Journal of Computer Applications
计算机应用,2018,38(6):1745 - 1750
ISSN 1001-9081
CODEN JYIIDU
2018-06-10
http://www.joca.cn

文章编号:1001-9081(2018)06-1745-06
DOI:10.11772/j.issn.1001-9081.2017122892

基于符号执行的底层虚拟机混淆器反混淆框架

肖顺陶,周安民,刘　亮*,贾　鹏,刘露平
(四川大学 电子信息学院,成都 610065)
(*通信作者电子邮箱 906120662@qq.com)

摘　要:针对 Miasm 反混淆框架反混淆后的结果是一张图片,无法反编译恢复程序源码的问题,在对底层虚拟机混淆器(OLLVM)混淆策略和 Miasm 反混淆思路进行深入学习研究后,提出并实现了一种基于符号执行的 OLLVM 通用型自动化反混淆框架。首先,利用基本块识别算法找到混淆程序中有用的基本块和无用块;其次,采用符号执行技术确定各个有用块之间的拓扑关系;然后,直接对基本块汇编代码进行指令修复;最后,得到一个反混淆后的可执行文件。实验结果表明,该框架在保证尽量少的反混淆用时前提下,反混淆后的程序与未混淆源程序的代码相似度为96.7%,能非常好地实现 x86 架构下 C/C++ 文件的 OLLVM 反混淆。

关键词:Miasm;底层虚拟机混淆器;反混淆;符号执行;指令修复;代码相似度
中图分类号:TP393.08　　**文献标志码**:A

Obfuscator low level virtual machine deobfuscation framework based on symbolic execution

XIAO Shuntao, ZHOU Anmin, LIU Liang*, JIA Peng, LIU Luping
(*College of Electronics and Information Engineering, Sichuan University, Chengdu Sichuan* 610065, *China*)

Abstract: The deobfuscation result of deobfuscation framework Miasm is a picture, which cannot be decompiled to recovery program source code. After deep research on the obfuscation strategy of Obfuscator Low Level Virtual Machine (OLLVM) and Miasm deobfuscation idea, a general OLLVM automatic deobfuscation framework based on symbolic execution was proposed and implemented. Firstly, the basic block identification algorithm was used to find useful basic blocks and useless blocks in the obfuscated program. Secondly, the symbolic execution technology was used to determine the topological relations among useful blocks. Then, the instruction repairment was directly applied to the assembly code of basic blocks. Finally, an executable file after deobfuscation was obtained. The experimental results show that, under the premise of guaranteeing the deobfuscation time as little as possible, the code similarity between the deobfuscation program and the non-obfuscated source program is 96.7%. The proposed framework can realize the OLLVM deobfuscation of the C / C ++ files under the x86 architecture very well.

Key words: Miasm; Obfuscator Low Level Virtual Machine (OLLVM); deobfuscation; symbolic execution; instruction repairment; code similarity

0　引言

底层虚拟机混淆器(Obfuscator Low Level Virtual Machine, OLLVM)[1]是瑞士西部应用科技大学信息安全小组于2010年6月发起的一个项目,该项目的目的是提供一个基于 LLVM 编译套件的开源工具,通过代码混淆和防篡改来提高软件安全性。凭借其出色的混淆效果、自定义的混淆方案、不依赖编程语言和平台架构等特性,OLLVM 在软件安全领域得到了广泛运用。恶意软件开发者也逐步在自己的软件中使用该混淆技术,以增加安全人员的分析难度,防止自己的恶意软件被破解。由于目前关于 OLLVM 的资料比较少,加上反混淆受 OLLVM 保护的程序难度较大,极少数大型杀毒公司虽有相应的 OLLVM 反混淆方案,但出于商业目的,并未开源自己的反混淆方案,因此实现 OLLVM 反混淆对于保护软件用户的合法权益和维护健康的软件安全生态环境都具有重要的现实意义,也是一个亟需解决的问题。

在 OLLVM 反混淆方面,目前多采用基于符号执行的方法来消除控制流平展化[2]。该方面研究比较出色的是国外的 Quarkslab 团队,该团队提出了一种基于 Python 的逆向工程框架 Miasm[3],其支持可执行的可移植(Portable Executable, PE)文件、可执行与可链接格式(Executable and Linkable Format, ELF)等多种文件格式解析,并且支持 x86、高级精简指令系统处理器(Advanced RISC Machines, ARM)、无互锁管道微处理器(Microprocessor without Interlocked Piped Stages, MIPS)等多种架构平台,可以通过中间表示(Intermediate Representation, IR)[4-5]表征汇编指令语义,并使用 JIT(Just In Time)[4]技术进行代码的模拟执行。

Miasm 虽然为目前最出色的针对 OLLVM 的反混淆工具,

收稿日期:2017-12-11;**修回日期**:2018-02-07;**录用日期**:2018-02-24。

作者简介:肖顺陶(1991—),男,四川仁寿人,硕士研究生,CCF 会员,主要研究方向:移动互联网安全;　周安民(1963—),男,四川成都人,研究员,主要研究方向:安全防御与管理、移动互联网安全、云计算安全;　刘亮(1982—),男,四川成都人,讲师,硕士,主要研究方向:漏洞挖掘、恶意代码分析;　贾鹏(1988—),男,河南郑州人,博士,主要研究方向:复杂网络、移动互联网安全、二进制安全;　刘露平(1988—),男,四川洪雅人,博士,主要研究方向:二进制安全、漏洞挖掘。

但其依然存在许多问题，如反混淆后的图形颜色粗陋、基本块中的 IR 中间表示晦涩难懂、反混淆后的图形无法反编译恢复程序源码等。针对这些缺点，本文在研究符号执行工具 angr[6] 和二进制分析逆向工程框架（Binary Analysis and Reverse engineering Framework，BARF）[7] 逆向框架的基础上，在 x86 架构的 Linux 平台下，提出了一种基于符号执行的 OLLVM 自动化反混淆框架。该框架以 C/C ++ 文件经 OLLVM 混淆后得到的 ELF 文件作为输入，利用 BARF 框架进行反汇编和基本块的分析，并结合自定义的基本块识别算法确定序言、主分发器、相关块、预分发器、返回块等基本块；接着，使用符号执行工具 angr 从各基本块起始地址开始符号执行，确定真实的可达路径以及各基本块之间的前后关系；最后，修复二进制程序，包括使用无操作（No OPeration，NOP）指令填充无用基本块、有用块（包含序言、相关块、返回块）指令替换、有用块跳转偏移量修正这三个方面。由于所有的修改都是直接针对混淆后程序的汇编代码进行的，因此经过反混淆后，最终得到一个可执行文件，并能实现反编译恢复出未混淆程序的原始代码。

1 背景知识

1.1 控制流平展化

控制流平展化（Control Flow Flattening）[8] 思想最先由美国维吉尼亚大学的 Wang 等[9] 提出，早在 2008 年便有学者将控制流平展化技术运用于 C/C ++ 代码的保护[2]。控制流平展化是在不改变源程序功能的前提下，将 C/C ++ 代码中的 if/while/for/do 等控制语句转换为等价的 switch 分支结构，以消除 case 代码块之间原有的逻辑调用关系，使得所有代码块看起来都为平行关系，从而达到对程序控制流混淆的目的。

控制流平展化的思想是将源程序函数分成一个入口块和多个相邻的基本块（case 代码块），每个基本块都有相应编号，并且这些基本块都有相同的前驱和后驱模块。前驱模块也叫主分发器，通过表征程序状态的控制变量（switch 变量）来完成基本块的分发，这使得基本块的分支目标不能轻易地通过静态分析方式来确定，从而阻碍逆向分析者对程序的理解，增加逆向分析者分析时间，其模型如图 1 所示。

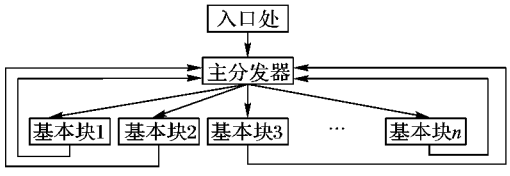

图 1 控制流平展化模型

Fig. 1 Control flow flattening model

目前 OLLVM 提供了三种混淆编译选项，本文主要使用控制流平展化选项，其可以使用-mllvm、-perFLA 两个参数来自定义控制流平展化程度，如设置参数-perFLA 的值为 100 对源程序所有函数开启控制流平展化保护。

1.2 符号执行框架 angr

符号执行（symbolic execution）[10-11] 就是把程序输入和程序变量都视为符号，并在符号执行时，更新每个变量的符号表达式。本文提出的反混淆方案使用 angr 符号执行框架，angr 是一款基于 python 的二进制文件分析框架，兼具静态分析和动态符号执行功能，其由 angr、simuvex、claripy、cle、pyvex、archinfo[6] 六大模块组成，本文主要使用前五个模块。

angr 模块是 angr 框架的分析和协调中心模块，用于完成二进制文件的分析和路径探索，本文提出的反混淆框架使用了 Project/Factory/Path 这 3 个容器。其中：Project 容器用于给待分析的二进制文件创建一个 angr 的 Project 对象；Factory 容器有多种类型，如：factory. block、factory. blank _state 等，本文则使用相应的 factory 类型来获取程序执行过程中生成的 SimState 对象。SimState 对象跟踪并记录着程序动态执行过程中的符号信息、符号对应的寄存器信息和符号对应的内存信息等；Path 容器用于存放程序动态执行过程中的 path 对象，本文使用 factory. path（state）来获取以指定程序状态 state 为起点的 path 对象，该 path 对象代表从 state 状态起点到程序执行终点的完整路径信息。

simuvex 模块是 angr 框架的 VEX IR 模拟执行引擎，负责记录程序运行状态并进行代码模拟执行。在给定 VEX IRSB（IR 基本块）和内存、寄存器的初始状态后，它可以进行静态分析和动态符号执行。

claripy 模块是 angr 框架的求解引擎，它将程序中的一些状态用符号表示，并将程序的每一个路径都翻译为一个逻辑表达式，形成表达式树（expression tree）[12]，最终得到一个非常大的路径公式，在完成公式求解后，就能得到覆盖所有路径的输入变量。本文提出的反混淆框架主要使用了 claripy 的位向量值（Bit Vector Value，BVV）构造函数 claripy. BVV（），用于指定表达式中临时变量的值，从而使程序选择不同的分支进行符号执行。

cle 模块是 angr 框架的文件解析加载器，angr 使用一个精简的加载器“CLE Loads Everything”[6]，该加载器并非完全精确，但是能加载 ELF/ARM 等可执行文件。

pyvex 模块则为 angr 框架提供了将二进制代码转换为 VEX IR 中间表示的接口。

1.3 逆向框架 BARF

BARF 是一个基于 Python 的，支持多平台和二进制指令转换为中间表示的逆向工具，其主要由 Core、Arch、Analysis[7] 三个模块组成。本文主要使用 BARF 对混淆后的二进制文件进行反汇编、生成包含逆向工程中间语言（Reverse Engineering Intermediate Language，REIL）[5] 表示的控制流图（Control Flow Graph，CFG）[13]，并提供相应的 IR 基本块分析功能。

Core 模块是 BARF 框架的核心模块，它定义了 REIL 中间表示、可满足性理论（Satisfiability Modulo Theories，SMT）求解器和二进制接口（Binary Interface，BI）。

Arch 模块指出了该框架所支持的平台架构，目前支持 x86 和 arm 两种平台。

Analysis 模块是本文提出的反混淆框架的核心功能模块，主要由 Basic Block 和 Code Analyzer 两部分组成，Basic Block 主要用于对生成的 CFG 进行分析，如获取基本块地址等，Code Analyzer 主要用于获取相应的 SMT[14-15] 表达式。

2 反混淆框架设计与实现

要实现 OLLVM 的反混淆，主要面临以下三个问题：

1）识别出有用块和无用块。经 OLLVM 控制流平展化混淆的程序中会增加很多无用的基本块以混淆程序逻辑，这就

需要设计有效的基本块识别算法,找出有用的基本块和无用的基本块。

2)确定有用块之间的前后关系,得到真实有效的程序执行路径,因为混淆程序中的很多基本块跳转逻辑并不是程序的实际执行流程。

3)修复二进制程序。在使用 NOP 指令填充无用基本块后,为使程序正常运行,我们需要对跳转指令的跳转偏移量进行修正;同时,还需要将 cmov 条件传送指令改写成相应的条件跳转指令,并在其后添加一条 jmp 指令,使其跳向另一分支。

围绕以上三个问题,本文提出了一种基于符号执行的 OLLVM 自动化反混淆框架,其架构如图 2 所示。

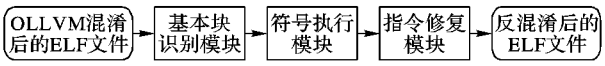

图 2　OLLVM 自动化反混淆框架架构图

Fig. 2　Architecture diagram of OLLVM automatic deobfuscation framework

2.1　基本块识别模块

为了得到有效的反混淆结果,需要找到混淆程序中的有用块和无用块,这里的有用块是指对恢复原始控制流图有用的基本块,包含序言、相关块、返回块,而无用块是指 OLLVM 混淆后添加的无用基本块,详细定义如下:

1)序言块:函数的起始块;

2)主分发器:序言块的直接后继块 ;

3)预分发器:后继为主分发器的块(非序言块) ;

4)相关块:后继为预分发器的块;

5)无后继的块为返回块;

6)剩下的为无用块。

其次,BARF 框架 Analysis 模块中的 Basic Block 子模块提供了 CFG 控制流图恢复功能,由该模块即可获得基本块的首地址、基本块指令、基本块分支等信息,因此可结合 BARF 框架的基本块分析功能设计相应的基本块识别算法。在对 OLLVM 控制流平展化混淆策略进行深入研究后,根据上述基本块定义,设计了如下的基本块识别算法:算法中 cfg 为 BARF 框架解析输入文件后得到的程序控制流图;blocks 代表程序控制流图中的所有基本代码块;predispatcher_Retn(cfg)用来获取返回块、预分发器的首地址;relevant_NOP_Blks(cfg)用来获取相关块、无用块的首地址,算法具体描述如下。

输入　OLLVM 混淆后的 ELF 文件;

输出　各基本块首地址。

```
Begin
  序言←sys.argv[2]                          /*序言块首地址*/
  主分发器←序言直接后继
  function predispatcher_Retn(cfg)
    for each block ∈ blocks do
      if block 没有分支 then
          返回块←block
      elif block 直接后继为主分发器 then
          预分发器←block
      end if
    end for
    return 预分发器,返回块
  end function
  function relevant_NOP_Blks(cfg)
    相关块←[ ]
    无用块←[ ]
    for each block ∈ blocks do
      if block 直接后继为预分发器 and
        block 指令条数 >2 then
          添加 block 到相关块
      elif block 非序言非返回块 then
          添加 block 到无用块
      end if
    end for
    return 相关块,无用块
  end function
End
```

相比 Miasm,本文提出的基本块识别算法,采用了更严格的相关块定义(指令数大于 2 条的后继为预分发器的块才是相关块,指令数小于等于 2 条的后继为预分发器的块只是完成到预分发器的跳转,对恢复程序控制流图没有意义,Miasm 在后续处理中也删除了这些无用的相关块),因而能在保证基本块完备性的前提下,得到真实有用的相关块。

由于在运行框架之初就会给定混淆文件特定函数入口地址(即序言块首地址),而且经 OLLVM 混淆后,序言块只有 1 个后继块,因此通过 BARF 框架获得的序言块直接后继块地址、以及没有后继分支的基本块地址(分别对应主分发器首地址、返回块首地址)都是正确的;依此类推,后续得到的基本块地址也都是正确的,这就保证了所得基本块首地址的正确性。

综上,经过上述算法的处理,即可得到完备的、正确的有用块和无用块的首地址列表,这为下一步的符号执行做好了准备。

2.2　符号执行模块

OLLVM 将源程序的控制流打乱了,因此需要找到真实的控制流,这里采用符号执行的方式进行路径探索,确定各个有用块之间的前后顺序以及分支关系。

虽然 Miasm 和 angr 两种符号执行引擎都能得到混淆程序的控制流字典变量{父节点:(子节点集合)},但 Miasm 所获得的控制流字典变量中,当父节点有 2 个子节点(即有分支)时,并不能看出子节点是满足条件的分支,还是不满足条件的分支,必须结合其符号执行的约束条件才能判断;而本文基于 angr 的符号执行模块,在符号执行过程中,当遇到有分支的相关块时,首先将存在分支的指令保留下来,然后利用 angr 的求解引擎 Claripy 设置两个 1 b 的位向量值〈BV1 1〉、〈BV1 0〉,分别进行符号执行,从而得到父节点的满足条件的左子节点和不满足条件的右子节点,即意义明确的控制流字典变量{父节点:(子节点集合)}。由于基本块识别模块已经得到完备的正确的有用块地址列表,因而能保证符号执行后得到正确的有用块之间的拓扑关系,详细实现过程如下:

首先,利用符号执行工具 angr,从基本块识别模块得到的有用块地址列表中的每一个地址(除去返回块首地址,因为返回块没有后续分支)处开始符号执行,一旦遇到 cmov 类型的条件传送指令,就将该基本块指令保存下来,并设置基本块分支标志位 has_branches 为 True,并且通过 angr 的求解引擎 Claripy 设置两个 1 b 的位向量值〈BV1 1〉、〈BV1 0〉,分别代表条件满足和条件不满足时的约束条件,用于在符号执行过程中改变临时变量的值,从而强行使程序走右侧条件满足时的分支或者走左侧条件不满足时的分支,以达到完整路径遍

历的目的；接着，在符号执行函数 symbexec（）中，调用 BP_inspect（）函数监控 angr IR 基本块中是否出现 ITE（If-Then-Else）[16] 条件表达式，一旦出现 ITE 条件表达式便在此处设置断点，并依次将 ITE 条件表达式中临时变量的值修改为〈BV1 1〉、〈BV1 0〉，继续符号执行，分别得到左右两侧不同分支对应的基本块首地址。当依次完成有用块地址列表中地址的遍历后，就能得到有用块之间的前后顺序及其分支关系了。

另外，如果遇到 call 指令，则将该 call 指令地址保存下来作为 hook 函数的 hook 地址，并使用 hook 的方式直接返回，而不去执行 call 指令调用的子程序，因为我们更关注的是该基本块的整体轮廓。经过上述步骤后，即可得到有用块之间正确的拓扑关系，但无用的基本块仍然存在，为了还原程序真实的控制流图，还需使用 NOP 指令填充无用的基本块，并完成有用块的指令修复，这就是指令修复模块的主要工作。

2.3 指令修复模块

混淆后的程序从一个有用块跳转至下一个有用块时，它们之间隔着许多无用的基本块，因此在使用 NOP 指令填充无用块后，还需修正跳转指令的跳转偏移量，使其能正常跳转到下一个有用块，所以指令修复模块主要完成无用块指令填充、有用块指令修复两方面的工作。

在指令修复模块中创新性地使用指令修复技术，通过对无用块进行 NOP 指令填充；对有用块（含直接跳转块（只有一个子节点）、有分支块（有两个子节点））进行指令替换、指令填充、偏移修正，最终得到一个可执行的反混淆文件，从而克服了 Miasm 由于反混淆后的结果是一种图片、无法进行反编译等缺点。由于符号执行模块已经得到了有用块之间正确的拓扑关系并且保存下了有分支的指令，因此可依据符号执行结果进行指令修复工作，并能确保指令替换、偏移修正工作的有效性，从而保证反混淆结果能正常运行，并完成对程序语义的完整还原，得到正确的反混淆结果。

为此，本文提出的反混淆框架编写了 2 个具有通用性的指令修复函数以自动完成指令修复工作。nop_padding（）用于向指定地址填充 0x90；jmp_padding（）用于填充新的跳转偏移量。对于无用块，只需调用 nop_padding（）函数使用 NOP 指令填充该基本块；对于有用块则须进行直接跳转块指令修复、有分支块指令修复。

2.3.1 直接跳转块

对于直接跳转块，只需将该基本块的最后一条指令改写为 jmp 指令，并完成跳转偏移量的修正。具体如下：首先，向该基本块最后一条指令首地址填充 jmp 的 opcode 值；接着调用 nop_padding（）函数向其后 4 个地址填充 0x90；然后按照下一跳有用块首地址，计算出修正的跳转偏移量，并调 jmp_padding（）函数填充新的跳转偏移量即可。

2.3.2 有分支块

对于含有分支的基本块，须将 cmov 指令改写成相应的条件跳转指令跳至符合条件的分支（如：cmovx 指令可改写成 jx 指令），并在其后添加一条 jmp 指令。jx 指令的下一跳是满足 X 条件的右分支基本块，jmp 指令的下一跳是不满足 X 条件的左分支基本块，然后分别根据两个分支基本块首地址进行跳转偏移量修正即可，分支基本块的首地址可从符号执行模块中获取。

3 实验评估

测试环境为装有最新版 LLVM-4.0 混淆框架、本文提出的 OLLVM 反混淆框架、Miasm 反混淆框架的 Ubuntu16.04 amd64 系统计算机，详细配置为：CPU 型号 Intel Core i7-4790 @ 3.60 GHz，内存 12 GB。

3.1 反混淆用时测试

经 OLLVM 混淆后的程序（为 ELF 文件）含有非常多的分支，在对其进行反混淆，特别是符号执行时势必会造成时间开销，因此本节就从反混淆用时指标上对本文提出的反混淆框架性能进行测试，并与 Miasm 反混淆框架性能进行对比。

测试方法一　将从 SPECint-2000[17]（https://www.spec.org/cpu2000/CINT2000/）下载的 200 个 C/C++ 程序使用相同的 OLLVM 混淆策略进行混淆后，便得到了 200 个 ELF 格式的样本程序。首先使用本文提出的反混淆框架对每个测试样本进行反混淆，为减小误差，取 20 次反混淆时间的平均值作为该样本最终的反混淆用时，直至完成所有样本的测试；然后，重启电脑，改用 Miasm 反混淆框架，仍采用上述的测试方案，完成所有样本的反混淆用时测试。为方便展示，此处从 200 个样本程序中选取 10 个具有较大范围覆盖率并具有代表性的样本程序进行结果展示，实验结果如图 3 所示。

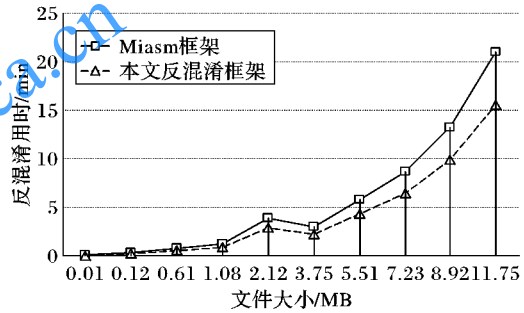

图 3　两种框架反混淆用时对比

Fig. 3　Time consumption comparison of deobfuscation for two frameworks

表 1 详细列出了测试时使用的 10 个样本程序大小，及各程序所包含的分支循环数。由图 3 可知，ELF 文件大小为 2.12 MB时，两个框架的反混淆用时增幅均较为明显，对比表 1 发现，该 ELF 文件对应的分支循环个数为 2 427 个，远远多于大小为 1.08 MB 和 3.75 MB 的 ELF 文件的分支循环个数。综合各实验数据可知，反混淆用时和 ELF 文件大小无关，而与分支循环个数呈正相关，分支循环个数越多，符号执行时耗费的时间也越多。

表 1　OLLVM 混淆后的 ELF 文件

Tab. 1　OLLVM obfuscated ELF files

测试程序	ELF 大小/MB	分支循环个数	测试程序	ELF 大小/MB	分支循环个数
1	0.01	155	6	3.75	1 732
2	0.12	430	7	5.51	2 669
3	0.61	737	8	7.23	3 250
4	1.08	952	9	8.92	4 055
5	2.12	2 427	10	11.75	5 230

其次，本实验中 Miasm 框架反混淆用时为 7 s ~ 21 min 不等，而本文提出的反混淆框架为 5 s ~ 16 min 不等，在混淆程序分支个数不多的情况下，二者反混淆用时较为接近，但随着

分支循环个数的不断增加,本文提出的反混淆框架反混淆用时优势越发明显。

综上所述,本文提出的反混淆框架相比 Miasm 框架具有更优秀的反混淆用时表现。

3.2　反混淆结果代码相似性测试

反混淆中最关心的是反混淆结果的正确性,因此为验证本文提出的反混淆框架的正确性,将从反混淆结果程序语义正确性(以代码相似性来衡量)、反混淆结果运行正确性两个方面进行测试。

由于 Miasm 框架反混淆后得到的只是保留程序语义的 Miasm IR Graph,无法比较代码相似性,因此只对本文提出的反混淆框架做该项测试。

测试方法二　使用 GNU 编译套件(GNU Compiler Collection, GCC)依次将 3.1 节中的 200 个 C/C ++ 文件编译为可执行文件,并使用二进制文件对比工具 BinDiff[18-19](http://www.zynamics.com/bindiff.html)逐个比较反混淆后程序与未混淆源程序的代码相似性。为进一步凸显本文提出的反混淆框架性能以及方便展示,这里仅展示 3.1 节中分支最多的 10 个程序的代码相似性对比结果,如图 4 所示。

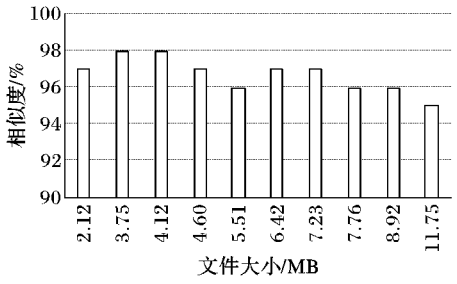

图 4　代码相似性

Fig. 4　Code similarity

从图 4 数据可知,10 个测试样本的平均代码相似度为 96.7%,表明反混淆后的程序与未混淆源程序具有极高的代码相似度。

3.3　反混淆结果运行正确性测试

由于反混淆后的程序均为可执行文件,因此可以通过对比未混淆程序和反混淆后程序的运行结果,来判断反混淆结果的正确性。

测试方法三　分别运行 3.2 节中的 200 个未混淆源程序及与之对应的反混淆后的程序,对比二者的运行结果一致度。为方便展示,此处仅展示分支循环个数最多的 10 个程序的未混淆程序与反混淆后程序运行对比结果,如表 2 所示。

由表 2 可知,未混淆源程序与反混淆后程序的执行结果完全一致,这说明反混淆结果是正确的,加之反混淆后的程序与未混淆源程序具有极高的代码相似度(见图 4),因此可以利用 IDA 等工具对反混淆后的程序进行反编译以恢复源程序代码。

3.4　对比分析

本节将从多文件格式支持、多平台架构支持、反混淆效果三方面对 Miasm 框架和本文提出的反混淆框架进行对比,详细的对比结果如表 3 所示。

由表 3 可知,本文提出的 OLLVM 反混淆框架的反混淆效果明显优于 Miasm 反混淆框架,特别是在反混淆结果的后续处理上,Miasm 反混淆后得到的只是保留程序语义的 Miasm IR Graph,不仅图形粗陋,而且图形中使用晦涩难懂的 Miasm IR 语言;更重要的是,其反混淆结果无法反编译以恢复未混淆程序源代码,而本文提出的通用型自动化反混淆框架直接针对 OLLVM 混淆后的程序的汇编代码进行操作,在保证程序语义完整性前提下,最终得到一个结果正确的可执行文件,因而可通过使用 IDA 进行加载,获得颜色分明、层次清晰的程序控制流图,并且控制流图中的代码块采用易懂的汇编语言;另外,还能通过 IDA 对反混淆后的程序进行反编译以恢复源程序的 C/C ++ 代码。这些都说明了本文提出的反混淆框架相比 Miasm 框架具有更出色的反混淆性能。

表 2　未混淆程序与反混淆后程序运行结果一致度

Tab. 2　Consistency of non-obfuscated and deobfuscated program in operation result

测试程序	未混淆程序大小/MB	反混淆后程序大小/MB	运行结果一致度/%
1	0.70	2.21	100
2	2.67	3.75	100
3	2.65	4.12	100
4	3.06	4.60	100
5	3.84	5.51	100
6	4.51	6.42	100
7	5.20	7.23	100
8	5.32	7.76	100
9	6.39	8.92	100
10	8.48	11.75	100

表 3　不同框架反混淆性能对比

Tab. 3　Deobfuscation performance comparison of different frameworks

对比项	Miasm	本文反混淆框架
多文件格式解析	√	√
支持多平台架构	√	√
反混淆结果易读	×	√
反混淆结果可执行	×	√
能恢复程序源码	×	√

4　结语

OLLVM 混淆技术的流行为恶意软件的肆虐提供了土壤,针对这一问题以及 Miasm 框架在 OLLVM 反混淆方面的缺陷,本文提出了一种基于符号执行的 OLLVM 通用型自动化反混淆框架,该框架结合符号执行、指令修复技术很好地克服了 Miasm 框架的缺点,并能恢复出未混淆程序源代码,实验结果表明该框架相比 Miasm 框架具有更优秀的反混淆性能。

目前本文提出的反混淆框架只能很好地实现 x86 架构下 C/C ++ 文件的 OLLVM 反混淆,而 Android SO 文件因采用 arm 指令,其在基本块识别上与 x86 指令具有不同的规则,但二者在反混淆的思路、核心技术上具有共通性,下一步将以此为基础研究 Android SO 文件的 OLLVM 反混淆。

参考文献(References)

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

[2]　赵楷. 基于控制流平展化代码变形技术的研究[D]. 哈尔滨: 哈尔滨工业大学, 2009: 16-18. (ZHAO K. Code obfuscation based on control flow flattening technique [D]. Harbin: Harbin Institute of Technology, 2009: 16-18.)

[3] DESCLAUX F. Miasm: framework de reverse engineering [EB/OL]. [2017-10-20]. https://www.sstic.org/media/SSTIC2012/SSTIC-actes/miasm_framework_de_reverse_engineering/SSTIC2012-Article-miasm_framework_de_reverse_engineering-desclaux_1.pdf.

[4] DUBOSCQ G, WÜRTHINGER T, STADLER L, et al. An intermediate representation for speculative optimizations in a dynamic compiler [C]// Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages. New York: ACM, 2013: 1-10.

[5] DULLIEN T, PORST S. REIL: a platform-independent intermediate representation of disassembled code for static code analysis [EB/OL]. http://moflow.org/ref/REIL%20-%20A%20platform-independent%20intermediate%20representation%20of%20disassembled%20code%20for%20static%20code%20analysis.pdf.

[6] SHOSHITAISHVILI Y, WANG R Y, SALLS C, et al. SOK: (state of) the art of war: offensive techniques in binary analysis [C]// Proceedings of the 2016 IEEE Symposium on Security and Privacy. Piscataway, NJ: IEEE, 2016: 138-157.

[7] HEITMAN C, ARCE I. BARF: a multiplatform open source binary analysis and reverse engineering framework [EB/OL]. [2017-10-20]. http://sedici.unlp.edu.ar/bitstream/handle/10915/42157/Documento_completo.pdf?sequence=1.

[8] LÁSZLÓ T, KISS Á. Obfuscating C++ programs via control flow flattening [EB/OL]. [2017-10-20]. https://www.inf.u-szeged.hu/~akiss/pub/pdf/laszlo_obfuscating.pdf.

[9] WANG C X. A security architecture for survivability mechanisms [D]. Charlottesville, VA: University of Virginia, 2000: 5-10.

[10] 陈厅. 动态程序分析技术在软件安全领域的研究[D]. 成都: 电子科技大学, 2013: 6-15. (CHEN T. Research on dynamic program analysis technique for software security [D]. Chengdu: University of Electronic Science and Technology of China, 2013: 6-15.)

[11] 黄晖, 陆余良, 夏阳. 基于动态符号执行的二进制程序缺陷发现系统[J]. 计算机应用研究, 2013, 30(9): 2810-2812. (HUANG H, LU Y L, XIA Y. Dynamic symbolic execution based defect detection system for binary programs [J]. Application Research of Computers, 2013, 30(9): 2810-2812.)

[12] HASTIE T, TIBSHIRANI R, BOTSTEIN D, et al. Supervised harvesting of expression trees [J]. Genome Biology, 2001, 2(1): 1-12.

[13] BRUSCHI D, MARTIGNONI L, MONGA M. Detecting self-mutating malware using control-flow graph matching [C]// Proceedings of the 2006 International Conference on Detection of Intrusions and Malware & Vulnerability Assessment, LNCS 4064. Berlin: Springer, 2006: 129-143.

[14] 任胜兵, 吴斌, 张健威, 等. 基于可满足性模理论求解器的程序路径验证方法[J]. 计算机应用, 2016, 36(10): 2806-2810. (REN S B, WU B, ZHANG J W, et al. Method of program path validation based on satisfiability modulo theory solver[J]. Journal of Computer Applications, 2016, 36(10): 2806-2810.)

[15] SEBASTIANI R. Lazy satisfiability modulo theories [J]. Journal on Satisfiability Boolean Modeling & Computation, 2007, 3(3): 141-224.

[16] Rhelmot. Intermediate representation [EB/OL]. [2017-07-14]. https://github.com/angr/angr-doc/blob/master/docs/ir.md.

[17] EGELE M, WOO M, CHAPMAN P, et al. Blanket execution: dynamic similarity testing for program binaries and components [C]// Proceedings of the 23rd USENIX Conference on Security Symposium. New York: ACM, 2014: 303-317.

[18] CHOW S, GU Y, JOHNSON H, et al. An approach to the obfuscation of control-flow of sequential computer programs [C]// Proceedings of the 2001 4th International Conference on Information Security, LNCS 2200. Berlin: Springer, 2001: 144-155.

XIAO Shuntao, born in 1991, M. S. candidate. His research interests include mobile Internet security.

ZHOU Anmin, born in 1963, reasearch fellow. His research interests include security defense and management, mobile Internet security, cloud computing security.

LIU Liang, born in 1982, M. S., lecturer. His research interests include vulnerability mining, malicious code analysis.

JIA Peng, born in 1988, Ph. D. His research interests include complex network, mobile Internet security, binary security.

LIU Luping, born in 1988, Ph. D. His research interests include binary security, vulnerability mining.

(上接第 1744 页)

[26] PRAJNA S, PAPACHRISTODOULOU A, PARRILO P A. Introducing SOSTOOLS: a general purpose sum of squares programming solver [C] // Proceedings of the 41st IEEE Conference on Decision and Control. Piscataway, NJ: IEEE, 2002: 741-746.

[27] PAPACHRISTODOULOU A, PRAJNA S. A tutorial on sum of squares techniques for systems analysis [C] // Proceedings of the 2005 American Control Conference. Piscataway, NJ: IEEE, 2005: 2686-2700.

[28] EVANS L C. An Introduction to Stochastic Differential Equations [M]. Berkeley, CA: University of California-Berkeley, 2013: 79-101.

[29] BENSIMHOUN M. N-dimensional cumulative function, and other useful facts about Gaussians and normal densities [EB/OL]. [2017-10-16]. https://upload.wikimedia.org/wikipedia/commons/a/a2/Cumulative_function_n_dimensional_Gaussians_12.2013.pdf.

[30] SLOTH C, WISNIEWSKI R. Safety analysis of stochastic dynamical systems [J]. IFAC-PapersOnLine, 2015, 48(27): 62-67.

This work is partially supported by the National Natural Science Foundation of China (61772203, 61632015, 61561146394), the Natural Science Foundation of Shanghai (17ZR1408300).

SHEN Minjie, born in 1993, M. S. candidate. His research interests include software engineering.

ZENG Zhenbing, born in 1963, Ph. D., professor. His research interests include symbolic computation.

LIN Wang, born in 1982, Ph. D., associate professor. His research interests include formal method.

YANG Zhengfeng, born in 1980, Ph. D., associate professor. His research interests include software engineering.