From hack to elaborate technique—a survey on binary rewriting[J]. ACM Computing Surveys (CSUR), 2019, 52(3): 1-37.
- [12] HOMESCU A, NEISIUS S, LARSEN P, et al. Profile-guided automated software diversity[C]//Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2013: 1-11.
- [13] MURPHY M, LARSEN P, BRUNTHALER S, et al. Software profiling options and their effects on security based diversification[C]//Proceedings of the First ACM Workshop on Moving Target Defense. 2014: 87-96.
- [14] HOMESCU A, JACKSON T, CRANE S, et al. Large-scale auto-

- mated software diversity—program evolution redux[J]. IEEE Transactions on Dependable and Secure Computing, 2015, 14(2): 158-171.
- [15] KRAHMER S. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique[R]. 2005.
- [16] 彭国军, 梁玉, 张焕国, 等. 软件二进制代码重用技术综述[J]. 软件学报, 2017, 28(8): 2026-2045.
PENG G J, LIANG Y, ZHANG H G, et al. Survey on software binary code reuse technologies[J]. Journal of Software, 2017, 28(8): 2026-2045.
- [17] DAVI L, SADEGHI A R. Building secure defenses against code-reuse attacks[M]. Berlin: Springer International Publishing, 2015.
- [18] 柳童, 史岗, 孟丹. 代码重用攻击与防御机制综述[J]. 信息安全学报, 2016, 1(2).
LIU T, SHI G, MENG D. A survey of code reuse attack and defense mechanisms[J]. Journal of Cyber Security, 2016, 1(2).
- [19] AHMED S, XIAO Y, SNOW K Z, et al. Methodologies for quantifying (Re-) randomization security and timing under JIT-ROP[C] //Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020: 1803-1820.
- [20] 侯宇. 基于动态随机化和只可执行内存的 JIT-ROP 防御研究[D]. 南京: 南京大学, 2016.
HONG Y. Defence against JIT-ROP based on dynamic randomization and executable only memory[D]. Nanjing: Nanjing University, 2016.
- [21] WILLIAMS-KING D, KOBAYASHI H, WILLIAMS-KING K, et al. Egalito: layout-agnostic binary recompilation[C]//Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 133-147.
- [22] LUK C K, COHN R, MUTH R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. ACM Sigplan Notices, 2005, 40(6): 190-200.
- [23] CHATTERJEE N, BOSE S, DAS P P. Dynamic weaving of aspects in C/C++ using PIN[C]//Proceedings of the International Conference on High Performance Compilation, Computing and Communications. 2017: 55-59.
- [24] 陈小全, 薛锐. 程序漏洞: 原因, 利用与缓解——以 C 和 C++ 语言为例[J]. 信息安全学报, 2017, 2(4).
CHEN X Q, XUE R. Cause, exploitation and mitigation of program vulnerability—C and C++ language as an example[J]. Journal of Cyber Security, 2017, 2(4).
- [25] MARCO-GISBERT H, RIPOLL I. Address space layout randomization next generation[J]. Applied Sciences, 2019, 9(14): 2928.
- [26] GUIDE P. Intel® 64 and ia-32 architectures software developer's manual[J]. Volume 3B: System programming Guide, Part, 2011, 2(11).
- [27] PRIYADARSHAN S, NGUYEN H, SEKAR R. Practical fine-grained binary code randomization[C]//Proceedings of Annual Computer Security Applications Conference. 2020: 401-414.
- [28] SALWAN J, WIRTH A. ROPGadget[EB].
- [29] COPPENS B, DE SUTTER B, DE BOSSCHERE K. Protecting your software updates[J]. IEEE Security & Privacy, 2012, 11(2): 47-54.
- [30] WARTELL R, MOHAN V, HAMLEN K W, et al. Binary stirring:

self-randomizing instruction addresses of legacy X86 binary code[C]//Proceedings of the 2012 ACM Conference on Computer and Communications Security. 2012: 157-168.

[作者简介]



何本伟 (1998-), 男, 安徽滁州人, 信息工程大学硕士生, 主要研究方向为网络空间安全和软件多样化。



郭云飞 (1963-), 男, 河南郑州人, 信息工程大学教授、博士生导师, 主要研究方向为网络空间安全、云安全和电信网安全。



王亚文 (1990-), 男, 河南郑州人, 信息工程大学助理研究员, 主要研究方向为拟态防御和云计算。



王庆丰 (1995-), 男, 河南周口人, 信息工程大学助理研究员, 主要研究方向为拟态防御和软件多样化。



扈红超 (1982-), 男, 河南商丘人, 信息工程大学研究员, 主要研究方向为云计算和网络安全。