·安全技术· 文章编号：1000-3428(2018)05-0113-06 文献标志码：A 中图分类号：TP309

基于操作码合并的 Python 程序防逆转算法

王小强[1a,2]，顾乃杰[1a,1b,2]

（1. 中国科学技术大学 a. 计算机科学与技术学院；b. 先进技术研究院，合肥 230027；
2. 安徽省计算与通信软件重点实验室，合肥 230027）

摘 要： 由 Python 编程语言编写的程序，其编译生成的字节码是针对 Python 虚拟机的具有特定结构的文件，该文件很容易被逆向工具反编译，从而损害开发者的经济利益和个人隐私。传统的防逆转方法存在其处理后的字节码文件易被破解、程序运行效率低等问题。为此，提出一种新的 Python 字节码文件保护算法。在不影响程序执行结果的前提下，将 Python 字节码文件中的多个操作码合并为一个新操作码，改变操作码序列的结构和语义，最终达到防逆转的目的。实验结果表明，该算法不仅能防止 Python 字节码文件被反编译，而且可以减小字节码文件的存储空间，提升程序执行效率。

关键词： 字节码文件；反编译；防逆转；虚拟机操作码；操作码合并

中文引用格式： 王小强，顾乃杰. 基于操作码合并的 Python 程序防逆转算法[J]. 计算机工程，2018，44(5)：113-118.
英文引用格式： WANG Xiaoqiang，GU Naijie. Python Program Anti-reversal Algorithm Based on Opcode Merging[J]. Computer Engineering，2018，44(5)：113-118.

Python Program Anti-reversal Algorithm Based on Opcode Merging

WANG Xiaoqiang[1a,2]，GU Naijie[1a,1b,2]

（1a. School of Computer Science and Technology；1b. Institute of Advanced Technology，
University of Science and Technology of China，Hefei 230027，China；
2. Anhui Province Key Laboratory of Computing and Communication Software，Hefei 230027，China）

【Abstract】 The program written by Python programming language is compiled and generated bytecode，which is a specific structure for Python virtual machine. The file is easily decompressed by reverse tools，thereby damaging the economic interests and personal privacy of developers. The traditional method of anti-reversal has the problems of easily deciphered bytecode files and low program efficiency after processing. To solve the above problem，a new Python bytecode file protection algorithm is proposed，which mereges multiple opcodes in the Python bytecode file into a new opcode without changing the execute result of the program，changes the opcode sequence structure and semantics，and achieves the purpose of anti-reversal finally. Experimental results show that the proposed algorithm not only prevents the bytecode files from being decompilated，but also reduces the storage space of the bytecode file and improves the efficiency of program execution.

【Key words】 bytecode file；decompile；anti-reversal；virtual machine opcode；opcode merging
DOI： 10.19678/j.issn.1000-3428.0046314

0 概述

Python 是一种面向对象的解释型计算机程序设计语言，由于语法简洁清晰、简单易学、免费、开源、扩展性强等优点，自从 20 世纪 90 年代诞生以来，就广受欢迎和使用，如网络文件同步工具 Dropbox、社交新闻站点 Reddit、网络问答社区知乎，都是由 Python 编程语言所开发。Python 字节码文件(.pyc)由 Python 编程语言编写的程序编译而成，与传统的针对特定处理器和操作系统的二进制文件相比，Python 字节码文件保留了 Python 源码文件中的全部信息，是针对 Python 虚拟机的具有特定结构和特征的文件，具有可跨平台使用的特性。正是因为字节码文件的这种结构和特征，导致其极易被攻击者反编译出其中的源码，如使用 uncompyle2[1]、Decompyle ++[2]、Easy Python Decompiler[3]等工具就可以将 Python 字节码文件反编译为源码文件，这不仅会造成重要数据结构、算法、业务逻辑的暴露，

基金项目： 安徽省自然科学基金(1408085MKL06)；高等学校学科创新引智计划项目(B07033)。
作者简介： 王小强(1991—)，男，硕士，主研方向为软件安全、软件脆弱性检测；顾乃杰，教授、博士生导师。
收稿日期： 2017-03-10 **修回日期：** 2017-05-16 **E-mail：** wxq9102@mail.ustc.edu.cn

而且会给开发者带来巨大的经济损失,有时甚至具有严重的安全隐患。

针对上述问题,本文提出一种基于操作码合并的 Python 程序防逆转算法。在不影响 Python 程序正确执行的前提下,引入新操作码对原 Python 操作码集进行扩充,并在编译生成字节码文件时使用新操作码来代替操作码序列中的 2 个或多个操作码,从而缩短 Python 字节码文件中操作码序列的长度,改变操作码序列的结构和语义。

1 传统 Python 字节码文件防逆转方法

1.1 代码混淆

代码混淆旨在通过布局混淆、数据混淆、控制流混淆[4-5]等措施,将一个程序转换为能够妨碍攻击者理解其中算法和数据结构或能够阻止攻击者从程序文本中提取有价值信息的另一种形式[6-7]。

虽然 Java、Python 等脚本语言经代码混淆后编译生成的字节码文件可以跨平台使用,但是其仍然遵守原来的文件格式和指令集,因此,代码混淆技术对 Python 字节码文件的保护能力不足。此外,代码混淆在处理多文件项目中的导入模块和对象名称时有局限性,且该方法会给代码执行效率带来一定的影响。

1.2 本地编译

本地编译是一种将 Python 脚本程序和解释器一起编译为平台相关程序的技术,其工作步骤为[8]:首先编写 Python 源码程序,然后借助特定的工具(如 py2exe[9])将目标脚本程序进行转换并且将 Python 解释器编译为平台相关的动态链接库,最后运用一个额外的可执行程序来运行该动态链接程序和转换后的文件。

虽然本地编译技术对 Python 程序起到了一定的保护作用,但是采用这种方式,Python 程序在发布时需要同时发布解释器对应的动态链接程序和脚本文件所依赖的所有库文件,导致目标程序所占用的空间大大增加。

1.3 数字水印

数字水印通常是永久镶嵌在宿主数据中的具有可鉴别性的数字信号或模式,其可以用来标识作者、所有者、发行者、使用者等信息[10-11],且不影响宿主数据的可用性。使用数字水印技术并不能阻止字节码文件被反编译[12],但是能够阻止数字产品被偷窃,或者当偷窃发生时为使用者提供数字产品的拥有权证明[13]。

虽然理论上水印能够作为软件所有权的有力凭证,但实际中无论是静态水印还是动态水印,都很容易通过混淆变换和优化等措施从软件中被移除[14-15],因此,水印技术并不能为 Java、Python 等字节码文件提供有效保护。

1.4 操作码替换

除上述防逆转方法外,研究人员在操作码替换领域也进行了研究。操作码替换技术是将字节码文件中的每个操作码替换为其他操作码,使得编译生成的字节码文件对于标准的解释器而言包含非法的操作码序列,以此达到防逆转的目的。

操作码替换技术已分别被基于虚拟机 Dropbox[16]和基于 PC 的应用,在操作码随机化的安全[17]中实现,且网络上已有工具 python-obfuscation[18]对 Python 操作码进行随机替换。但是,该技术的实质是利用密码学中的单表代替密码[19],如果攻击者了解采取的保护方式,就可以利用操作码的一些统计学规律进行分析,如采用频率分析进行攻击,因此,该技术的安全性不足。

2 操作码合并

Python 字节码文件易被反编译的主要原因是字节码文件保留了 Python 源码文件的所有信息,且字节码文件中操作码序列的结构和每个操作码的含义极易被攻击者分析和理解。因此,改变操作码序列的结构或隐藏操作码的含义,是保护 Python 字节码文件的重要方法,但是,这类方法又会对虚拟机执行字节码文件的结果造成影响。鉴于此,本文将探索如何利用窥孔优化策略,在不影响字节码文件正确执行的前提下,通过引入新操作码来对 Python 操作码集进行扩充,进而改变 Python 字节码文件中操作码序列的结构,最终达到防逆转的目的。

2.1 基本概念

定义 1 操作码

操作码即 Python 虚拟机中的指令码,占 1 Byte 的长度,其规定了 Python 虚拟机需要执行哪一条指令。目前,Python 2.7.9 版本的虚拟机规范中定义了 110 个操作码,其中不带参数的操作码共 61 个,带一个参数的操作码共 49 个,每个参数占 2 Byte 的长度。

定义 2 操作码序列

一个字节码文件的操作码序列 S 由操作码和其所带的参数构成,操作码序列结构如图 1 所示,其中,OP_i 为操作码,ARG_{ij} 为 OP_i 的参数。OP_i、ARG_{ij} 均为正整数,$1\leqslant i\leqslant n$,$1\leqslant j\leqslant 2$,n 为一个字节码文件所包含的操作码总数。

OP_1	OP_2	ARG_{21}	ARG_{22}	OP_3	ARG_{31}	ARG_{32}	…

图 1 操作码序列示意图

Python 虚拟机执行字节码文件的本质是利用循环依序读取操作码序列中的一个操作码 OP_i 和 2 个字节参数 ARG_{i1} 与 ARG_{i2},然后查找并执行 OP_i 对应的解释程序,修改虚拟机内部众多的状态值,接着再读取下一个操作码 $OP_{(i+1)}$ 和字节参数 $ARG_{(i+1)1}$ 与 $ARG_{(i+1)2}$,重复上述步骤直至操作码序列读取结束。

定义 3　基本块

基本块是由 S 中顺序执行的若干个操作码构成的序列。通常情况下,若 S 中没有出现条件跳转、绝对跳转操作,则这些操作码处于同一个基本块中。

定义 4　基本块信息

字节码文件的基本块信息 B 是一个由正整数构成的单调不减序列,其长度与操作码序列 S 相同,且其中的每个元素与 S 中的每个操作码一一对应,元素值为对应的操作码在 S 中所在的基本块号。图 2 所示为一个基本块信息,从图中可以看出,S 中第 1 个操作码处于第 1 个基本块,第 2 个 ~ 第 5 个操作码处于第 2 个基本块,依此类推。

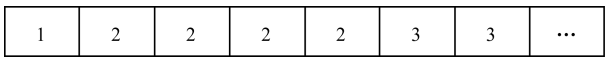

图 2　基本块信息示意图

2.2　操作码合并算法

本文提出的操作码合并算法流程如图 3 所示。其中,操作码序列提取与频率统计的目的是寻找待合并的操作码序列;基本块信息提取是为了从大量的待合并操作码序列中选出可以合并的序列,减少对操作码集扩充的干扰;操作码集扩充是添加新操作码的定义与解释执行。

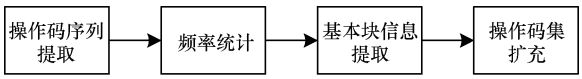

图 3　操作码合并算法流程

2.2.1　操作码序列提取

为了使操作码合并算法适用于不同的 Python 应用程序,且合并后的操作码序列结构较合并前发生较大变化,需要从大量的操作码序列中选择最常出现的操作码排列进行合并,即从 Python 应用程序文件 $\text{file}^1.\text{py}, \text{file}^2.\text{py}, \cdots, \text{file}^k.\text{py}$ 编译生成的字节码文件 $\text{file}^1.\text{pyc}, \text{file}^2.\text{pyc}, \cdots, \text{file}^k.\text{pyc}$ 中,提取每个字节码文件 $\text{file}^x.\text{pyc}$ 包含的操作码序列 $S^x = [OP_1^x, ARG_{11}^x, ARG_{12}^x, OP_2^x, ARG_{21}^x, ARG_{22}^x, \cdots, OP_n^x, ARG_{n1}^x, ARG_{n2}^x]$,其中,$1 \leqslant x \leqslant k, k >> 100$。

本文提取 Python/Lib 下的1 950 个库文件源程序对应的 1 950 个操作码序列。这些程序通常提供接口给其他模块进行导入,其功能多样且不会随意发生改变,选它们进行操作码序列提取较为合理。1 950 个操作码序列中,最长的操作码序列包含 10 000 多个操作码,最短的也包含 60 个操作码,共有操作码240 263 个。

2.2.2　频率统计

频率统计的目的是获取候选合并对象。在 2.2.1 节获取多个操作码序列 S^x 的基础上,统计出 S^x 中所有长度为 len 的操作码子序列出现的次数,并按出现次数从高到底进行排序,其中,$2 \leqslant len, 1 \leqslant x \leqslant k$。

2.2.3　基本块信息提取

基本块信息的提取是在众多的候选子序列中筛选出可以合并的子序列。在 2.2.2 节的排序结果中,靠前的对象并不一定都是可以进行合并的操作码子序列,原因是有些子序列中的操作码处于不同的基本块中,如果将该类操作码进行合并,将会破坏程序的执行逻辑。因此,需要进一步根据程序执行逻辑来提取上述操作码序列 S^x 的基本块信息 B^x,计算表达式如下:

$$B^x = [VAL_1^x, VAL_2^x, \cdots, VAL_i^x, VAL_{i+1}^x, \cdots, VAL_n^x]$$

其中,n 为 S^x 中操作码的个数,$1 \leqslant i < n$。

若 B^x 中存在 $VAL_i^x = VAL_{i+1}^x$,则 S^x 中 OP_i^x 与 OP_{i+1}^x 处于同一个基本块中,是一对可以合并的操作码。如表 1 所示,为上述 1 950 个操作码序列中部分长度为 2 Byte 且处于同一个基本块中的操作码子序列以及其出现的次数。

表 1　部分长度为 2 Byte 的操作码子序列以及其出现次数

OP_1	OP_2	出现次数
LOAD_CONST	MAKE_FUNCTION	9 269
LOAD_CONST	IMPORT_NAME	7 762
BUILD_CLASS	STRORE_NAME	6 285
IMPORT_FROM	STORE_NAME	3 732
CALL_FUCNTION	POP_TOP	2 871

2.2.4　操作码集扩充

操作码集扩充是利用窥孔优化策略将出现频率较高且处于同一个基本块中的操作码子序列($OP_1, OP_2, \cdots, OP_{\text{len}}$)合并为一个新的操作码 OP_{new},并在 Python 虚拟机中添加对 OP_{new} 的解释执行过程,使得对 OP_{new} 的解释执行效果与依次执行($OP_1, OP_2, \cdots, OP_{\text{len}}$)的效果等价。

如图 4 所示,利用操作码合并算法可以将 S^x 中的 2 个操作码 OP_1 与 OP_2 合并为一个操作码 OP_{new},从而将 S^x 转换为一个新的操作码序列 S_{new}^x。在 S_{new}^x 中 OP_{new} 带 2 个参数,因此,在解释 OP_{new} 时需要一次将 2 个参数都读出来。

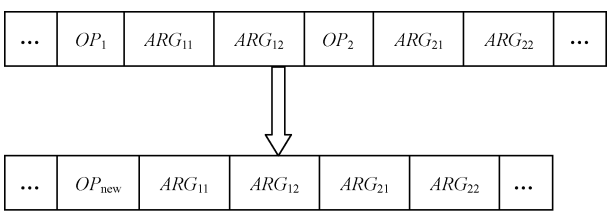

图 4　操作码合并示意图

因为在 Python 字节码序列中任意一个操作码仅占 1 Byte 的大小,所以最多可以定义 2^8 个操作码。而在 Python 2.7.9 版本的虚拟机规范中仅使用了其中的 110 个操作码,因此,可以利用剩余的 146 个操作码来定义新操作码,从而在利用操作码合并缩短序列

的同时隐藏操作码序列的含义。本文对表 1 中的 5 个子序列进行合并,并在 Python 2.7.9 中利用剩余的 146 个操作码中的 5 个来表示合并后的 OP_{new}。

Python 3 在 Python 2 的基础上对某些语法和模块进行调整和改动,因为同样遵循操作码仅占 1 Byte 长度的特性和操作码解释执行的机制,所以操作码合并算法可以在 Python 3 上实现,如 Python 3.6.1 使用了 256 个操作码中的 119 个,只能利用剩余的 137 个操作码来定义新操作码。

在操作码仅占 1 Byte 长度的基础上进行操作码合并,会大大限制操作码合并的扩展空间。若要生成和使用更多的操作码,则必须增加操作码所占存储空间的大小(如将操作码大小全部设为 2 Byte 或将部分操作码设为 2 Byte),但该操作会因多次读取操作码而导致程序执行效率下降。

2.3 算法性能分析

设 Python 源码文件 file^x.py 在操作码合并前编译生成的字节码文件为 file^x.pyc,其中包含的操作码序列为 S^x,用 n^x 和 m^x 分别表示 S^x 中所含的操作码个数和参数个数;file^x.py 在操作码合并后编译生成的字节码文件为 file$^x_{new}$.pyc,其中包含的操作码序列为 S^x_{new},用 N 表示 Python 虚拟机规范定义的基础操作码个数,num 表示扩展的操作码总个数,$g^x(num)$ 表示 S^x 与 S^x_{new} 的长度之差,即因操作码合并导致 S^x 减小的长度,$h^x(num)$ 表示 S^x_{new} 包含的新操作码个数。

2.3.1 安全性

与操作码替换[18]相比,操作码合并扩展了 Python 的操作码集,将符合某些特征的高频操作码序列合并为一个新的操作码,不仅削弱了操作码的统计学特性,而且改变了操作码序列的长度和结构,因此,操作码合并的安全性比操作码替换高。理论上,如果有耐心编写合并过程和解释过程支持 OP_{new} 的生成与执行,可以对操作码序列中的任意多个操作码进行合并,共有 $total(N)=A_N^2+A_N^3+\cdots+A_N^N$ 种可能的合并方式,该数目远大于指数函数 2^N(在 Python 2.7.9 中 $N=110$)。攻击者要确定新操作码序列中新操作码的个数,该过程的复杂度为指数函数,因此,操作码合并的安全性足够高。但是,一方面,由于一些操作码子序列的排列根本不会出现,出现的操作码子序列中若操作码处于不同的基本块,也不能合并,因此,实际中达不到该理论的可能合并方式;另一方面,操作码合并利用了操作码仅占1 Byte的特性,因此,仅能使用 256 个操作码中剩余的 146 个来定义新操作码。式(1)所示为操作码合并的强度,即利用穷举攻击从 S^x_{new} 恢复出 S^x 需要穷举的操作码序列个数。

$$O^x=\min(256^{n^x+2m^x},f) \tag{1}$$

其中,$f=C^{g^x(num)}_{n^x+2m^x-g^x(num)}\times N^2$,$C^{g^x(num)}_{n^x+2m^x-g^x(num)}$ 表示从长度为 $n^x+2m^x-g^x(num)$ 的 S^2_{new} 中确定 $g^x(num)$ 哪些字节是新操作码需要尝试的次数,N^2 表示在确定某个字节是新操作码的前提下把它复原为原来 2 个字节(实验中将原 2 个字节合并为 1 个字节)的操作码时需要尝试的次数,$1\leqslant num\leqslant 146$。

2.3.2 执行效率

当 Python 虚拟机执行字节码文件时,每次只能读取一个操作码和一个参数,并调用相应的解释执行过程。操作码合并前后 Python 虚拟机执行字节码文件运行过程如图 5 所示。

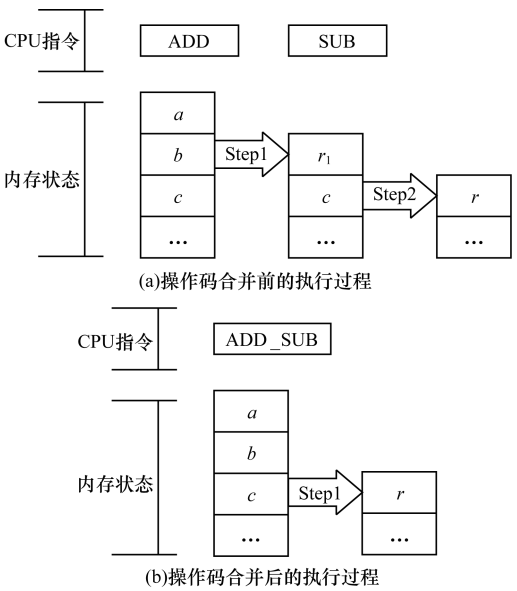

图 5 操作码合并前后执行过程对比

如图 5(a)所示,由于操作码序列中包含 ADD、SUB 列操作,因此需要读取 2 次操作码和参数以对栈顶的 2 个元素 a 和 b 出栈、求和,将结果 r_1 压栈,然后对栈顶元素 r_1 和 c 出栈、求差,将结果 r 再压栈。式(2)为操作码合并前 Python 虚拟机执行源码文件 file^x.py 的时间。

$$t_b^x=C^x+\sum_{i=1}^{n^x}(read_i^x+exec_i^x) \tag{2}$$

其中,C^x 表示编译文件 file^x.py 所花费的时间,$read_i^x$ 和 $exec_i^x$ 分别表示虚拟机读取和解释 S^x 中的每个操作码所花费的时间,$1\leqslant i\leqslant n^x$。

如图 5(b)所示,操作码合并后用新操作码 ADD_SUB代替原先的 2 个操作码,用新操作码一次完成栈顶 3 个元素 a、b、c 出栈,并将 $a+b-c$ 的结果 r 运算完之后再压栈。该操作虽然增加了编译源码文件所需的时间,但减少了冗余的读取操作及修改 Python 虚拟机内部状态的次数,且编译后的字节码文件在以后的运行过程中无需再次编译,从而提升了字节码文件的执行效率。

式(3)为操作码合并后 Python 虚拟机执行源码文件 file^x.py 的时间。

$$t_a^x = C^x + C_{\text{num}}^x + \sum_{i=1}^{n^x - g^x(num)} (\overline{read}_i^x + \overline{exec}_i^x) \quad (3)$$

其中,C_{num}^x表示合并操作码在编译阶段额外花费的时间,$\overline{read}_i^x$ 和 $\overline{exec}_i^x$ 分别表示操作码合并后每个操作码的读取与执行时间。

操作码合并算法的整体性能与合并前后的字节码文件执行时间有关。式(4)为操作码合并前后字节码文件执行时间的变化率。

$$t_\Delta^x = \frac{(t_a^x - (C^x + C_{\text{num}}^x)) - (t_b^x - C^x)}{(t_b^x - C^x)} \times 100\% = \left(\frac{(t_a^x - (C^x + C_{\text{num}}^x))}{(t_b^x - C^x)} - 1 \right) \times 100\% = \left(\frac{\sum_{i=1}^{n^x - g^x(num)} (\overline{read}_i^x + \overline{exec}_i^x)}{\sum_{i=1}^{n^x} (read_i^x + exec_i^x)} - 1 \right) \times 100\% \quad (4)$$

由式(4)可以看出,在操作码合并前字节码文件执行时间恒定的情况下,合并后的执行效率取决于新操作码序列的执行时间,而新操作码序列的执行时间与操作码序列 S_{new}^x 中操作码的数量成反比,与 S_{new}^x 中每个操作码的执行时间成正比,因此,通过合并操作码来减少 S_{new}^x 中操作码的数量或缩短每个操作码的解释执行时间,是提高程序运行速度的重要途径。虽然在不影响操作码序列执行结果的前提下,很难通过缩短操作码的执行时间来提高执行效率,但是可以通过操作码合并来实现执行效率的提高。本文正是依此在实现防逆转的同时提高字节码文件的执行效率。

2.3.3　存储空间

操作码合并将操作码序列 S^x 中连续出现的多个操作码合并为一个新操作码,在不减少 S^x 中参数个数的同时减少了操作码的个数,从而缩短了操作码序列的长度,如原本需占用($n^x + 2m^x$) Byte 的操作码序列,现只需占用($n^x + 2m^x - g^x(num)$) Byte 即可,字节码文件的大小缩小了($g^x(num)$) Byte。在需要存储和导入大量模块时,该方法有利于节约计算机磁盘和内存。同时,基于操作码合并生成的字节码文件需要新的解释器来解释执行,这就需要客户安装指定的 Python 解释器。虽然在首次使用时会给客户带来一些不便(以后无需再次安装),但该过程也是为了确保程序的安全性。

3　实验结果与分析

3.1　实验环境

本文基于 Python 2.7.9 版本,实验测试平台为 Intel Xeon E5-2690,主频 2.6 GHz,252 GB 内存,操作系统为 CentOS 6.7 64 位,实验对象为表 1 中的 5 对操作码。采用 microbenchmark[20] 来对新生成运行环境的防逆转效果、执行效率、文件大小进行检验。microbenchmark 包含数字运算、字符处理等多个方面的 Python 源码文件,是一个较好的测试集。

3.2　结果分析

3.2.1　安全性

本文通过操作码合并改变了原操作码序列的结构,对原操作序列信息进行了较好的隐藏。传统的反编译工具 uncompile2、Decompyle ++ 等无法识别合并后的操作码序列结构,且无法理解新操作码所包含的语义信息,因此,无法对新的字节码文件进行反编译。

3.2.2　执行效率

操作码合并算法用一个新操作码来完成原来多个操作码包含的任务,减少了冗余的读取操作与修改虚拟机状态的次数,从而提升了执行效率。表 2 是操作码合并前后 microbenchmark 下部分测试脚本编译后的文件运行时间及其变化率数据,其中,$t_\Delta = \frac{t' - t}{t} \times 100\%$。

表 2　操作码合并前后文件执行时间比较

文件名	合并前 t /ms	合并后 t' /ms	变化率 t_Δ /%
test.pyc	606	513	-15.35
pydigits.pyc	621	524	-15.61
attribute_lookup.pyc	1 306	1 210	-7.56
attrs.pyc	16 505	16 375	-0.79
contains_ubench.pyc	10 953	10 513	-4.02

从表 2 中的实验数据可以看出,使用操作码合并算法在参数个数不变的前提下,减少了字节码文件中操作码序列的操作码个数,即减少了 CPU 读取操作码的次数和多个操作码执行之间的跳转次数,从而缩短了程序的执行时间。

从表 2 实验数据中还可以看出,操作码合并后 test.pyc 和 pydigits.pyc 文件执行时间较之前有明显的减少,通过分析发现其原因为,test.pyc 操作码序列中包含的操作码个数较少,只需要合并若干个操作码就可以使其个数有明显的减少;合并前的 pydigits.pyc 操作码包含在循环中,因为循环中的操作码被多次执行,每次执行的时间较合并前都有所减少,所以使得整个操作码序列的执行时间有明显减少。

3.2.3　存储空间

操作码合并算法用一个新操作码代替原来的 2 个或更多的操作码,从而生成较短的操作码序列,较短的操作码序列使得新生成的字节码文件占用的内存空间和磁盘空间较小。表 3 是操作码合并前后

microbenchmark 下部分测试脚本编译后的文件大小及其变化率数据,其中,$s_\Delta = \frac{s' - s}{s} \times 100\%$ 。

表 3 操作码合并前后文件大小比较

文件名	合并前 s /Byte	合并后 s' /Byte	变化率 s_Δ /%
test. pyc	1 508	1 498	-0.66
namedtuple_ubench. pyc	408	466	-2.91
dict_globals_ubench. pyc	538	526	-2.23
getattrfunc_ubench. pyc	673	659	-2.08
exceptions_ubench. pyc	577	571	-1.04
re_finditer_bench. pyc	399	395	-1.00

从表 3 可以看出,操作码合并后所有字节码文件的大小都有所减小。

4 结束语

本文提出基于操作码合并的 Python 字节码文件防逆转算法,与文献[21]基于隐藏参数的方法相同,都是通过扩充操作码集来改变操作码序列的内容和结构,最终达到防逆转的目的。虽然目前两者都受限于操作码仅占 1 Byte 长度的特性,存在扩展空间不足的问题,但本文算法建立在对操作码序列特征统计分析的基础上,其不会因程序运行时参数的变化而产生影响,通用性较强。实验结果表明,在字节码文件执行效率和文件大小两方面,本文算法在操作码合并前后都有明显提升。为进一步提升字节码文件的安全性和算法的适用性,下一步将考虑通过抽象语法树的变换和优化,对多基本块之间的操作码进行合并操作。

参考文献

[1] Mysterie. A Python 2.7 byte-code decompiler[EB/OL]. [2017-03-10]. https://github.com/wibiti/uncompyle2.

[2] Niwit. Decompyle-Python decompiler[EB/OL]. [2017-03-10]. https://sourceforge.net/projects/decompyle/?source=typ_redirect.

[3] Extremecoders. Python 1.0-3.4 bytecode decompiler[EB/OL]. [2017-03-10]. https://sourceforge.net/projects/easypythondecompiler/.

[4] 蒋 华,刘 勇,王 鑫. 基于控制流的代码混淆技术研究[J]. 计算机应用研究,2013,30(3):897-899.

[5] 杨 乐,周强强,薛锦云. 基于垃圾代码的控制流混淆算法[J]. 计算机工程,2011,37(12):23-25.

[6] KUZURIN N,SHOKUROV A,VARNOVSKY N,et al. On the concept of software obfuscation in computer security[C]//Proceedings of International Conference on Information Security. Washington D. C.,USA:IEEE Press,2007:281-298.

[7] 徐海银,雷植洲,李 丹. 代码混淆技术研究[J]. 计算机与数字工程,2007,35(10):4-7.

[8] 鲍福良,彭俊艳,方志刚. Java 类文件保护方法综述[J]. 计算机系统应用,2007,16(6):124-126.

[9] Py2exe:A Python distutils extension which converts Python scripts into executable windows programs[EB/OL]. [2017-03-10]. http://www.py2exe.org/.

[10] 陈明奇,钮心忻. 数字水印的研究进展和应用[J]. 通信学报,2001,22(5):71-79.

[11] 孙圣和,陆哲明. 数字水印处理技术[J]. 电子学报,2000,28(8):85-90.

[12] 陈 晗,赵轶群,缪亚波. Java 字节码的水印嵌入[J]. 计算机应用,2003,23(9):96-98.

[13] COLLBERG C S,THOMBORSON C. Watermarking tamper-proofing and obfuscation[J]. IEEE Transactions on Software Engineering,2000,28(8):735-746.

[14] HAMILTON J,DANICIC S. An evaluation of static Java bytecode watermarking[EB/OL]. [2017-02-25]. https://jameshamilton.eu/sites/default/files/JavaBytecodeWatermarkingSurvey.pdf.

[15] KUMAR K,KEHAR V,KAUR P. An evaluation of dynamic Java bytecode software watermarking algorithms[J]. International Journal of Security and Its Applications,2016,10(7):147-156.

[16] KHOLIA D,WEGRZYN P. Looking inside the (Drop) box[EB/OL]. [2017-03-01]. https://www.usenix.org/system/files/conference/woot13/woot13-kholia.pdf.

[17] J. C. 斯普拉德林. 通过操作码随机化的安全:CN 102592082 A[P]. 2012-07-18.

[18] Omab. Python-obfuscation[EB/OL]. [2017-03-10]. https://github.com/citrusbyte/python-obfuscation.

[19] STALLINGS M. Cryptography and network security[M]. 王章宜,杨 敏,杜瑞颖,等,译. 北京:电子工业出版社,2012.

[20] MODZELEWSKI K. Microbenchmarks[EB/OL]. [2017-03-10]. https://github.com/dropbox/pyston/tree/master/microbenchmarks.

[21] GUELTON S. Building an obfuscated Python interpreter: we need more opcodes[EB/OL]. [2017-03-10]. https://blog.quarkslab.com/building-an-obfuscated-python-interpreter-we-need-more-opcodes.html.

编辑 吴云芳