

分支混淆中的条件异常代码构造研究

耿普, 祝跃飞

(信息工程大学, 河南 郑州 450001)

摘要: 当前分支混淆技术通过构造条件异常代码和异常处理替代条件跳转指令, 隐藏分支选择指令的地址, 提高约束条件获取的难度, 从而对抗符号执行。当前方法构造的条件异常代码中, 关键数据具有二值性问题, 有利于分支混淆的检测、发现和约束条件获取, 降低了混淆的隐蔽性和与符号执行的对抗性; 基于该缺点, 提出一种使关键数据具有多样性特征的条件异常代码构造方法, 增加了混淆的隐蔽性和约束条件的获取难度, 提高分支混淆对抗符号执行的强度; 通过结构化异常处理实现了基于新型条件异常代码构造的分支混淆原型系统, 并对混淆进行了测试和分析。

关键词: 条件异常代码构造; 分支混淆; 符号执行; 结构化异常处理

中图分类号: TP311

文献标识码: A

doi: 10.11959/j.issn.2096-109x.2020061

Research on construction of conditional exception code used in branch obfuscation

GENG Pu, ZHU Yuefei

Information Engineering University, Zhengzhou 450001, China

Abstract: Using conditional exception code construction and exception handler to replace conditional jump code, the branch obfuscation get the right branch selection, but the address of branch point was concealed, so this obfuscation method can defeat symbolic execution by impeding the constraint condition collecting. The normal method of conditional exception code construction has a fault that the key data in conditional exception code has two-value problem, this fault make down the ability of branch obfuscation in impeding symbolic execution. Based on this shortcoming, a novel method which can make the key data in conditional exception code diversity was proposed. This method can improve the difficulty of constraint condition fetching, so the ability to defeat symbolic execution of branch obfuscation was enhanced. At last, a prototype obfuscation system based on structural exception handler was implemented to test the new method of conditional exception code construction.

Key words: conditional exception code construction, branch obfuscation, symbolic execution, structural exception handler

收稿日期: 2019-12-02; 修回日期: 2020-02-04

通信作者: 耿普, 23015636@qq.com

基金项目: 国家重点研发计划 (2016YFB0801601, 2016YFB0801505)

Foundation Item: The National Key R&D Program of China(2016YFB0801601, 2016YFB0801505)

论文引用格式: 耿普, 祝跃飞. 分支混淆中的条件异常代码构造研究[J]. 网络与信息安全学报, 2020, 6(6): 25-34.

GENG P, ZHU Y F. Research on construction of conditional exception code used in branch obfuscation[J]. Chinese Journal of Network and Information Security, 2020, 6(6): 25-34.

1 引言

商业软件在发行后容易受到 MATE (man at the end) 攻击, 由于 MATE 攻击者对运行环境和软件都具有完全控制权, 可以采用任意程序逆向分析技术和工具对软件实施分析、调试、修改, 使软件的知识产权、私密数据和算法遭受危害, 给软件市场的健康发展造成损失; 更有甚者, 通过逆向分析发现软件漏洞, 对运行软件的信息系统和网络基础设施造成重大的安全威胁。基于 MATE 攻击和逆向分析技术带来的危害, 软件安全研究者提出了一种新的代码保护技术——代码混淆^[1-4]。该技术通过对程序进行变换, 增加程序逻辑和算法的理解难度, 提高攻击者在分析软件过程中的投入和消耗, 降低攻击者的攻击收益, 从而迫使攻击者放弃攻击, 达成软件保护的目的。在后续研究中, 代码混淆与软件加密^[5-9]等其他软件保护技术快速融合, 形成了一大批具有较好效果的软件保护技术。但是, 随着计算和存储能力的爆发式增长, 曾受限于大量计算和存储资源消耗的符号执行技术^[10-11]获得了新生, 通过与污点分析、动态插桩等技术配合, 实现了一种新的程序自动分析技术, 极大提高了逆向工程的效率, 降低了程序分析的消耗, 因此, 符号执行在逆向工程领域得到了快速发展。针对符号执行在逆向工程中的强大威力, 学者进行了抗符号执行^[12-16]的代码混淆技术研究, 并提出了一种新的代码混淆技术——分支混淆^[13-14, 17-18], 通过保护程序的分支信息, 对抗符号执行对约束条件获取和求解的难度, 降低符号执行分析的有效性。

基于异常处理实现路径分支混淆^[13, 17]是一种实现简单、对抗符号执行效果良好的混淆技术, 其本质是通过构造条件异常代码替代条件跳转指令, 从而隐藏分支选择指令地址, 增加符号执行实现约束条件获取的难度, 实现混淆与符号执行的对抗。因此, 在该类分支混淆中, 条件异常代码的构造方法对混淆的隐蔽性和混淆与符号执行的对抗性具有关键作用。然而, 当前研究中, 尚无针对条件异常代码构造方法的专门研究。当前的构造方法中, 为实现混淆的隐蔽性, 通常使用

常见的除法指令和内存访问指令构造条件异常代码。但对除数和内存访问地址等关键数据的构造方法较为简单, 使关键数据具有二值性: 一方面二值性特征影响了混淆的隐蔽性; 另一方面, 通过对关键数据进行内存污点分析, 同样可以列出约束条件, 实现符号执行, 降低了分支混淆对抗符号执行的强度。基于此, 本文提出了一种新的异常代码构造方法, 通过对关键数据的构造进行改进, 使关键数据具有多样性特征, 解决了二值性问题, 提升了混淆的隐蔽性; 并对标志寄存器的使用进行了优化处理, 增加了约束条件获取的难度, 提高分支混淆对抗符号执行的强度。

本文的贡献如下。

1) 对当前分支混淆中的条件异常代码构造进行了分析, 提出了条件异常代码构造中的关键数据二值性问题, 并分析了该问题在分支混淆隐蔽性和混淆与符号执行对抗性上的缺点。

2) 针对条件异常代码中关键数据的二值性问题, 提出了一种新的条件异常代码构造方法, 解决了关键数据二值性问题, 提高了混淆的隐蔽性; 改进后的构造方法, 使关键数据具有更多的选择, 具有多样性特征, 提升了符号执行获取约束条件的难度。

3) 实现了一种基于结构化异常处理和新条件异常代码构造的分支混淆方法的原型系统, 并对分支混淆进行了测试和分析。

2 基于异常处理的分支混淆研究现状分析

2007 年, Popov^[2]等提出了基于异常处理的二进制代码混淆技术, 通过构造陷门代码替代无条件跳转指令, 使程序执行到陷门代码时注册的信号处理函数得到程序控制权, 通过陷门产生时的程序状态, 区分陷门是否来源于替代条件跳转指令的代码, 并在信号处理函数中对程序执行流程进行修改, 使混淆后代码与混淆前代码具有相同的程序语义; 并通过在陷门代码后插入垃圾指令, 对抗反汇编工具对二进制代码进行反汇编, 提高二进制代码对抗反汇编技术的能力, 对抗程序分析, 实现代码保护。在 Popov 的方法中, 提出了使用非法指令、浮点数异常和内存访问异常构造陷门代码, 从而产生 SIGILL、SIGFPE 和

SIGSEGV 信号。但是,一方面,文献[2]中没有给出具体的条件异常代码的构造方法;另一方面,非法指令、浮点数异常指令在程序中不具有普遍性,导致条件异常代码隐蔽性不高。

2011年,在 Popov 等工作的基础上,贾春福等^[13]把 Popov 的方法从无条件跳转指令扩展到了条件跳转指令,通过使用条件异常代码替代条件跳转指令,当异常产生时,使用异常处理机制获取程序控制权,并在异常处理函数中通过控制权转移实现分支目标地址选择,使混淆后程序语义不变。在该混淆方法中,使用条件异常代码替代跳转指令是为了防止分支点地址和分支条件泄露,阻碍符号执行工具获取约束条件。因此,构造的条件异常代码需要具有较好的隐蔽性,难以同正常代码进行区分。文献[13]使用除零异常和内存访问异常构造条件异常代码,虽然构造条件异常代码使用的 mov 和 div 指令都具有较好的隐蔽性,但作为除法指令的除数和内存访问指令的地址数据具有二值性,从而为发现分支点地址带来了一定的便利,并且二值性容易被符号执行技术利用,用于降低获取约束条件的难度和对抗符号执行的强度。另外,div 指令产生的异常易被识别,从而有利于混淆还原。

2013年, Lin 等^[17]提出了基于异常处理和远程可信实体结合的二进制代码混淆技术,主要是在贾春福^[13]等工作的基础上增加了一个远程可信实体,用于保存分支的目标地址信息,增加逆向分析的难度,但并没有就条件异常代码构造问题进行分析 and 讨论。

虽然条件异常代码的构造在基于异常处理的分支混淆技术中具有重要作用,决定着分支混淆的隐蔽性和分支混淆对抗符号执行的强度。但是,在当前的研究中,还没有针对条件异常代码构造方法的详细分析和讨论,本文希望通过对该问题的研究和讨论,对条件异常代码的构造具有一定的指导和推动作用。

3 当前条件异常代码构造的问题

在基于程序异常的代码混淆方法中,条件异常代码是实现程序控制流变化的关键因素。条件异常代码是一种在特定条件下产生异常而其他条

件下不产生异常的一段特定代码,该代码对条件取值具有敏感性,能够根据条件取值不同,使程序执行不同的路径,因此条件异常代码需要具备较好的隐蔽性。在当前基于程序异常实现的代码混淆方法^[13,17]中,通常使用标志寄存器参与运算来构造条件异常代码,虽然使用了常见的 MOV 等赋值指令,使条件异常代码结构具有较好的隐蔽性,但在使用标志寄存器的过程中,对标志寄存器的处理和使用导致产生异常的数据具有二值性,因此不仅为混淆检测提供了便利的特征,而且数据的二值性便于符号执行技术获取约束条件,从而降低了混淆对抗符号执行的效能。

3.1 基于异常处理的代码混淆

基于程序异常的代码混淆是一种通过程序控制方式变换实现的代码混淆技术,使用条件异常代码替代条件跳转指令(也可以使用隐蔽的异常产生代码替代无条件跳转指令实现单分支混淆)实现程序分支信息保护,阻碍分支信息泄露、增加约束条件获取的难度,从而对抗以符号执行为核心的自动程序分析技术。

当前执行流程主要通过输入数据和条件跳转指令、无条件跳转指令实现路径分支选择,因此,符号执行工具一般通过跳转指令定位实现约束条件收集。基于程序异常的代码混淆正是针对跳转指令定位实现约束条件获取这一特征,对抗以符号执行为基础的程序分析。

基于程序异常的代码混淆方法如图1所示,通过构造条件异常代码,并替换原控制指令,实现分支控制方式的混淆,图1中的条件异常代码是在分支输入使条件取值为真时产生异常,从而使代码块B得到执行,而在条件异常代码不产生异常时,即分支输入使条件取值为假时,代码块C得到执行,从而保证代码混淆不会改变程序语义。

条件异常代码如图2所示,通过 sbb 指令使 EFLAGS 寄存器的标志位 CF 参与计算,使 ecx 的取值根据 CF 位的不同而不同,当 CF 为 1 时,ecx 取值为 0,从而产生异常;CF 为 0 时,ecx 取值为 0x80000000,由于不产生异常而使代码顺序执行。

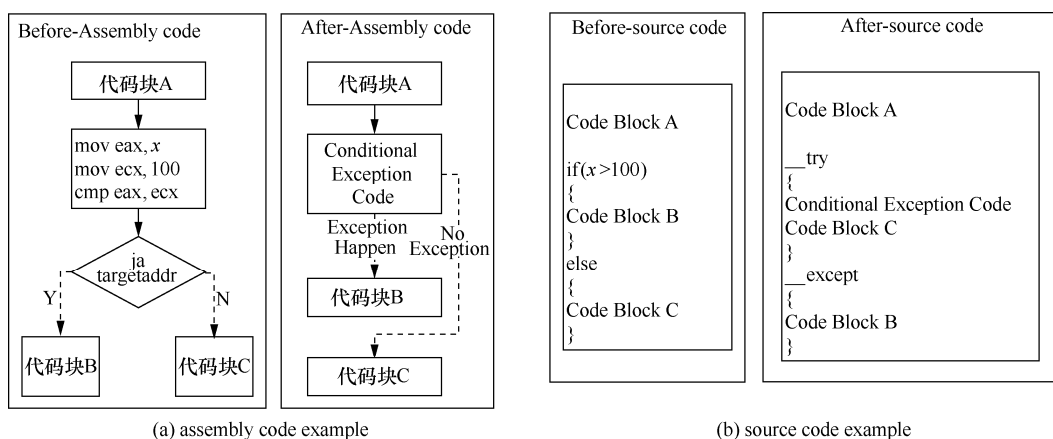


图 1 基于程序异常的代码混淆方法
Figure 1 Example of obfuscation method

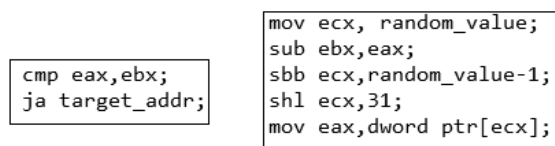


图 2 条件异常代码示意
Figure 2 Conditional exception code

3.2 条件异常代码构造存在的问题

条件异常代码构造是为了在特定条件下产生异常,并且保证构造出的代码具有较好的隐蔽性,使分支混淆在保证程序语义的同时难以被检测。由于分支混淆的目标是实现与程序分析技术——符号执行的对抗,因此条件异常代码的构造还需要考虑其对符号执行对抗的影响。一方面,具有二值性的内存地址访问指令容易被符号执行技术用于约束条件获取;另一方面,当前基于程序异常实现的分支混淆研究中,尚无条件异常代码与对抗符号执行功能影响分析的研究,也没有专门针对条件异常代码构造方面的研究。

当前的条件异常代码构造相对简单,主要为实现混淆的隐蔽性和程序语义的保持,而在对抗符号执行影响方面尚无分析研究。在实现混淆隐蔽性方面,条件异常代码构造都采用内存访问指令和数学计算指令实现,如 `mov`、`add`、`sub`、`div`、`mul` 等常见指令,使条件异常代码具有一般性,能够更好地与原程序代码融合,使分支混淆没有明显特征,实现混淆的隐蔽性。在对抗符号执行上,一方面是跳转指令被条件异常代码替代,另

一方面是条件异常代码具有较好隐蔽性,从而增加了约束条件收集的困难,提高了符号执行分析的难度。

虽然内存访问构造的代码在样式上没有明显特征,隐蔽性较好,提升了符号执行收集约束条件的难度,但在内存访问地址构造上还存在缺陷,为实现程序的条件异常,内存访问地址在构造上具有如下缺点。

(1) 内存地址构造具有二值性,导致地址值不会随分支输入的改变而连续变化。

为保持程序语义,需要条件异常代码访问的内存地址值中,一个值使程序产生异常而改变执行流程;另一个值程序不产生异常,保持程序指令顺序执行。二值性产生的主要原因为标志寄存器的使用,为使程序在特定条件下产生异常,使用标志寄存器的特定比特位构造需要访问的内存地址,即使用条件异常代码替换跳转指令的本质仅仅是换了另外一种方式使用标志寄存器,但对标志寄存器的使用过程中仅用到了控制跳转的比特位,导致内存地址具有二值性。

由于内存访问地址这一关键数据具有二值性,因此该值难以随着分支输入的改变而产生连续性变化,降低了逆向分析的难度。

(2) 内存地址二值性为分支混淆的检测和识别引入了明显特征,且给符号执行收集约束条件和实现分支翻转提供了便利条件。

首先,在混淆检测和识别方面,使用污点分析技术可以识别疑似条件异常代码的位置,即内

存地址二值性降低了分支混淆的隐蔽性。以下3个特征为基于内存访问构造的条件异常代码的显著特征:

- ① 当前的标志寄存器受输入的污染;
- ② 当前内存访问指令使用的地址值受标志寄存器的污染;
- ③ 地址数据的取值具有二值性。

若当前指令访问的内存地址变量满足以上3个特征, 则当前代码为可疑的条件异常代码; 然后根据内存地址的取值对代码进一步分析, 可判断当前位置是否为条件异常代码。

其次, 内存地址二值性为符号执行收集约束条件提供了便利。若某一段内存访问代码被选定为疑似条件异常代码, 则对该代码访问的内存地址取值可以进行约束条件收集。设当前访问的内存地址 `addrA` 的约束为 `cond1=addrA`, 则反向路径的约束可以通过修改 `cond1=addrA` 为 `cond1≠addrA` 实现。即内存地址的二值性降低了符号执行收集约束条件的难度, 从而降低了混淆对抗符号执行的效能。

4 新的条件异常代码构造方法

为解决条件异常代码构造中, 内存访问地址具有二值性的问题, 本文通过引入数据多样化方法实现条件异常代码构造, 使内存访问地址的取值具有多样性, 从而难以区分正常地址访问指令和产生条件异常的地址访问指令, 使构造的条件异常代码具有更好的隐蔽性; 另外, 即使条件异常代码被识别后, 由于内存访问地址取值的多样性, 即代码产生异常和不产生异常的内存地址值取值具有多样性, 因此, 难以使用内存访问地址的取值列出约束条件, 即新构造的条件异常代码在对抗符号执行上具有更好的效果。

4.1 条件异常代码构造的限制

条件异常代码由于其特殊的用途, 因此在构造上具有一定的限制。首先是功能上, 能够替代跳转指令的功能, 保持程序语义不变; 其次是构造出的代码需要具有隐蔽性, 即与未混淆的代码之间不具有明显区分性。

在功能上, 为实现特定条件下程序产生内存

访问异常, 而其余条件下则保持程序顺序执行而不产生异常, 需要使用分支输入和 `EFLAGS` 标志寄存器参与条件异常代码的构造。在隐蔽性方面, 主要是构造异常使用的指令需要是常见指令和常见操作, 不应具有特殊性。因此, 采用内存访问操作和指令构造条件异常代码。为保证分支混淆的隐蔽性, 应使用常见指令和操作构造条件异常代码不能改变, 但分支输入和 `EFLAGS` 标志寄存器的使用方法可以进一步改进, 解决内存访问地址变量的二值性问题, 从而提升分支混淆的隐蔽性和分支混淆对抗符号执行的能力。

4.2 内存访问地址二值性解决方法

第3节分析了当前基于内存访问的条件异常代码构造的问题, 主要是内存地址的二值性问题导致分支混淆在隐蔽性和与符号执行分析技术对抗性上的下降。为解决内存地址二值性问题, 本文提出了一种新的内存访问地址构造方法。

内存地址的二值性来源于标志寄存器比特位的二值性, 即在条件异常代码的构造中, 仅使用了标志寄存器中会影响跳转指令进行分支选择的标志位, 导致生成的内存地址具有二值性, 使条件异常代码具有明显特征, 降低了混淆隐蔽性和与符号执行分析技术的对抗性。本文提出了一种使用标志寄存器所有比特位参与运算的构造方法。一方面提高了攻击者发现标志寄存器中影响分支选择标志位的难度; 另一方面, 构造的内存地址值具有更多选择, 提升了分支混淆后符号执行收集约束条件的难度, 并且通过去除二值性特征, 提高了分支混淆的隐蔽性; 还可以通过使用分支输入与随机内存处的数据做对比, 多次获取标志寄存器的值参与目标地址值的计算, 从而增加程序分析的复杂度。

内存地址构造方法的具体步骤如下。

1) 找到一块内存访问会产生异常的线性地址空间, 其起始地址为 `expAddrStart`, 内存块长度为 `expLen`, 如 32 位程序线性地址空间中的 0~64 kB 空间, 或者应用程序访问会产生异常的内存空间; 一块能够正常访问的线性地址空间, 如程序加载后占用的地址空间。当前方法以 0~64 kB 空间和程序加载的数据段地址空间为例, 并假设数据段加载的起始地址为 `dataBaseAddr`, 空间大

小为 dataLen。

2) 使用影响分支选择的标志位作为变量 index 的低位, 即 index 取值为 0 或者 1; 并取标志寄存器的值赋给变量 regValue; 最后计算内存访问地址 $\text{targetAddr} = \text{index} \cdot \text{database_Addr} + (1 - \text{index}) \cdot \text{expAddrStart} + (\text{flagRegValue} \cdot \text{rand}()) \% \text{Len}$, 其中 $\text{Len} = \text{datalen} < \text{expLen} ? \text{dataLen} : \text{expLen}$ 。即通过计算, 使访问的目标内存地址不是 0~64 kB 中的非法访问地址, 就是程序数据段的合法访问地址。

4.3 关键数据多样化构造的方法

在分支混淆构造的条件异常代码中, 内存访问地址是决定是否产生异常的关键数据, 而标志寄存器受程序代码影响而产生变化的比特位有限, 导致内存地址的选择个数有限。为扩大内存地址的选择范围, 本文提出了内存访问地址数据具有多样性选择的构造方法, 使用分支输入参与运算, 提供多个可选异常访问内存区域等手段, 提升符号执行获取约束条件的难度, 提高混淆与符号执行技术的对抗性。

多样化内存地址的构造方法具体步骤如下。

1) 选择多个可合法访问的内存区域和多个非法访问的内存区域, 使用数组 $\text{memSec}[2][n]$ 记录, 其中, $\text{memSec}[0]$ 表示合法访问区域集合, $\text{memSec}[1]$ 表示非法访问区域集合。

2) 使用影响分支选择的标志位作为变量 index 的低位, 即 index 取值为 0 或者 1; 计算 $\text{offset} = ((\text{flagRegValue} + \text{branchInput}) \cdot \text{rand}()) \% \text{Len}$;

计算 $\text{seq} = (\text{flagRegValue} + \text{branchInput}) \cdot \text{Rand}() \% n$, 其中, n 为内存访问区域的个数。

3) 取内存访问地址 $\text{targetAddr} = \text{memSec}[\text{index}][\text{seq}].\text{baseAddr} + \text{offset}$ 。

由上述步骤计算出来的内存访问地址, 受分支输入 (branchInput)、标志寄存器 (flagRegValue) 和随机数 (rand()) 的污染, 因此, 计算出的 targetAddr 取值的变化范围更大, 即保证了内存访问地址的多样性。图 3 是采用了具有多样性内存地址的条件异常代码实现分支混淆的一个例子, 其中, 标志寄存器的比特位既可以直接使用 EFLAGS 的值, 也可以使用数值计算结果替代。例如, 对 $\text{if}(x==100)$, 通过汇编指令 {mov eax,x;sub eax,100} 计算 eax, 使用 eax 的各比特位取或的结果来获取标志位, 效果与使用 EFLAGS 寄存器的 ZF 位一致。

5 实验和分析

本文实验是基于搭载 Windows 7 系统的主机实现, 主机使用的 CPU 为 Intel Core i5-750, 内存为 8 GB 的 DDR3 内存, 源代码的编译使用的是 visual studio 2015。

5.1 实现

本文使用 SEH 异常机制替代条件跳转指令实现分支混淆, 在条件异常代码的构造中, 使用了本文提出的新方法, 构造了具有多样性的内存访问地址, 避免了内存地址二值性带来的缺陷。分支混淆过程示例如图 4 所示。

<pre> 1 unsigned int x; 2 if (x<1000) 3 { 4 CodeBlockA; 5 } 6 CodeBlockB; </pre>	<pre> 1 unsigned int offset = 0; 2 int seqNum = 0; 3 struct memSection memSec[2][5]; 4 int secLen = 4096; 5 int Irandom = rand(); 6 int index = 0; 7 int baseAddr = 0; 8 unsigned targetAddr = 0; 9 __try 10 { 11 __asm 12 { 13 mov eax, x; 14 cmp eax, 1000; 15 pushf; 16 pop ax; 17 mov edx, eax; 18 and ax, 0x01; // cf 19 mov ebx, eax; 20 mov ecx, edx; 21 and ecx, 0x040; // zf 22 shr ecx, 6; 23 xor ecx, 0x01; 24 mul cx; 25 mov index, eax; 26 add edx, x; 27 mov ecx, edx; 28 and ecx, x; 29 mov seqNum, ecx; 30 mov eax, Irandom; 31 mul edx; 32 mov offset = eax; 33 } 34 targetAddr = memSec[index][seqNum].baseAddr + 35 offset % secLen; 36 unsigned int uTmp = *targetAddr; 37 } 38 __except(1) 39 { 40 CodeBlockA; 41 } 42 CodeBlockB; </pre>
---	--

图 3 使用新条件异常代码的分支混淆示例

Figure 3 Example of branch obfuscation using new conditional exception code

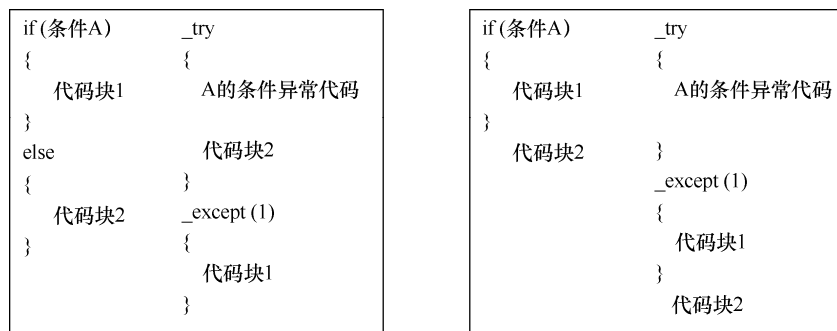


图 4 分支混淆的条件代码替换示例
Figure 4 Example of branch-replace-with-conditional exception code

通过对一个 IP 地址判断程序进行分支混淆 (IP 地址判断程序功能是判断一个输入 IP 地址是否属于网络地址 192.168.128.0/17)，混淆前后程序的控制流图对比如图 5 所示。由于采用 `_try- _except` 结构替代了 `if-else` 结构，因此编译后的二进制代码中，条件跳转代码被条件异常代码和异常控制机制替代，即使用隐式的分支选择替代了显示分支选择，导致混淆后程序的控制流图中没有了分支选择，而是成为多个并列的顺序执行代码流。

5.2 新旧条件异常代码属性对比

在 Lin 等^[17]和贾春福等^[13]的工作中，同样使用了条件异常代码构造实现分支选择，从而保证混淆后程序的语义不发生改变，但在他们的工作中，对内存访问异常指令中的关键数据——内存访问地址的构造，直接使用原分支条件执行后受影响的标志寄存器标志位，导致构造的内存访问地址具有二值性，从而容易被检测、识别，甚至应用标志寄存器特定标志位翻转实现符号执行分析。

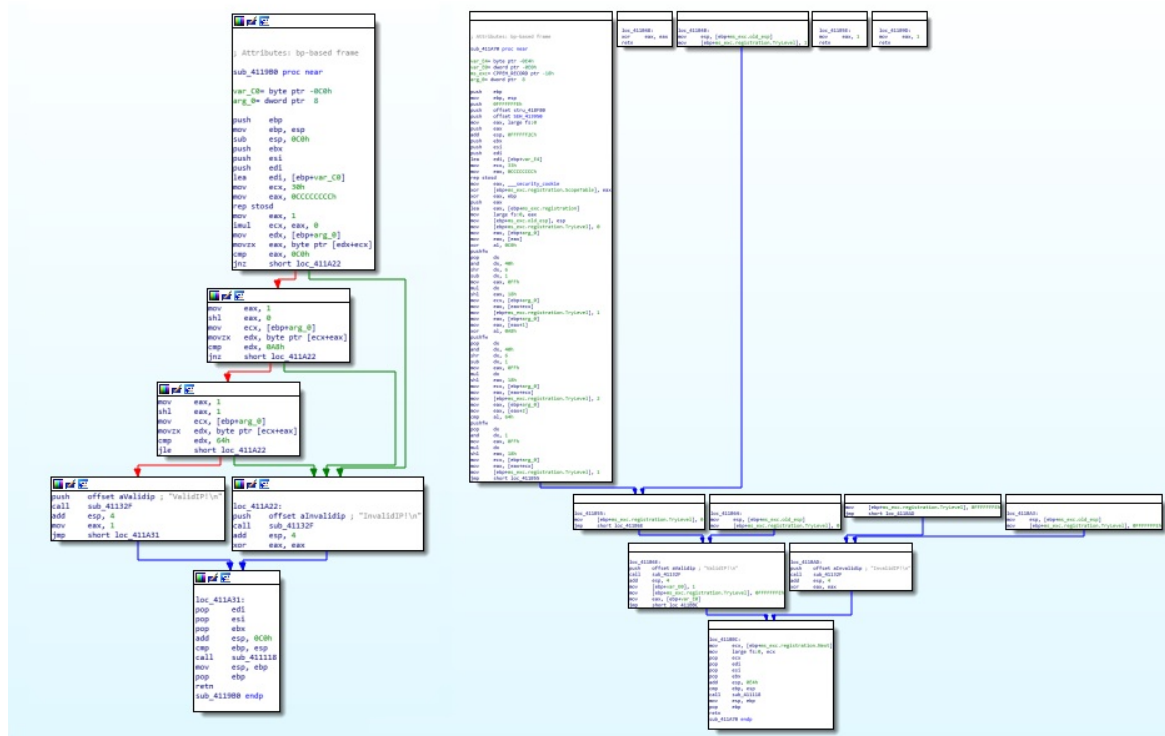


图 5 混淆前后控制流图对比
Figure 5 Comparison of program's CFG

而改进后的条件异常代码中,一是内存访问目标地址取值具有多元性;二是使用整个标志寄存器参与内存访问目标地址运算,而不是仅使用受原分支条件污染的标志位参与运算;三是引入了分支输入参与运算,使污染分析更为困难;四是内存访问的目标地址随分支输入的变化而每次产生变化,而在 Lin 和贾春福等的工作中,目标内存地址在分支输入整个取值变化范围内仅产生一次变化。在使用 `_try-_except` 结构替代 `if-else` 结构,使用 56~60 kB 之间的 4 kB 地址作为异常访问地址,使用程序加载地址作为正常访问地址的例子下,本文方法与文献[17]和文献[13]中的条件异常代码产生的关键数据对比如表 1 所示。即新的条件异常代码构造方法中,内存访问地址的变化更加随机,且地址取值因输入的不同而变化;相比旧构造方法中地址值仅在分支条件取值产生变化时而不同的状态,新构造方法能够产生更加随机的地址值,因此能够更好地对抗约束条件获取和代码混淆位置的定位,提升了代码混淆的性能。

5.3 隐蔽性和抗符号执行分析

在隐蔽性上,采用关键数据具有多样性的条件异常代码后,用作混淆的条件异常代码没有了关键数据二值性的特征,且用户混淆的异常处理使用 SEH 机制构造,再加上使用常见 `mov` 指令实现异常产生,使攻击者难以区分用作混淆的异常和正常产生的异常,因此,混淆的隐蔽性得以提高,分支混淆具有较好的隐蔽性。

在抗符号执行上,使用改进前的条件异常代码构造方法的分支混淆后程序,可以通过修改符号执行分析工具的方法,自动实现对被混淆分支的发现和对应路径遍历。本文通过修改卡内基梅隆大学开发的 BAP (binary analysis platform) ver0.9.6 对异常代码构造改进前后的分支混淆性能进行分析,对比条件异常代码构造方法改进的作用。

对 BAP 平台的修改主要是对二进制插装产生执行轨迹的模块和污染分析模块的代码进行修改,具体修改如下。

一是监控标志寄存器使用指令,记录此前最近一条影响标志寄存器的指令,并从此开始记录标志寄存器作为污染源的污染轨迹。

二是定位分支混淆位置和分支指令还原,通过增加对内存访问指令的判断,在每个内存访问指令处,首先判断访问地址对应内存是否被污染,如果被污染则继续判断是否被标志寄存器的某个或者某两个比特污染;如果是被寄存器污染,则根据标志寄存器参与污染的比特位来查表获取对应的条件跳转指令,并在程序输入作为污染源的污染轨迹中使用条件跳转指令替代内存访问指令。

修改后,即可对当前轨迹中替换内存访问指令的条件跳转指令进行反转,获取反转后对应的约束条件进行求解,得到当前轨迹中被混淆处相反路径的程序输入。

仍然以 IP 地址判断程序的混淆后程序为例进行实验测试,其中,IP 地址判断程序功能是判断一个输入 IP 地址是否属于网络地址 192.168.128.0/17,初始输入 IP 地址为 172.16.10.1。实验结果如表 2 所示,显然在改进前,修改后的 BAP 平台实现了符号求解,完成了对程序的分析,找到了满足网络地址 192.168.128.0/17 的解 192.168.128.1;而使用改进后的条件异常代码构造方法的分支混淆中,则无法找到相应的被混淆分支、列不出约束条件,从而无法求解。

改进前,由于对条件异常代码构造仅仅使用了被条件判断影响的标志寄存器比特位数据,因此通过修改 BAP 平台的代码,能够检测出使用内存访问产生的条件异常代码,并且可以自动实现条件跳转指令的恢复,从而顺利实现符号执行的约束获取和求解。但在条件异常代码构造方法改

表 1 新旧条件异常代码的关键数据对比

Table 1 The key data comparison between original and improved conditional code construction method

条件异常代码构造方法	地址个数	地址与分支输入的关系	参与计算地址值的 EFLAGS 寄存器位数
改进前的方法	2 个	分支输入参与地址计算;地址值随输入的变化而不同	1 或 2 位
本文的改进方法	4 096 个	分支输入不参与地址计算;仅当输入导致分支条件取值变化时,地址值改变	32 位

表2 条件异常代码构造改进前后的混淆性能对比
Table 2 The obfuscation effect comparison between original and improved conditional code construction method

混淆中的条件异常代码构造方法	被寄存器污染的内存访问地址数	轨迹中可反转的条件跳转指令数	按照可反转指令深度依次反转后的解
改进前的方法	3	3	192, 168, 128
本文改进后方法	0	0	无解

进后,内存访问地址的构造上引入了多样性特征,并且使用随机数、分支输入和标志寄存器的所有比特位参与了地址值的构造,即使发现了分支混淆的位置,也由于内存访问地址值的多样性,难以对访问地址的取值进行分支条件反转和约束条件获取,即增加了约束条件获取的难度,改进后的条件异常代码构造方法使分支混淆提升了对抗符号执行的强度。

5.4 地址随机化与条件异常代码构造

地址随机化不仅对本文提出的方法具有影响,对其他基于内存访问异常构造条件异常代码的方法具有同样的影响。地址随机化对本文提出方法的影响主要在于如何选取可以正常访问的地址和访问会产生异常的地址。由于正常访问地址可以通过获取程序进程块结构获取加载基地址和内存映射关系的方式动态获取。相应地,一方面,没有映射的地址是访问会产生异常的地址,另一方面,有一些固定地址是应用程序访问会产生异常的地址,如从0地址开始的前64 kB内存空间、内核使用的内存空间等。因此,地址随机化因素虽然一定程度上增加了条件异常代码构造的难度,但可以通过动态获取具有特定属性的内存访问空间来规避。

6 结束语

本文分析了当前条件异常代码构造在对抗混淆检测和符号执行上存在的问题。即关键数据二值性问题容易被利用于混淆的检测和识别,以及标志寄存器仅使用原分支条件影响的标志位,造成攻击者可以使用数据翻转替代分支翻转,从而实现符号执行分析的问题。并且,本文根据出现的问题提出了一种新的条件异常代码构造方法,使新的条件异常代码在内存访问地址值上具有多样性,且随着分支输入的变化而一直改变,解决

了原条件异常代码中出现的二值性问题和标志寄存器标志位使用带来的数据可翻转问题,提高了条件异常代码的隐蔽性,从而增强了分支混淆的隐蔽性;关键数据二值性问题和关键数据可翻转问题的解决,使利用新条件异常代码实现的分支混淆在对抗符号执行上具有更好的效果。

参考文献:

- [1] VITICCHIÉ A, REGANO L, TORCHIANO M, et al. Assessment of source code obfuscation techniques[C]//2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). 2016: 11-20.
- [2] POPOV I V, DEBRAY S K, ANDREWS G R. Binary obfuscation using signals[C]//USENIX Security Symposium. 2007: 275-290.
- [3] SCHRITTWIESER S, KATZENBEISSER S, KIESEBERG P, et al. Covert computation: hiding code in code for obfuscation purposes[C]//The 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. 2013: 529-534.
- [4] AVIDAN E, FEITELSON D G. From obfuscation to comprehension[C]//2015 IEEE 23rd International Conference on Program Comprehension. 2015: 178-181.
- [5] BALACHANDRAN V, KEONG N W, EMMANUEL S. Function level control flow obfuscation for software security[C]//2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS). 2014: 133-140.
- [6] BALACHANDRAN V, EMMANUEL S. Potent and stealthy control flow obfuscation by stack based self-modifying code[J]. IEEE Transactions on Information Forensics and Security, 2013, 8(4): 669-681.
- [7] BALACHANDRAN V, EMMANUEL S. Software protection with obfuscation and encryption[C]//International Conference on Information Security Practice and Experience. 2013: 309-320.
- [8] BRUNI R, GIACOBazzi R, GORI R. Code obfuscation against abstract model checking attacks[C]//International Conference on Verification, Model Checking, and Abstract Interpretation. 2018: 94-115.
- [9] CIMITILE A, MARTINELLI F, MERCALDO F, et al. Formal methods meet mobile code obfuscation identification of code reordering technique[C]//2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises

- (WETICE). 2017: 263-268.
- [10] BRUMLEY D, JAGER I, AVGERINOS T, et al. BAP: a binary analysis platform[C]//International Conference on Computer Aided Verification. 2011: 463-469.
- [11] KING J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394.
- [12] BANESCU S, COLLBERG C, GANESH V, et al. Code obfuscation against symbolic execution attacks[C]//The 32nd Annual Conference on Computer Security Applications. 2016: 189-200.
- [13] 贾春福, 王志, 刘昕, 等. 路径模糊: 一种有效抵抗符号执行的二进制混淆技术[J]. 计算机研究与发展, 2011, 48(11): 2111-2119.
- JIA C F, WANG Z, LIU X, et al. Branch obfuscation: an efficient binary code obfuscation impede symbolic execution[J]. Journal of Computer Research and Development, 2011, 48(11): 2111-2119.
- [14] 王志, 贾春福, 刘伟杰, 等. 一种抵抗符号执行的路径分支混淆技术[J]. 电子学报, 2015, 43(5): 870-878.
- WANG Z, JIA C F, LIU W J, et al. Branch obfuscation to combat symbolic execution[J]. Acta Electronica Sinica, 2015, 43(5): 870-878.
- [15] BANESCU S, COLLBERG C, GANESH V, et al. Code obfuscation against symbolic execution attacks[C]//The 32nd Annual Conference on Computer Security Applications. 2016: 189-200.
- [16] BANESCU S, COLLBERG C, PRETSCHNER A. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning[C]//The 26th USENIX Security Symposium. 2017.
- [17] LIN H, ZHANG X, YONG M, et al. Branch obfuscation using binary code side effects[C]//International Conference on Computer, Networks and Communication Engineering (ICCNC 2013). 2013.
- [18] 陈喆, 贾春福, 宗楠, 等. 随机森林在程序分支混淆中的应用[J]. 电子学报, 2018, 46(10): 2458-2466.
- CHEN Z, JIA C F, ZONG N, et al. Branch obfuscation using random Forest[J]. Acta Electronica Sinica, 2018, 46(10): 2458-2466.

[作者简介]



耿普 (1982-), 男, 云南宣威人, 信息工程大学博士生, 主要研究方向信息安全。



祝跃飞 (1962-), 男, 浙江杭州人, 博士, 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。