



Y3226514

中国科学技术大学

University of Science and Technology of China

# 硕士学位论文



论文题目 基于操作码的 Python 程序防逆转算法

研究与实现

作者姓名 王小强

学科专业 计算机系统结构

导师姓名 顾乃杰 教授

完成时间 二〇一七年三月

中国科学技术大学

# 硕士学位论文



## 基于操作码的 Python 程序防逆转算 法研究与实现

作者姓名： 王小强

学科专业：计算机系统结构

导师姓名： 顾乃杰 教授

完成时间：二〇一七年三月



University of Science and Technology of China  
A dissertation for master's degree



# Research and Implementation of Anti-reversal Method Based on Python Opcode

Author's Name: Xiaoqiang Wang  
Speciality: Computer Architecture  
Supervisor: Prof. Naijie Gu  
Finished Time: March, 2017

## 中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名： 王心强

签字日期： 2017.5.26

## 中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☒ 公开    ☐ 保密 (\_\_\_\_ 年)

作者签名： 王心强

导师签名： 张明

签字日期： 2017.5.26

签字日期： 2017.5.26

## 摘 要

Python 编程语言自 20 世纪 90 年诞生至今,得益于其简单易学、语法简洁清晰、可扩展性强、支持面向对象等诸多优点,已被广泛的应用于系统管理任务和 Web 编程等诸多领域。但使用 Python 编程语言编写的源码文件(.py)编译生成的字节码文件(Bytecode file, .pyc)很容易被逆向工具反编译,这不仅是会侵害开发人员的知识产权和经济利益,而且具有严重的安全隐患。于此同时现有的代码混淆技术、文件加密技术、本地编译技术、数字水印技术等防逆转方法存在安全性不足、容易造成字节码文件的执行效率下降、应用体积增加等问题。

为此本文围绕基于操作码替换与合并的 Python 字节码文件防逆转策略展开研究工作,本文的主要的研究内容和成果包括以下三个方面:

(1) 通过对 Python 运行框架和 Python 字节码文件编译、解释执行机制的分析,根据 Python 虚拟机对字节码文件中的操作码逐一进行解释执行的特性,将 Python 字节码文件的核心内容 `co_code` 域进行简化抽象,建立字节码文件的操作码序列模型与基本块模型。

(2) 针对现有的代码混淆技术和数字水印技术安全性不足的问题,本文以字节码文件中的操作码序列为基础,结合单表替换密码,设计出了一种适用于 Python 字节码文件的操作码的操作码替换策略。该策略通过操作码替换来改变操作码序列中操作码的值来达到改变操作码序列内容和防逆转的目的。最后对操作码替换策略在 Python 2.7.9 中予以实现,并根据单表替换密码的特性,利用操作码的统计学规律,评估操作码替换策略的安全性。

(3) 针对文件加密技术易对字节码文件的执行效率造成影响和本地编译技术造成目标程序体积增加的问题,本文设计出一种操作码合并策略。该策略以字节码文件中的操作码序列的基本块为基础,利用窥孔优化技术将处于同一个基本块中连续出现的多个操作码进行合并,并使用新操作码来代替原来操作码序列中连续出现的多个操作码。通过操作码合并大大缩短了操作码序列的长度,改变了操作码序列的结构和内容,最终达到防逆转的目的。最后对操作码合并策略在 Python 2.7.9 中予以实现,并对操作码合并策略产生的字节码文件的安全性、执行效率、以及文件大小进行评估与实验。

**关键词:** Python 字节码文件 虚拟机 防逆转 操作码替换 操作码合并



## ABSTRACT

Python programming language since its birth in the 20th century, 90 years, thanks to its simple and easy to learn, simple and concise syntax, scalability, support for object-oriented and many other advantages, has been widely used in systems management tasks and Web programming and many other areas. However the bytecode file (.pyc) compiled from Python source code file (.py), which written in Python programming language, is easily decompiled by the reverse tool, that will not only infringes the developer's intellectual property and economic interests, but also has serious security hidden danger. But at the same time, the code confusing technology, file encryption technology, local compilation technology, digital watermarking technology and other anti-reversal method is lack of security, easily resulting in the decrease of byte code file's efficiency, or application's volume increases and other issues.

So this paper working on anti-reversal strategy the of Python bytecode files's operate code replacement and merging, The main contents and achievements of this paper include the following three aspects:

(1) By the analysis of Python runtime framework and the Python bytecode file's compilation execution, base on the characteristics of Python Virtual Machine work on bytecode file's opcodes one by one, simplify the bytecode file's co\_code domain, set up the opcode sequence and it's basic block model of the bytecode file.

(2) Focus on the lack of security of existing code obfuscation technology and digital watermarking technology, This article based on the opcode sequence in the bytecode file, combined with the single table replacement encryption, design an opcode replacement strategy for Python opcodes. The strategy is replace the opcodes in the opcode sequence to achieve the purpose of change the contents opcode sequence contents and achieve the purpose of anti-reverse. At Last, implement the opcode replacement strategy in Python 2.7.9 and according to the single table replacement encryption's characteristic, use the opcodes' statistical law, evaluate the it's security.

(3) Aim at the efficiency problem caused by file encryption technology and storage problem of the native compile technology, this article design a opcode merge strategy. This strategy base on the opcode sequence in the bytecode file, take peephole optimization technique to merge a serious of opcodes who located in same basic block to a expanded value. The opcode mergeing strategy not only greatly decrease the length opcodes sequence and change it's structure and content but also achieve the purpose of

reverse . At last, the opcode merging strategy is implement in Python 2.7.9, and evaluate and experimete on the security, execution efficiency, and file size of the bytecode file generated by the strategy .

**Key Words:** Python, Bytecodefile, Virtual Machine, Anti-reversal Engine, Opcode Replacement, Opcode Merging



# 目 录

摘要 .....	I
Abstract .....	III
第 1 章 绪论 .....	1
1.1 研究背景 .....	1
1.1.1 脚本语言和 Python 的发展 .....	1
1.1.2 Python 程序和字节码文件 .....	3
1.2 操作码替换与合并的研究意义 .....	5
1.3 论文的主要工作 .....	6
1.4 论文结构 .....	7
第 2 章 Python 总体架构与传统的防逆转方式 .....	9
2.1 Python 总体架构 .....	9
2.1.1 Python 模块 .....	9
2.1.2 Python 运行时系统 .....	10
2.1.3 Python 虚拟机 .....	11
2.2 传统的 Python 字节码文件防逆转方式 .....	12
2.2.1 代码混淆技术 .....	12
2.2.2 文件加密技术 .....	16
2.2.3 本地编译技术 .....	20
2.2.4 数字水印技术 .....	21
2.3 本章小结 .....	25
第 3 章 Python 字节码文件与其解释执行机制 .....	27
3.1 Python 字节码文件 .....	27
3.1.1 PyCodeObject .....	27
3.1.2 co_code 域 .....	29
3.2 Python 字节码文件的解释执行 .....	30
3.3 本章小结 .....	34
第 4 章 操作码替换与合并的设计 .....	35
4.1 操作码序列 .....	35
4.2 操作码替换的设计 .....	37
4.2.1 寻找操作码编译和解释执行规律 .....	38

4.2.2 打乱操作码映射关系 .....	39
4.3 操作码合并的设计 .....	40
4.3.1 提取操作码信息 .....	41
4.3.2 频率统计 .....	41
4.3.3 提取基本块信息 .....	42
4.3.4 扩充操作码集 .....	42
4.4 操作码合并示例 .....	43
4.4.1 获取段代码的操作码信息 .....	43
4.4.2 频率统计 .....	44
4.4.3 获取基本块信息 .....	44
4.4.4 扩充操作码集 .....	45
4.4.5 操作码序列合并前后比较 .....	46
4.5 本章小结 .....	46
第 5 章 操作码替换与合并方式的性能评测 .....	47
5.1 操作码替换性能评测 .....	47
5.1.1 操作码替换安全性分析 .....	47
5.1.2 操作码替换实验检验 .....	48
5.2 操作码合并性能评测 .....	50
5.2.1 安全性分析 .....	50
5.2.2 执行效率分析 .....	51
5.2.3 存储空间分析 .....	53
5.2.4 操作码合并实验 .....	53
5.3 本章小结 .....	55
第 6 章 全文总结 .....	57
6.1 研究工作与成果 .....	57
6.2 下一步工作 .....	58
参考文献 .....	59
致谢 .....	63
在读期间发表的学术论文与取得的研究成果 .....	65

## 图目录

2.1 Python 总体架构 .....	9
2.2 Python 模块示例代码 .....	10
2.3 Python 内存管理机制的层次结构 .....	11
2.4 代码混淆的分类 .....	13
2.5 混淆前的代码 .....	14
2.6 混淆后的代码 .....	14
2.7 混淆变换的弹性度量 .....	16
2.8 对称密码模型 .....	17
2.9 英文字母相对频率 .....	18
2.10 非对称密码模型 .....	19
2.11 py2exe 示例 .....	21
2.12 数字水印算法流程图 .....	23
2.13 哑函数的嵌入流程 .....	24
2.14 哑函数调用示例 .....	24
3.1 PyCodeObject 类型的定义 .....	28
3.2 PyCodeObject 中各个域的含义 .....	29
3.3 co_code 的逻辑结构示意图 .....	30
3.4 PyObject_Header 的逻辑结构示意图 .....	30
3.5 Python 遍历字节码指令序列状态示例 .....	31
3.6 Python 虚拟机解释执行逻辑示意 .....	31
3.7 Python 虚拟机中几个重要的宏定义 .....	32

---

3.8 Python 虚拟机状态示例代码 .....	32
3.9 Python 虚拟机执行流程图 .....	33
4.1 Python 操作码定义示例 .....	35
4.2 操作码序列示意图 .....	36
4.3 基本块信息示意图 .....	37
4.4 操作码替换示意图 .....	38
4.5 操作码替换设计 .....	38
4.6 操作码替换的分类 .....	38
4.7 操作码替换前后操作码序列变化示意 .....	39
4.8 操作码替换示例 .....	40
4.9 操作码合并算法流程 .....	40
4.10 操作码合并示意图 .....	43
4.11 test.py 示例代码 .....	44
4.12 频率较高的操作码序列 1 .....	44
4.13 频率较高的操作码序列 2 .....	44
4.14 操作码合并实例图 .....	45
5.1 操作码相对使用频率曲线 .....	48
5.2 破译目标操作码相对使用频率曲线 .....	49
5.3 相对频率比较曲线 .....	49
5.4 操作码合并前后执行过程对比图 .....	51
5.5 反编译失败示例 .....	54

表目录

4.1 部分长度为 2 的操作码子序列以及出现次数 ..... 42

5.1 操作码合并前后字节码文件执行时间比较 ..... 54

5.2 操作码合并前后字节码文件大小比较 ..... 55



# 算法索引





## 第 1 章 绪论

### 1.1 研究背景

脚本编程语言快速开发、容易部署等优点为其发展和普及提供了有利的条件。Python 编程语言作为诸多脚本语言之一，自诞生以来已被广泛应用于系统管理任务和 Web 编程等诸多领域，然而 Python 源码文件编译生成的字节码文件很容易被逆向工具反编译。因此，本文将探索如何在保证 Python 字节码文件执行结果正确性与执行效率的前提下，为 Python 字节码文件提供一种通用的防逆转措施。

#### 1.1.1 脚本语言和 Python 的发展

随着脚本编程语言的发展和普及，越来越多的个人和企业在进行编程或软件开发时都开始选择一种脚本编程语言作为主要开发语言，脚本编程语言是一种为了缩短传统的“编写（edit）、编译（compile）、链接（link）、运行（run）”过程而创建的计算机编程语言，与传统的 C/C++ 等编程语言开发的程序一次编译永久运行不同，使用脚本语言开发的程序一般是依赖于虚拟机（或解释器）解释运行的。以简单的方式快速完成某些复杂的事情是创建脚本语言的重要原则，所以脚本语言通常与 C/C++ 之类的系统编程语言相比有易学易用、快速开发、容易部署等优点，因此脚本语言在作业批处理、游戏编程、网页编程、工业控制等领域都有广泛的使用。使用脚本编程语言编写的源程序称为脚本，通常以文本形式保存，只是在被调用时逐行解释（或翻译）执行，所以这也导致了脚本语言在执行时速度可能很慢、而且运行时更耗内存等缺点，尽管这样也难敌脚本语言带来的开发快速、使用方便、节约开发成本等优点。

脚本语言根据其使用场景可以分为以下几个大类<sup>[1]</sup>：（1）shell 语言和工程控制语言，这类脚本语言通常负责系统的启动和行为动作，即多用于自动化工程控制，此外如 MS-DOS COMMAND.COM 和 Unix Shell 等命令行界面通常是这类脚本语言的解释器，这类脚本语言常见的有 4DOS、sh、bash 等；（2）在用户和图形界面、菜单、按钮等之间互动的 GUI 脚本，这类脚本经常用于自动化重复性动作，或设置一个标准状态。理论上可以使用 GUI 脚本控制运行于基于 GUI 的计算机上的所有应用程序，但实际上 GUI 脚本是否被支持还要看应用程序和操作系统本身，常见的 GUI 脚本有 AutoHotKey、Autolt、Expect；（3）大型的应用程序根据用户需求而定制的惯用脚本语言，这类脚本语言通常是为一个单独的应用程序所设计，虽然它们与一些通用语言的语言（如 QuakeC, modeled after）

有些相似，但它们都有自定义的功能，常见的应用程序订制的脚本语言有 Action Code Script、Matlab Embedded Language、Visual Basic for Applications 等等；（4）专门处理互联网通信，用于提供 Web 页面的自定义功能使网页浏览器作为用户界面的 Web 编程脚本，常见的 Web 编程脚本有 ColdFusion、IPTSCRAE、Lasso 等；（5）用于处理基于文本记录的文本处理脚本，这类脚本语言现在已经成了全面成熟的语言，如 Unix 环境下的 awk、perl、sed 等；（6）从一门脚本语言发展而来，成为更通用的编程语言通用动态语言，由于它们已经用于编写应用程序，但使用这类脚本语言开发的程序仍有“解释执行，内存管理，动态”等特性，所以这类语言仍被称为脚本语言，常见的 Python、Ruby、PHP、perl 等；（7）被设计为通过嵌入应用程序来替换应用程序功能的定制的少数脚本语言称为扩展可嵌入语言，这类脚本语言优点在于可以在应用程序之间传递一些数据，程序开发者（如使用 C 等其它系统语言）嵌入脚本语言可以控制应用程序的 hook，则后期可以通过编写脚本语言程序来控制程序的运行，常见的可嵌入语言有 ch（C/C++ interpret）、Tcl 还有被用于网页浏览器内的主要编程语言 JavaScript 等；（8）还有一些脚本语言不属于以上几类，例如 BeanShell（scripting for Java）、CobolScript 等等。

Python<sup>[2]</sup> 是由荷兰人 Guido van Rossum 1989 年发明、1991 发布第一个公开发行版的通用脚本语言语言，因其语法简洁清晰、简单易学、免费开源、运行速度快（Python 底层使用 C 实现，许多标准库和第三方库也都是使用 C 开发）、并且支持面向对象等特性使得 Python 自其诞生以来就收到了广发的欢迎和使用，TIOBE 在 2011 年 1 月将 Python 编程语言评为 2010 年度语言。Python 经常被用于处理系统管理任务和网络程序的编写，例如内如管理系统 Plone、社交新闻站点 Reddit、文件分享应用 Dropbox 等都是使用 Python 编程语言开发，它也非常适合完成各种高级任务，并且 Python 虚拟机本身几乎可以在所有的作业系统中运行，例如使用诸如 py2exe、PyInstaller 之类的工具可以将 Python 编写的源代码转换成可以脱离 Python 解释器运行的程序。

Python 支持面向对象和丰富而强大的库使得 Python 在各个层面都有广泛的应用，例如：

（1）Python 编程语言经常被用于编写各种各样的网络爬虫程序、Web 程序和服务器软件开发，这都得益于 Python 在支持各种网络协议方面的优势，如使用 Python 语言编写的 Web 服务器 Gunicorn 同时也能够运行 Python 语言编写的 Web 程序，还有 Pyramid、Django、Tornado、TurboGears、web2py、Zope、Flask 等一些 Web 框架，可以让程序员轻松地开发和管理复杂的 Web 程序；

（2）使用 Tkinter 库能够支持简单的 GUI 开发，使用 wxPython 或者 PyQt 等 GUI 包可以开发跨平台的桌面软件，并且使用这些库开发的桌面软件运行速度

快、可以契合用户的桌面环境；

(3) 在很多操作系统中 Python 编译器和解释器都是作为操作系统预装组件的一部分在操作系统安装时也一同安装和配置，例如 MacOS X 系统和大多数 Linux 操作系统（如 Ubuntu、Centos 等）都预装了 Python 编译器和解释器，用户可以在 shell 中使用命令来启动 Python 解释器或运行 Python 程序，或者也可以直接在操作系统终端下编写和运行 Python 程序。在 RPM 系列 Linux 发行版中，有一些系统组件就是用 Python 编写的，例如 CentOS 下的包管理器 yum，Gentoo Linux 下使用 Python 来编写软件包管理系统 Portage。

除此之外，还可以使用 Numpy、SciPy、Matplotlib 等库编写科学计算程序，Python 可以轻松地使用其他语言（尤其是 C/C++ 语言）编写的模块联结接在一起的特性使得程序员可以使用 Python 进行繁琐的游戏逻辑编程，而在编写图形显示等对执行效率要求较高的模块时则可以使用 C++/C++ 语言完成，因此 Python 也常被称为“胶水语言”。

### 1.1.2 Python 程序和字节码文件

使用 Python 编程语言编写的源码程序文件（.py）可以直接在 Python 虚拟机（Python Virtual Machine）上解释运行，或者经过 Python 编译模块（py\_compile 或 compileall）将使用 Python 编程语言编写的源码文件（.py）编译为 Python 字节码（Bytecode file）文件（.pyc 或 .pyo）之后再到虚拟机上运行。当然 Python 虚拟机在直接执行源码文件（.py）时，同样利用了其编译有编译模块将源码文件（.py）编译为字节码文件（.pyc），只是 Python 将编译环节生成的中间量（.pyc 字节码文件）保存在了内存中，直接读取执行，而没有显式地写到磁盘上，等到程序执行完毕之后直接释放掉了内存中字节码文件所占用的空间而已。Python 字节码文件（.pyc 或 .pyo）的实质是 Python 源码源码的文件名、路径、常量、所有的符号、操作码序列等信息在编译之后构成的序列，字节码文件（.pyc 或 .pyo）与传统的针对特定处理器和操作系统的二进制文件（.exe 或者 .out）相比，是针对 Python 虚拟机（Python Virtual Machine）的具有特定的结构和特征的文件，保留了 Python 源码文件的全部信息，具有可以跨平台使用的特性，所以 Python 软件开发者，常常是将使用 Python 编程语言开发的软件编译之后的字节码文件（.pyc）发布给客户使用，客户在拿到字节码文件之后（.pyc）只需要在自己的环境中安装特定版本的 Python 运行环境，即可使用 Python 编程语言开发的软件。

于此同时软件逆向工程已经成为一种流行的软件分析技术，所以不免有一些 Python 软件使用者或攻击者为了避免支付合理的软件版权费用，而利用 Python 字节码文件为了跨平台性而保留了源码文件全部信息的特性，费劲心思来分析

(破解)使用 Python 编程语言开发的软件 (.pyc)。所以正是因为 Python 字节码文件这种的特殊结构和特征,导致了字节码文件极易被攻击者反编译出其中的源码,造成软件知识产权被侵犯、关键数据结构、重要算法、关键业务逻辑的暴露,这不仅仅会侵犯开发者知识产权和给开发者带来巨大的经济损失,而且具有严重的安全隐患。编译是将高级语言转换为低级语言(如机器码或字节码)的行为,而反编译是相反的是将低级语言转换为高级语言的行为,与反编译机器代码构成的二进制文件相比的反编译字节码文件就显得更容易、且成本低廉。例如使用自由和开源的软件 uncompile2<sup>[3]</sup>、Decompyle++<sup>[4]</sup>、Easy Python Decompiler<sup>[5]</sup>等工具就可以轻易将 Python 字节码文件反编译为高质量的源码文件。

鉴于 Python 字节码文件极易被攻击者反编译出其中源码信息的现状,为了更好地保护 Python 软件开发者的知识产权和利益,对使用 Python 编程语言开发的软件采取适当的手段予以保护显得十分必要。目前有很多公司都对使用 Python 编程语言开发的软件都采取一定的技术手段予以保护,意图干扰或阻止攻击者从字节码文件中提取出源码信息,但这些字节码文件保护技术都有一定的不足与缺陷,例如:

(1) 代码混淆技术<sup>[6]</sup>:代码混淆技术是一种保留程序语义的程序变换技术,它只是将一个程序从一种形式形势转换为能够妨碍攻击者理解其中的算法和数据结构,或者能够阻止攻击者从程序文本中提取有价值的信息的另一种形式<sup>[7]</sup>,只是使得程序在缺少注释的情况下难以阅读<sup>[6]</sup>。但混淆技术生成的字节码文件仍然遵照原来的文件格式和指令集,不需要其他任何辅助手段或模块,生成字节码文件仍可以跨平台使用,这是代码混淆技术的一大优势。一方面代码混淆技术不能处理多模块之间导入的变量或对象名称,另一方面使用代码混淆技术会给程序执行效率带来较大的影响,再者代码混淆技术并没有改变 Python 字节码文件的指令集与结构,所以代码混淆技术并不能 Python 程序提供足够的保护,也不是一个理想的保护方法。

(2) 字节码文件加密技术<sup>[8]</sup>:加密技术无论是对软件可执行程序或是字节码文件的保护都是一种很有效的保护方式,但加密方案对 Python 字节码文件的保护强度依赖于所采取的加密方法的复杂度和密钥的长度。该方案也存在自己的缺陷:首先果解密算法太过复杂,则会对 Python 虚拟机的整体性能带来较大的影响;其次需要妥善的存储和保管密钥,如果密钥被他人所获取,则前面所做的工作就前功尽弃了;再者抛开以上两点缺点,由于载入内存的字节码文件是解密后的文件,容易被他人采用直接读取内存的方式获得。

(3) 本地编译技术<sup>[9]</sup>:本地编译技术是一种将 Python 脚本程序和解释器一起编译为平台相关的程序的技术,例如利用该技术可以将 Python 源码 (.py) 或字节码文 (.pyc 或.pyo) 编译为 Windows 平台上的可直接运行的二进制文件 (.exe)。

该方案表面上能够对 Python 字节码文件起到一定的保护作用，但实际使用中即使为了发布一个体积很小的软件，也要同时将 Python 解释器对应的动态链接库文件（.dll）和诸多依赖的 Python 库文件同时再发布一遍，大大增加了软件占用的体积，也违背了软件模块复用的原则。

（4）数字水印技术：数字水印是指为了标识数字产品所有权信息、或者为了保护数字产品知识产权、版权而在数字产品中植入的具有一定特征的可见或不可见的数字信息，通常其不仅可以用来标识作者、所有者、发行者、使用者等信息<sup>[6][10]</sup>，而且并不影响宿主数据的可用性。虽然使用水印技术并不能阻止字节码文件被反编译，但是能够阻止数字产品被偷窃，或者当偷窃发生时为软件开发人员提供数字产品的拥有权证明。虽然理论上的水印能够作为软件所有权的有力凭证，但实际中都很容易通过混淆变换和优化等措施从软件中移除<sup>[11][12]</sup>，所以水印技术并不能为 Java、Python 等字节码程序提供有效的保护。

## 1.2 操作码替换与合并的研究意义

众多的 Python 字节码文件防逆转方式的提出和采用的目的是为了改变 Python 字节码文件易遭攻击者逆转的现状，从而保护开发者的知识产权和利益。操作码替换与合并策略的提出是为了扩充 Python 字节码文件防逆转可以采取的技术手段和方式，并且改变传统的防逆转方式安全性不足，易给 Python 字节码文件执行效率、给应用体积带来影响等问题。本文提出的基于操作码替换与合并的字节码文件防逆转方案，其设计不仅适用于 Python 语言，也适用于任何基于虚拟机解释执行的编程语言。本文之所以研究基于操作码合并的 Python 字节码文件防逆转方式，其原因有三点：

（1）有越来越多的个人和机构都认识到 Python 编程语言语法简介清晰、易于理解、开发速度快等优点，并在学习和工作中使用 Python 编程语言作为自己编程或软件开发的的首选语言。

（2）使用 Python 编程语言开发的软件极易被攻击者利用工具反编译出其中的源码信息，从而侵犯软件开发者知识产权，给软件开发者带来巨大的经济损失。

（3）传统的 Python 字节码程序防逆转方式虽然能够给使用 Python 编写的应用程序带来一定的保护，但大多都存在安全性不足、会给 Python 程序的执行效率 and 应用程序体积带来负面影响等问题。

基于 Python 字节码文件使用广泛和易遭逆转的背景，对传统的 Python 程序防逆转方式进行改进，或者提出新的 Python 字节码文件防逆转方式已经称为了 Python 字节码文件防逆转的新需求，新提出的防逆转方式首先要不影响 Python

字节码文件的执行结果的正确性；其次要能为 Python 字节码文件提供一定的保护以阻止攻击者反编译出其中的源码信息；此外在达到以上两点要求的同时力争提升目标程序执行效率与减小应用体积。所以 Python 字节码程序防逆转方面值得研究的一个问题。

### 1.3 论文的主要工作

根据 1.2 节所述的意义，本论文对 Python 字节码文件易遭攻击者反编译的问题和目前存在的防逆转方式的现状进行了深入研究，在对 Python 虚拟机架构和 Python 字节码文件编译、优化、执行机制进行了深入分析的基础上，根据 Python 虚拟机解释执行字节码文件的特点，提出并实现了基于操作码替换与合并的 Python 字节码文件防逆转方式。重新映射了 Python 操作码与操作码值之间的对应关系，并扩充了 Python 操作码集，重新设计了 Python 字节码文件的操作码序列结构，并利用理论分析和实验检验了操作码替换与合并方案的所能达到的安全性、对字节码文件执行效率的影响、对字节码文件占用内存和磁盘的影响。

本论文对操作码替换与合并方式的设计和实现基于 Python-2.7.9 版本，同时该方法也适用于其它 Python 版本。主要研究内容包括以下三个方面：

(1) 通过对 Python 运行框架和 Python 字节码文件编译生成、执行执行机制的分析，根据 Python 虚拟机对字节码文件中的操作码逐一进行解释执行的特点，将 Python 字节码文件的核心内容 `co_code` 域的执行流程进行简化抽象，建立字节码文件的操作码序列与其基本块模型。

(2) 针对现有的代码混淆技术和数字水印技术安全性不足的问题，本文以字节码文件中的操作码序列为基础，结合单表替换密码，在不改变 Python 字节码文件执行正确性的前提下，设计出了一种适用于 Python 操作码的操作码替换策略。该策略通过操作码替换来改变操作码序列中操作码的值来达到改变操作码序列内容，最终达到防逆转的目的。最后对操作码替换在 Python 2.7.9 中予以实现，并根据单表替换密码的特性，利用操作码的统计学特性，对操作码替换前后的字节码文件中的操作码频率进行统计、比较分析，评估操作码所替换策略的安全性进行与实验。

(3) 针对文件加密技术易对字节码文件的执行效率造成影响和本地编译技术目标程序体积增加的问题，本文设计出了操作码合并策略。该策略对 Python 操作码集进行扩充，在不改变 Python 字节码文件执行结果和逻辑的前提下，以字节码文件中的操作码序列的基本块为基础，利用窥孔优化技术将处于同一个基本块中连续出现的多个操作码进行合并，使用新的操作码来代替原来操作码

序列中连续出现的多个操作码。通过操作码合并改变了操作码序列的结构和内容，大大缩短了操作码序列的长度，达到防逆转的目的。最后对操作码合并并在 Python 2.7.9 中予以实现，并操作码合并策略产生的字节码文件的安全性、执行效率、以及文件大小进行评估与实验。

## 1.4 论文结构

本文主要介绍基于操作码替换与合并的 Python 字节码文件防逆转方式的设计与实现，整个文章共分六章，并且行文内容按照如下结构组织：

第一章主要介绍论文的研究背景，论文主要实现的内容、所采取的方法方法以及整篇文件的内容结构。

第二章主要介绍 Python 体系结构，以及传统的 Python 字节码文件防逆转方式和存在不足。

第三章详细是主要介绍了 Python 字节码文件结构、操作码序列的结构和含义、Python 字节码文件解释执行机制，并由此引入操作码合并方法。

第四章是整篇文章的核心，首先在第三章的基础上对 Python 字节码文件的核心内容 `co_code` 域进行简化抽象，建立字节码文件的操作码序列模型与基本块模型；其次对基于 Python 操作码替换的 Python 字节码程序防逆转方式设计与实现进行了详细讨论与实现；最后详细讨论了基于虚拟机解释执行的 Python 操作码合并的设计与实现，包括整个系统出发点和框架结构。

第五章对基于操作码替换与合并的 Python 字节码防逆转方式进行理论分析与实验检验，本文主要从安全性（防逆转效果）、程序执行效率、占用存储空间，三个方面进行分析。

第六章对全文进行概括总结，回顾了本文的主要研究内容和工作，同时对下一步工作作出展望。





## 第 2 章 Python 总体架构与传统的防逆转方式

在本章中将首先对 Python 的总体架构进行介绍，以期对 Python 的实现和运行有个宏观的认识，并为在第 3 章理解 Python 字节码文件和字节码文件的解释执机制打下基础。其次还要对对传统的字节码文件防逆转方式的研究现状进行介绍，来认识传统的防逆转方式存在的问题和不足。

### 2.1 Python 总体架构

Python 总体架构如图2.1所示，分为模块文件、编译器和解释器构成的 Python 核心以及运行时状态 3 个部分，为了方便理解 Python 字节码文件编译和执行原理，下面将对 Python 架构的这 3 个方面进行简要的介绍。

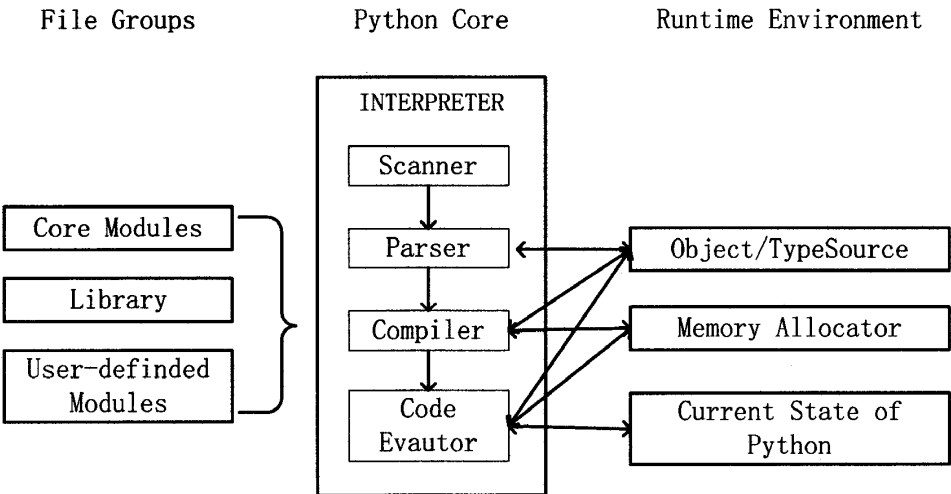


图 2.1 Python 总体架构

#### 2.1.1 Python 模块

在图2.1的左边是 Python 提供的大量的模块、库以及用户自定义的模块。Python 提供了广泛的标准库，这些模块包括用 C 语言实现的用以提供访问系统功能（如 I/O 等）的内建（built-in）模块。例如 Python 虚拟机在执行图2.2所示的代码中的 `import sys` 语句时，这个 `sys` 模块就是 Python 的内建模块；其次 Python 还为解决日常问题还提供了大量的用 Python 编程语言实现标准解法，例如图2.2中的 `os.getcwd()` 函数就是 Python 提供的用于获取当前工作目录的接口；当然开发人员还可以在实际开发中根据自己的需求添加其他模块，例如著名的

```
1 import sys,os
2
3 if __name__=="__main__":
4     print sys.path
5     print os.getcwd()
```

图 2.2 Python 模块示例代码

Numpy 模块、PyGame、Matplotlib 模块等等，来扩展和完善 Python 系统。

### 2.1.2 Python 运行时系统

在图2.1的右边是由对象/类型系统（Object/Type Structures）、Python 内存分配器（Memory Allocator）和运行时状态信息（Current State of Python）3个部分构成的 Python 运行时环境。

（1）对象类型系统：对象类型系统（Object/Type Structures）主要是对 int 类型、string 类型、tuple 类型等、各种内建对象的类型以及用户自定义的数据类型的的支持。因为 Python 编程语言与 C/C++ 等编程语言不同，支持在声明和使用数据对象时不使用数据类型，所以对象类型系统在 Python 实现中占有非常重要的地位。因此对 Python 对象类型的管理和存储变得十分重要，在 Python 实现中为了支持各种数据类型以及用户自定义的数据类型，也将数据类型定义为也是一种类，比如 int 类、dict 类都是类类型的实例。

（2）内存分配器：Python 为了方便程序员编程，省区了程序员对内存空间的使用和管理这项繁琐的任务，实现了如图2.3所示的层次结构的一套内存管理策略。因为程序在执行过程中随时会创建和销毁数据对象，如果不能合理的利用内存空间不仅会影 Python 响程序的执行效率，而且会造成内存资源的浪费，所以高效并且合理的的内存管理的策略无疑是 Python 虚拟机运行效率的的前提条件。与 C/C++ 编程语言相同，Python 编程语言的内存管理策也分为对内存的申请、释放两部分，在 Python 对内存的申请只不过是 Python 对 C 编程语言的 malloc 函数进行了一层包装，而对内存的回收释放则是依赖 Python 实现的 GC（GGarbage Collection）模块。

Python 实现了对内存的动态管理，从而将开发人员从内存管理的噩梦中解放出来。然而 Python 中的内存动态管理与 Java、C# 有着很大的不同。在 Python 内部实现中，大多数对象的生存周期都是通过对对象的引用计数器来控制的。表面上看，引用计数只是一种内存垃圾收集机制，但也是一种最简单、最直观的垃圾收集技术。虽然引用计数必须在每次申请和释放内存对象时需要额外控制对象的引用计数，然而与其他垃圾回收机制必须某些特殊条件下（比如内存分配失败）才会回收内存相比，引用计数的优点就是时效性，任何对象一旦没有指向它

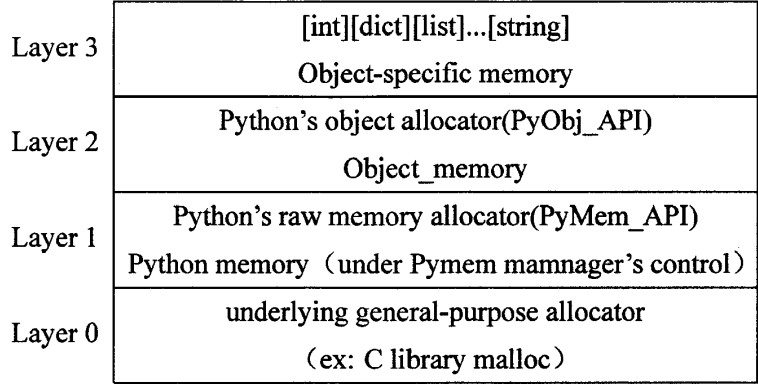


图 2.3 Python 内存管理机制的层次结构

的指针即被立即收回。

(3) 运行时状态：运行时状态保存 Python 了虚拟机在执行 PyCodeObject 对象中的 co\_code 序列时所有可能的状态和负责管理各个状态之间的切换工作，例如正常状态、异常状态（发生错误）、由 finally 语句引起的异常、break 语句引起的异常、continue 语句引起的异常、yield 操作引起的异常各种状态之间的切换，可以将 Python 运行时状态抽象地理解为一个复杂的有穷状态机。

2.1.3 Python 虚拟机

Python 的核心模块即 Python 解释器（Python Interpreter），在图2.1中间的部分，或者称其为 Python 虚拟机（Python Virtual Mechine），在图2.1所示的 Python 总体架构中的 Python 解释器模块中。自上而下的箭头的方向代表在 Python 运行时数据的流向。其中 Scanner 用于对 Python 编程语言编写的源代码进行的词法分析器，完成将输入的源代码切分为一个个的 token 的功能；而 Parser 在 Scanner 的产生的 token 的基础上上进行语法分析，用于建立抽象的语法树 AST（Abstract Syntax Tree）；Compiler 是编译模块，就像 Java 编译器和 C# 编译器所做的一样，负责根据 Parser 建立的 AST 生成 Python 虚拟机指令集——Python 字节码对象（PyCodeObject）；而 Code Evaluator 模块则负责来解释执行 Compiler 阶段生成的字节码对象 PyCodeObject，因此又将 Code Evaluator 模块称为 Python 解释器。

在图2.1中，在 Python 解释器与右边运行时状态之间向左的箭头表示 Python 虚拟机“使用”的对象/类型系统、内存分配器以及虚拟机状态模块；而向右箭头表示 Python 虚拟机在运行时会“修改”运行时状态，即 Python 虚拟机解释执行 PyCodeObject 对象中的 co\_code 序列时会时时刻刻修改解释器所处的状态，并且维护在不同的状态之间进行切换。

## 2.2 传统的 Python 字节码文件防逆转方式

伴随软件盗版是软件行业中一个很严重的问题<sup>[13]</sup>。随着着的网络化、信息化的发展，对数字产品的进行必要的版权保护迫在眉睫<sup>[14]</sup>，因此在这一节当中将介绍几种用于保护软件安全的常见技术，尤其是可以用于保护 Python 字节码文件的技术。

### 2.2.1 代码混淆技术

代码混淆技术是一种程序源码转换技术，其试图在保留程序功能和语义的前提下将一个程序从一种形式形势转换为能够妨碍攻击者理解其中的算法和数据结构，或者能够阻止攻击者从程序文本中提取有价值的信息的另一种形式<sup>[15]</sup>。代码混淆的目的不是改变源程序功能而是破坏源程序的可读性，试图使攻击者利用反编译工具分析和理解代码后所获的收益远大于其在分析代码时所花费的成本<sup>[16]</sup>。

混淆概念最早是由 Diffie 和 Hellman 在他们的文章<sup>[17]</sup>(当时并没有使用术语“obfuscation”)中开创性地提出。混淆器是基于代码转换的应用，就如同某些情况下优化器所做的工作一样。下面我将以对代码混淆形式化的定义开始，来评估混淆器的效力（到底给人为阅读代码造成多大困难）、混淆的弹性（反混淆会如何攻击）、成本（为了混淆代码需要做多少工作）描述、分类并评估混淆器。代码混淆技术在一定程度上可以看作是一种源程序代码转换技术，它将原程序  $P$  转换成新的程序  $P'$ 。除了  $P'$  与  $P$  所依赖的运行环境、使用方法等前提条件完全相同之外， $P'$  与  $P$  的功能也完全相同，唯一不同的是程序  $P'$  比程序  $P$  的对攻击者来说逻辑更加错综复杂、可读性更差，代码安全性能较强<sup>[18]</sup>。

(1) 代码混淆变换的概念：

可以将混淆变换形式化地定义为利用公式2.1所示的变换将程序  $P$  到转换为程序  $P'$  的技术<sup>[15]</sup>。

$$P \xrightarrow{\tau} P' \quad (2.1)$$

如果在公式2.1中程序  $P$  和程序  $P'$  都有着相同的明显行为，则  $P \xrightarrow{\tau} P'$  成为混淆变换。更精确的说，如果  $P \xrightarrow{\tau} P'$  是一个混淆变换，则它必须拥有下面两个必要条件：如果程序  $P$  不能顺利执行结束或者以执行错误结束，则程序  $P'$  同样不能执行结束；否则，程序  $P'$  和程序  $P$  必定都顺利执行结束且有相同的执行结果（包括输出结果）。

明显的行为则可以是“用户体验能够感知到的行为”，定义比较宽松。这意味着程序  $P'$  可能长期有着程序  $P$  没有的，但一些用户也无法感知的功能一些功

能（如创建文件、通过网络发送数据等等）。这里需要注意，并不要求程序  $P'$  和程序  $P$  有着相同的执行效率，实际上通过混淆变换得到的程序  $P'$  往往比原始程序  $P$  执行更加缓慢并且消耗更多的内存空间。

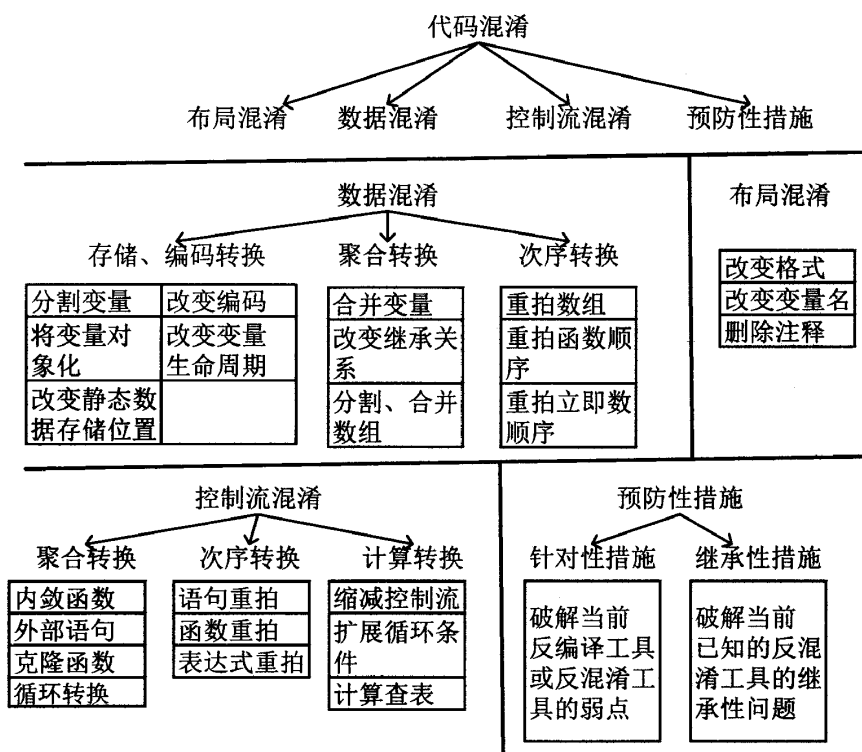


图 2.4 代码混淆的分类

代码混淆技术的主要分类如图2.4所示，主要以混淆变换所涉及到的目标，来进行分类的。有一些简单的混淆变换以代码的词汇结构（布局），例如源码的格式，变量的名称等等。例如词汇转换，修改程序的词法结构，打乱标识符，目的是将程序中无关紧要的信息进行删除或者对程序中的类名、方法名等进行替换。使其违背见名知义的软件工程原则。此外还有一些更加复杂的混淆变换，既不是对应用程序中的数据结构进行变换也不是程序的控制流进行变换。

可以按混淆变换在目标对象上所做的操作来对混淆变换分类，（如图2.4中的控制流混淆），例如对控制流或者数据进行聚合变换，这类混淆变换通常会拆分程序所创建的抽象结构，或者通过将不相关的控制流或者数据进行捆绑来创建新的虚假的抽象结构。比如常见的数据变换包括存储与编码转换、次序转换、聚集转换、分割转换。

除此之外还有一种控制流混淆算法是基于代码中插入垃圾代码来实现的<sup>[19]</sup>，它通过引入 Hash 函数来限制分支垃圾代码和循环垃圾代码的插入，意图来抵御攻击者的逆向分析的攻击。这些混淆变换会影响程序的数据或者控制流顺序，这

些顺序在大部分情况下为两个对象声明的顺序，或着实两次对程序的执行结果没有影响的执行顺序。无论是对编程人员或是逆向工程，必定有一些重要的信息都是以特定的顺序编写的。临近的一些对象或者时间要么是以空格，要么是以时间划分的，所以更高层面上说，他们在某种程度上有一定的关联。所以顺序变换则试图通过打乱变量的声明顺序或者计算的执行顺序来破坏这种关联。此外控制流的变换依赖于不透明谓词的存在，利用不透明谓词，就可以构建破坏程序控制流的模糊变换。在保证程序运行结果不变和功能完善的前提下，通过引入混沌不透明谓词来对代码逻辑进行混淆是近些年新出现的一种代码混淆方法<sup>[20]</sup>，它可以通过在程序代码中引入不透明谓词来插入无关的代码逻辑从而改变程序的控制流来。

```

1 import sys
2 from math import sqrt
3 def getpath():
4     print sys.path
5 def isprime(n):
6     k=int(sqrt(n))
7     for k in range(2,k+1):
8         if n % k == 0:
9             return 0
10            return 1
11 if __name__=='__main__':
12     getpath()
13     isprime(4)

```

图 2.5 混淆前的代码

```

1 import sys
2 from math import sqrt
3 def getpath () :
4     print sys . path
5 def isprime (n) :
6     oo000o = int(sqrt(n))
7     for oo000o in range (2,oo000o+1):
8         if n % oo000o == 0 :
9             return 0
10            return 1
11 if __name__ == '__main__' :
12     getpath()
13     isprime(4)

```

图 2.6 混淆后的代码

现有的一些代码混淆工具已经十分出色了，如 Pyobfuscate 可以删除注释、添加空白、可以混淆函数名、混淆类名、混淆变量名；Pyminifier 可以混淆函数名、混淆类名、混淆变量名、可以对多文件进行混淆提供精简代码的功能等。如果合

理地应用这些工具的确能够对自己的 Python 产品提供一定的保护。例如图 2.6 就是使用代码混淆工具对图 2.5 所示的代码进行布局混淆的前后结果。此外还有一些工具同样可以针对 Java 字节码文件进行混淆保护<sup>[21]</sup>。

## (2) 代码混淆性能评测：

由于通常认为程序  $P'$  更加晦涩（复杂或者更不易与阅读）往往基于个人的认知能力，所以要清晰地定义混淆的效力也更加困难。幸运的是，目前可以采纳软件工程分支 *Software Complexity Metrics* 中的概念。在这里，度量指标的设计伴有理解软件结构的可阅读性、可依赖性已经可维护性等目的。通常情况下度量指标的测评基于统计源代码文本众多的属性和集合这些统计结果的。虽然目前提出的一些度量指标的计算公式是从真实的研究中得出的，但也免不了有些公式具有投机性质。软件复杂性度量的详细公式可以从某些指标度量相关的文献中找到，这里不必介绍。不过可以对复杂性度量指标用一般性语言进行描述“如果除了程序  $P'$  比程序  $P$  额外包含属性  $q$  外，程序  $P'$  和程序  $P$  是相同的，则认为程序  $P'$  比程序  $P$  更加复杂。”复杂性度量指标一般包括程序长度（程序中操作符和操作数的个数）、环杂度（函数或方法中谓词的个数）、嵌套层次复杂度（函数中条件嵌套的层数）、数据流复杂度（函数内部涉及到的局部变量的个数）、参数复杂度（函数涉及到的参数个数，全局数据结构）、数据结构复杂（程序中声明的静态数据结构的复杂度）度等，在软件工程中通常的目标是降低这些指标，而在混淆工程中则期望放大这些指标。这些复杂性指标使得度量混淆变换的效力称为可能，通俗的讲，如果混淆变换能通过隐藏原始代码的意图来干扰攻击者的分析，则可以认为这个混淆变换是有力的。或者换句话说，混淆变换的效力可以认为是（对一个人）理解混淆之后的代码比原始代码到底有多困难来衡量。

在介绍混淆变换的弹性之前，有必要对混淆变换的效力和弹性进行区分。一个混淆变换如果能阻止人的阅读则是说它是有效力的，如果它能阻止反混淆器的工作，则可以认为它是有弹性的。对混淆变换的评价尺度从琐碎（trial）到单向变换（one-way）如图 2.7 所示。单向变换非常特殊，意味这种混淆永远也不能被逆向。这是因为这种混淆变换移除了对人阅读有用，而程序执行时非必需的一些信息，例如移除程序格式、打乱变量名称等等。其他的混淆变换通常则是添加一些不会改变程序行为的信息，但是它增加了人为阅读代码时的负担。这些变换则因变换的程度而不能被取消。图 2.7 也显示了去混淆变换所做的努力，既不是以多项式时间也不是指数时间。混淆变换以  $T$  函数的作用域常作为自动的逆混淆变换所需要的工作衡量标准。这是基于容易为仅影响小部分程序（而不是影响整个程序）的混淆变换构造“对抗手段”的直觉。混淆变换的作用域用代码优化理论中的术语进行定义：如果  $T$  仅影响单个基本块中的控制流图（CFG, control flow graph）则  $T$  是局部变换，如果  $T$  影响到整个程序的 CFG，则  $T$  为

全局变换。如果  $T$  影响到了多个过程之间的信息流，则  $T$  称为过程间的混淆变换。如果  $T$  影响到了独立执行的线程之间的交互控制，则  $T$  称为进程间的变换。评价混淆变换的性能指标，就是混淆变换给应用执行带来额外时间/空间的消耗。这里需要注意混淆变换实际的性能消耗往往和它所应用的场景有密切的关系。例如，在程序最顶端插入一个简单的赋值操作，仅会为程序增加常量级的开销，同样的语言插入到循环中将会带来更高的开销。除非另有说明，否则通常认为混淆变换应用在程序源码的最外层。

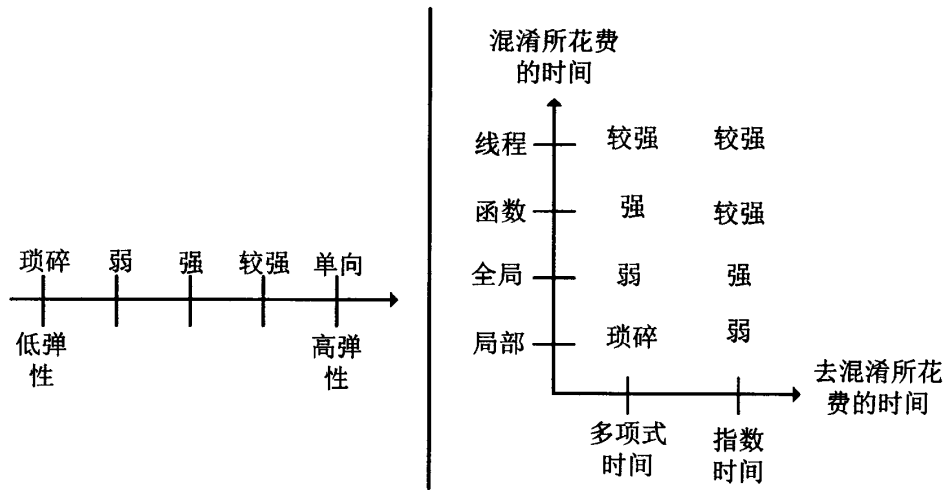


图 2.7 混淆变换的弹性度量

因为被代码混淆技术混淆过的程序代码只是使得程序在缺少注释的情况下难以阅读，生成的字节码文件仍然遵照原来的文件格式和指令集，所以不需要其他任何辅助手段或模块，生成字节码文件仍可以跨平台使用，这是代码混淆技术的一大优势。但是此方法在处理多文件项目中的导入模块和对象名称时仍有局限，并且混淆会给代码执行效率带来一定的影响，代码混淆技术并不能 Python 程序提供足够的保护，所以代码混淆技术并不是一个理想的保护方法。

2.2.2 文件加密技术

加密技术无论是对软件可执行程序或是字节码文件的保护都是一种很有效的保护方式。就采用的加密方法分类，可以将加密技术分为对称加密体系和非对称（公钥）加密体系两类。这里需要知道原始的消息（或数据）称为明文，而加密原始数据得到的数据称为密文。将明文经过变换得到密文的工作称为加密；将密文经过变换得到原始数据的工作称为解密。

对称密码<sup>[22]</sup> 是一种将明文利用置换、代替等手段替换为密文的算法，也是迄今为止，历史最久、应用最广泛的加密技术，在非对称加密算法<sup>[22]</sup> 被提出和



发明之前唯一可以使用的加密算法。如图2.8所示，由于该算法加密密钥和解密密钥使用同一个密钥  $Key$ ，并且加密过程与解密过程恰好是两个相反的过程，所以称其为对称加密算法。通常情况下，要使用对称加密算法必须要满足以下两个条件：

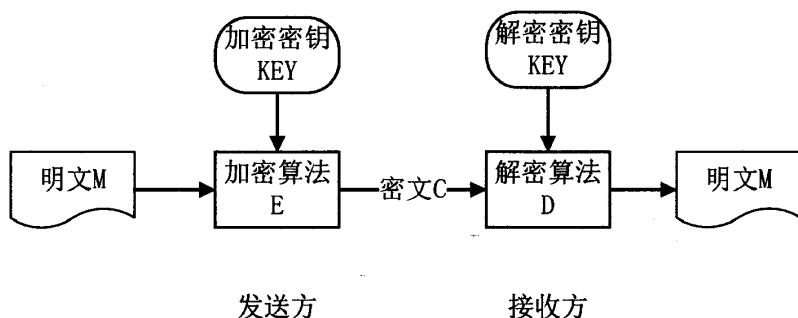


图 2.8 对称密码模型

(1) 攻击者在得到加解密算法，和某些密文与其对应的明文时，不能轻易的从这些数据中计算出加解密所采用的密钥，也就说对加密算法是有一定要求的；

(2) 密钥的传送和分发必须是加密端和解密端在安全渠道下进行，并且加密端和解密端相互都要保证密钥的安全。所以一旦密钥泄漏或者攻击者通过某种手段获得了密钥，加密端和解密端就必须立即在安全的渠道下重新更换密钥；

此外，还有需要注意，加解密双方在选取加密算法时必须选取攻击者在得到大量密文的情况下无法从中计算出加密密钥或者恢复出明文的算法，或者攻击者从中恢复出明文的日期远远超过密文的有效日期的算法，只有至少满足以上两点之一才能保证通信的安全。

目前有许多加密方法都是对称加密的衍生，如对于字母表中的每个字母用它之后的第三个字母代替的 Casser 密码，利用字母表中元素的置换来替换字符的单表代替密码、将明文中的双字母单元进行替换的 Playfair 密码、将  $m$  个连续的明文替换为  $m$  个密文字母的 Hill 密码、对单表替换进行改进的多表代替密码 (Vigenere 密码、Vernam 密码)。除此之外还有著名的 DES 算法 (Data Encryption Standard)、3DES 算法 (Triple DES)、AES 算法 (Advanced Encryption Standard)、RC2 算法、RC4 算法、RC5 算法和 Blowfish 算法等都是属于对称加密的范畴。

对传统的对称加密算法的攻击的最终目标是获得加解密端和解密端在通信时双方使用的密钥  $key$ ，而不是简单的获取某几条密文  $C$  所对应的明文  $M$ 。所以，为了从有限的信息中获取通信双方的密钥，可以通过以下面两种方法来对传统的密码体系实施攻击：

(1) 密码学分析：攻击者利用加解密算法的性质、当前所获得密文与明文对的一些特征来推导加解密算法使用的密钥。利用这种破译方式最典型的例子就是根据如图2.9所示的英文字符的频率特性来破译单表置换密码。这种攻击方式破解一些简单的加密算法十分有效，而破解 DES (Data Encryption Standard), AES (Advanced Encryption Standard) 等一些复杂加密算法则比较困难。

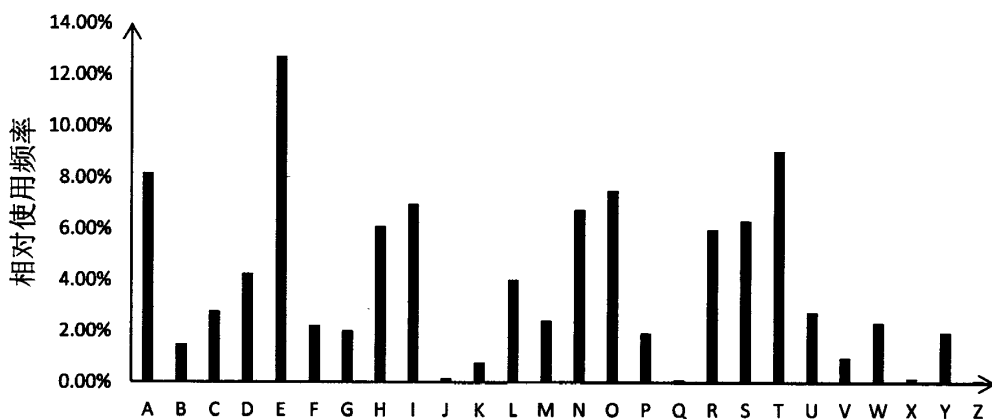


图 2.9 英文字母相对频率

(2) 穷举攻击：攻击者在截获一条密文消息并且获得了通信双方使用的假睫毛算法之后，利用解密算法对密文以可能的密钥进行解密，指导把密文转换为一条可读的有意义的信息。很显然想利用这种方式成功解密密文平均而言要尝试一半的密钥空间，例如对于密钥长度为 56 位的 DES (Data Encryption Standard) 算法就需要至少尝试  $2^{55}$  个密钥，这必然是一个十分耗时的过程。

非对称密码（或称公钥密码）<sup>[22]</sup> 的产生和应用对发展几个世纪对称密码学产生深远的影响。与之前基于置换、代替等初等变换的对称密码相比，非对称密码的实现和应用是基于数学函数的如图2.10所示。与图2.8中的对称密码在加密端和解密端采用同一个密钥 key 对数据加解密相比，非对称密码加密端采用的加密密钥和解密端采用的解密密钥是两个不同的密钥。

图2.10中，通常由于加密段加密之后的密文只有特定的解密端的密钥才能解密，其他密钥很难解密，所以一般将加密密钥公开发布而将解密密钥保密存储，所以通常将加密端使用的密钥称为 Public Key，而将解密端使用的密钥称为 Private key。

最典型和成功的非对称加密算法要数 Ron Rivest 等人在 1977 年在 MIT 时发明的 RSA 算法，并且 RSA 算法也是最早提出的非对称加密算法。RSA 算法是基于大数分解难题和分组密码而建立的算法，其依赖的输入有一个长为明文消息

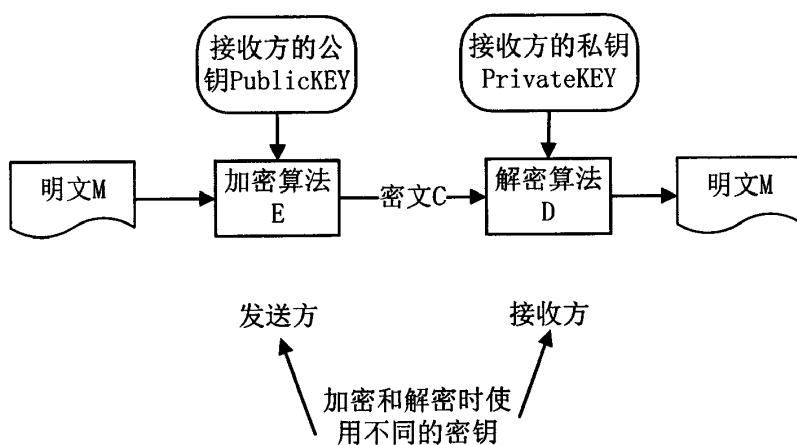


图 2.10 非对称密码模型

$M$  的, 加密密钥  $e$  解密密钥  $d$ , 以及一个模数  $n$ , 其加解密过程如公式2.2所示。

$$C = M^e \bmod n \quad (2.2)$$

$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

公式2.2中  $e$  和  $d$  都是两个提前准备的大素数, 而  $n$  是收发双方都已知的, 并且满足公式2.3,  $n$  通常情况下是个 1024 位的二进制数, 或者 309 位的 10 进制数。由于大数分解的难题在已知  $n$  和  $e$  的情况下导出  $d$  是非常困难的 (密钥长度 1024 位时破解 RSA 算法需要 2000 多年的时间)。

$$n = ed \quad (2.3)$$

从公式2.2中可以看出, 使用 RSA 算法加密之后的密文  $C$  所得取值范围为 0 到  $n - 1$ , 同样为了能够从密文  $C$  中顺利恢复出明文  $M$ ,  $M$  的大小也必须属于 0 到  $n - 1$ , 也就说说明文  $M$  的长度必须  $len$  必须满足公式2.4, 否则从密文回复出的明文将会被截断。

$$0 \leq len \leq \log_2 n + 1 \quad (2.4)$$

RSA 加密算法使用公钥和私钥两个密钥和不对称的方法来实现它的公钥密码体系。公钥加密方法依赖于公钥加密算法、公钥解密算法以及公钥密钥。公钥用于加密消息或验证签名, 任何人都可以知道; 私钥用于解密消息或创建签名, 加密消息的正文无法解密消息。素数分解在计算上非常困难, 所以 RSA 算法不那么容易破解, 因为如果  $n$  足够大的话, 寻找  $n$  的两个因子  $p$  和  $q$  的难度非常大, 所以 RSA 的安全性极高。明文通过公钥加密并成为密文, 并且可以在网络中传输, 接收者可以通过私钥解密密码, 同时私钥可以用于密钥分发或数字签名。素数因子分解在计算上非常困难, 所以它不能容易地破解。公钥用于加密消息或验

证签名,任何人都可以知道。私钥用于解密消息或创建签名。加密消息的正文无法解密消息。明文通过公钥加密并成为密文,并且可以在网络中传输。接收者可以通过私钥解密密码。它可以用于密钥分发或数字签名。有三种攻击 RSA 加密的方法:强力关键搜索、数学攻击和定时攻击。由于字符,它可以用于软件的注册。作者可以使用私钥加密用户名并为用户创建密钥。软件可以通过公钥解密密钥以获得正确的注册。即使破解者得到公钥,它也不能获得私钥来创建注册密钥。RSA 的缺点是缓慢的,特别是对于快速运行的应用程序,并且消息必须分成块,并且每个块被单独加密。

无论是对称加密方法,还是非对称加密,加密术在字节码文件保护中是利用 Python 虚拟机导入字节码文件到系统中执行的过程进行实现的,其最核心的技术就是重载 Python 文件加载器对象,嵌入解密算法。其可以分为两步完成:首先通过特定的加密算法玩成对字节码文件的加密;其次在 Python 虚拟机载入字节码文件前对其进行解密,由于 Python 与 JAVA 类似,虚拟机每次读取需要的字节码文件时都需要利用到加载器对象(loader),因此解密阶段可以重载该对象中的接口(loader\_module),完成对加密后的文件的解密工作。但加解密方案应用在 Python 字节码保护上也存在自己的缺陷:

(1) 如果解密算法太过复杂,则会对 Python 虚拟机的整体性能带来较大的影响;

(2) 需要妥善的存储和保管密钥,如果密钥被他人所获取,则前面所做的工作就前功尽弃。

除以上两点,由于载入内存的字节码文件要为虚拟机运行做准备,是解密后的文件,也容易被攻击者采用转储内存的方式获得解密后的字节码文件。

### 2.2.3 本地编译技术

本地编译技术是一种将 Java、Python 等脚本程序和解释器一起编译为平台相关的程序的技术。这种技术其本质可以为一个打包过程。它借助于特定的工具,如 py2exe,将 Python 解释器编译为平台相关的动态链接库,然后将目标脚本程序进行加密,最后将使用一个额外的可执行程序来利用上面的动态链接程序和加密的文件。例如即使 Python 程序开发人员仅需要发一个简单的 test.py 文件(大小约为 2KB)构成的应用,也要发布如图2.11中所示的一大堆文件(大小约为 7MB)。

对 Java 程序进行本地化处理是指将使用 Java 编程语言编写的应用程序转换为依赖特定平台的应用程序(如 Windows 下的.exe 程序)<sup>[23]</sup>。例如有针对 Java 的本地编译工具 TurboJ<sup>[24]</sup>,它可以在离线的环境下将 Java 代码转换为针对特定平








Name	Date modified	Type	Size
 test.exe	2016/11/2 18:41	Application	26 KB
 python27.dll	2016/6/27 15:25	Application extension	3,316 KB
 _hashlib.pyd	2016/6/27 15:26	PYD File	1,444 KB
 bz2.pyd	2016/6/27 15:25	PYD File	92 KB
 select.pyd	2016/6/27 15:25	PYD File	13 KB
 unicodedata.pyd	2016/6/27 15:25	PYD File	677 KB
 library.zip	2016/11/2 18:41	WinRAR ZIP 压缩文件	1,791 KB

图 2.11 py2exe 示例

台的代码。在 TurboJ 运行时，它将产生拥有一个用于内存和线程管理、类文件和相应的库载入操控的标准的 JVM（Java Virtual Machine）接口的代码。这种结合方式的的优势就是支持无缝代码混合执行，一些类文件被提前编译，而另一些类文件则在代码执行时由 JIT（Just-In-Time compiler）解释执行或者动态编译。TurboJ 目前可以和 HP/UX、Linux、Solaris、ScoUNIX 和 VxWorks 下针对 Java 的 Wind River’s Tornado 等众多平台下的 JVM 配合工作。如果 Java 应用在编译时它所依赖的所有库文件都是可用的，即应用不需要动态地从网络上加载类文件，则可以说该应用是闭合的（closed），否则称该应用是开放的（open）。在嵌入式领域，这对 TurboJ 来说是一个重要的目标，然而 TurboJ 目前不支持开放的（open）应用的转换。

虽然该技术对 Python 程序的起到了保护作用，但是采用这种方式使 Python 程序失去了跨平台性，这是以牺牲 Python 作为脚本程序最优秀的特征为代价的<sup>[25]</sup>。采用这种方式使 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，大大增加了目标程序的所占用的空间。所以本地编译技术也不是一个较好的 Python 字节码文件防逆转方式。

2.2.4 数字水印技术

数字水印作为近年来出现的数字产品版权保护的技术，因其以可携带软件所有权等信息，并且在软件开发商维护自身利益时提供必要的证明，防止数字产品的传播和肆意更改，所以在数字产品的认证、分发、保护、防伪和用户信息标识等方面有着诸多的应用<sup>[26]</sup>，所以数字水印技术也是当前国际学术界的研究热点问题<sup>[14]</sup>。数字水印是用于标识软件所有权等信息而嵌入在软件中的的特定信息，并且数字水印的存在不会对软件功能性、可靠性、易适用性和性能产生影响。数字水印的存在并不能阻止攻击者对 Java 源代码编译生成的.class 文件和 Python 源代码编译生成的.pyc 文件的反编译，但是能够阻止数字产品被偷窃，或者当偷窃发生时为认证人员提供数字产品的拥有权证明<sup>[27]</sup>。数字水印技术目前

是防止软件盗版最苛刻的技术<sup>[28]</sup>。

数字水印可以根据不同的分类标准分为许多类型并且各具优缺点<sup>[29]</sup>。可以根据数字水印是否是预先设置的分为以下两类：嵌入在程序的数据结构或代码中的静态水印，和嵌入在程序运行时构建的数据结构中的动态水印两大类<sup>[11][12]</sup>。除此之外，数字水印还有诸多的分类标准和对应类型，如从数字水印的实现算法和外观上进行分别分为，变换域与空间域水印、不可见与可见水印...但无论是那种类型的水印它都应该具有一下4个特征<sup>[26]</sup>：

(1) 鲁棒性：指数字水印具有在各种攻击之后存活的能力，如果攻击者只知道嵌入在软件中的部分数字水印信息，那么当他试图从软件中去除嵌入在其中的水印信息时应会给软件的功能性、可靠性、易使用性和效率等各个方面带来一定的负面影响，引起软件质量下降的问题<sup>[14]</sup>；

(2) 隐形性：指数字水印是不可被知觉的，且不影响宿主数据的正常使用；

(3) 可鉴别性：水印信息能够为受保护的数字产品的所有权提供全面和可靠的证据；

(4) 安全性：指数字水印信息只有唯一正确的符号来识别，只有授权的合法的用户才能检测、提取甚至修改水印信息。

数字产品水印算法，如图2.12所示，一般分为嵌入水印和识别水印两个环节<sup>[28]</sup>。嵌入水印是指将水印信息  $W$  嵌入到目标程序  $P$  中，可以用过程  $P' = E(P, W, k)$  表示，其中  $P'$  为带有水印的程序， $k$  为密钥；识别水印过程又可以分为提取水印和检测水印为两个子过程来，提取水印是指从带有水印的程序  $P'$  中提取水印信息  $W'$ ，可以用过程  $W' = \bar{E}(P', k)$  表示，其中  $\bar{E}$  为嵌入操作  $E$  的逆操作， $W'$  为通过上述步骤获得的水印信息；检测水印即检测  $W'$  是否为当初嵌入的水印  $W$  信息，可以用过程  $P' = D(P', W, k)=0$  或  $1$  描述，其中  $1$  代表就是嵌入的原始水印  $W$ ，而  $0$  代表不是。

由于 Java、Python 等程序代码容易被反编译，从而导致对使用 Java、Python 等编程语言开发的程序在未经开发商授权的情况下肆意的拷贝和修改再出版的问题十分严重。所以当软件开发商为维护自身利益，在某写情况下需要证明程序版权的所有者时，就可以利用嵌入在软件中的数字水印作为证据。根据数字水印是否在程序运行之前被预设，数字水印分为嵌入在程序数据结构或代码中的静态水印，和嵌入在程序运行时的数据结构中的动态水印两大类。与静态水印相比，动态水印的优势在于将水印嵌入在程序运行时存储在内存的数据中，而不是在程序代码中，所以对程序代码进行的混淆变换和优化等去水印技术的攻击有一定的免疫作用。

针对 Java、Python 等程序的静态水印的嵌入流程如图2.13所示，一般分为2个步骤<sup>[30]</sup>：

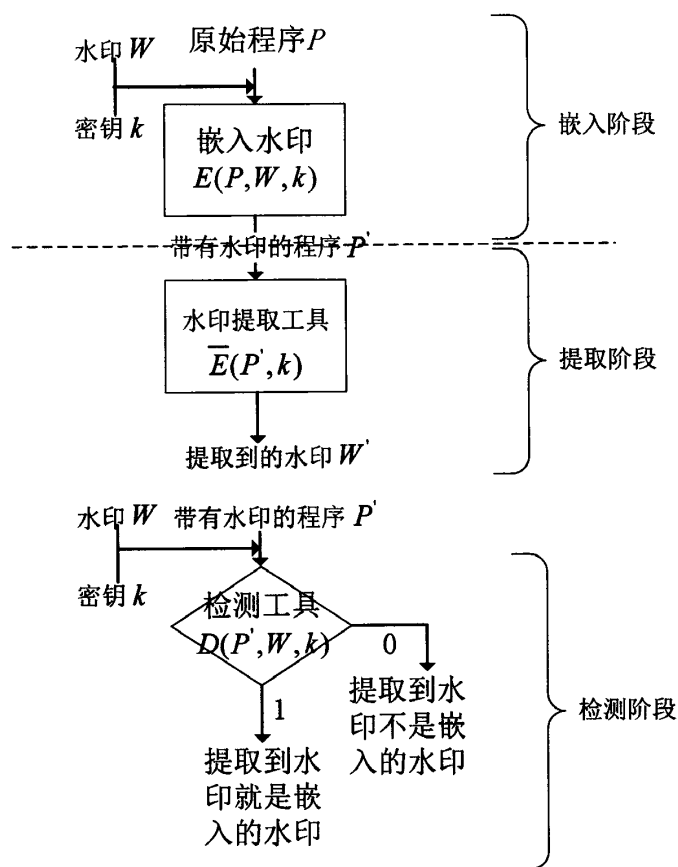


图 2.12 数字水印算法流程图

(1) 创造水印空间：在 Java、Python 等程序中插入静态数字水，首先需要创造嵌入数字水印的空间。静态数字水印的空间多是通过在 Java、Python 等代码中插入一个函数体来实现的。该函数的函数体具有哪些内容、可以完成什那些功能都不重要，重要的是该函数的函数必须拥有可以嵌入特定数字水印的空间。

通常情况下用以嵌入数字水印的函数是不会被程序逻辑调用执行的，因此称之为“哑函数”。虽然从表面上看用于嵌入数字水印的哑函数的函数体只是用于嵌入特定的数字水印，对于应用程序的功能扩充和性能提升毫无作用，但是由于 Java 源码文件编译生成的.class 文字和 Python 源码文件编译生成的.pyc 文件很容易被攻击者利用 JD-GUI 和 uncompyle2 等反编译工具反编译出其中的源代码，因此如果程序开发人员仅仅在 Java 编程语言或 Python 编程语言开发的应用中声明一个用于存放数字水印的函数，而不能与程序的代码中的其他逻辑发生关系，那么富有经验的 Java 和 Python 程序员就可以轻易从反编译出来的源代码中找出隐藏在其中的用于隐藏数字数字的水印的函数的声明<sup>[31]</sup>，从而轻易地从开发人员用 Java、Python 等编程语言开发的应用源代码中将哑函数删除。所以在哑函数设计之处必须仔细全面的考虑，使用于嵌入数字水印的函数在内容和

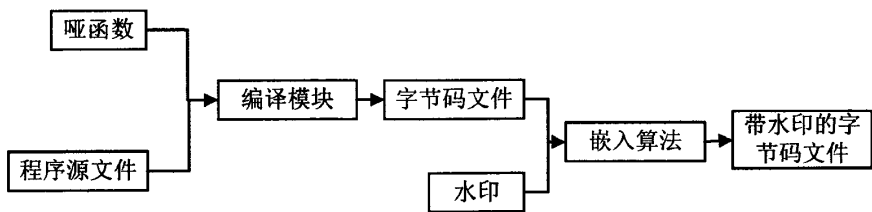


图 2.13 哑函数的嵌入流程

功能和应用程序其他部分尽可能的相似，并且必须要与程序的其他部分发生合理的联系，例如可以在程序的其他部分以图2.14所示的方式来调用用于嵌入数字水印的哑函数。

```
1  ...
2  switch(value){
3      ...
4      case flag:
5          WaterMarking();
6      ...
7  }
8  ...
9  //
10 if(CheckResult())
11     WaterMarking();
12 ...
```

图 2.14 哑函数调用示例

利用图2.14中的所示的 switch 语句或者 if 语句就可以使哑函数和程序逻辑发生联系，其中 flag 是一个 value 永远也取不到的值，并且 CheckResult() 是一个永远为假的复杂函数或复杂的表达式。这样，在使用用于嵌入数字水印的函数和程序的其他部分发生联系后，就可以将使用 Java 和 Python 编程语言编写的源代码利用相应的编译器编译成对应的字节码文件。

(2) 水印编码：

在将数字嵌入目标软件时，通常依赖于一个嵌入函数  $g(x)$  将.class 或.pyc 字节码文件中哑函数所对应的空间，在符合字节码规范的前提下转换成特定的数据。例如，当程序开发人员要嵌入的水印为“Bob’s Message for Watermarking”，就可以使用函数  $g(x)$  将这个水印信息转换为其对应的比特流“110101011010110110...”，并替换哑函数中的数据。

虽然理论上的水印能够作为软件所有权的有力凭证，但实际中都很容易通过混淆变换和优化等措施从软件中移除<sup>[11][12]</sup>，所以水印技术并不能为 Java、Python 等字节码程序提供有效的保护。



## 2.3 本章小结

本章首先介绍了 Python 总体架构，Python 的内建模块、运行系统和虚拟机相关的内容，以期为后文认识与理解 Python 源程序和字节码文件的运行提供必要的背景基础。

接着对代码混淆技术进和本地编译技术进行了介绍和简单的实验评测，介绍了基于对称密码和非对称密码的文件加密技术应用于 Python 字节码文件保护时的实现方案和存在的缺点，对数字水印技术应用在 Python 字节码文件的实现算法和缺点进行了简要的讲解。由于现有的诸多 Python 字节码文件防逆转方式都存在一定的缺陷，因此急需对现有的防逆转方式予以改善或提出一种新的字节码文件防逆转方式来革新 Python 字节码文件防逆转的现状。



## 第3章 Python 字节码文件与其解释执行机制

Python 字节码文件中保留了源码文件名、路径、常量、符号以及操作码序列等全部源码信息是针对 Python 虚拟机的具有特定结构的文件，具有可以跨平台使用的特性。而 Python 字节码文件之所以容易被反编译正是基于字节码文件上的特殊结构和其中保存的诸多信息之上，所以本章将首先对 Python 字节码文件的结构、运行机制进行介绍，以便为下一章操作码替换与合并打下基础。

### 3.1 Python 字节码文件

当使用 Python 编程语言编写了用以完成特定功能的程序 `programe.py` 时，在命令行输入 `python programe.py` 后，就可以看到操作系统会按照预期执行源码文件 `program.py` 中的操作，但 Python 虚拟机在执行使用 Python 编程语言编写的源码程序（.py）之前，要完成一个非常重要而且复杂的工作，那就是编译源码（.py）文件。事实上，Python 解释器在执行任何一个 Python 源码文件前，首先需要对源码文件中的代码编译为虚拟机可以理解接受的对象，即一组 Python 的字节码（byte code），然后将字节码交给 Python 虚拟机（Python Virtual Machine），由虚拟机按照字节码中操作的顺序一条一条的执行，从而完成对 Python 源码文件的执行操作。

通常认为 Python 编译器对 Python 编程语言编写的源码程序编译的结果是 `pyc` 文件，但这种说法是不太准确的。Python 源码文件中一般包含一些字符串、常量等对象，还有一些对这些对象的操作等静态信息，通过对源码文件的编译，这些包含在 Python 源码文件的静态信息全部被 Python 编译器进行了收集和整理，并存储到一个运行时对象当中（`PyCodeObject`），等到程序运行结束 Python 就会将这个对象（`PyCodeObject`）中包含的信息写到常见的字节码文件（.pyc）中。经过上面的介绍，知道在 Python 源码运行期间，其编译结果存在于内存中的 `PyCodeObject` 对象当中，等程序运行完毕之后，编译的结果就被写到了磁盘上的 `.pyc` 文件中。当程序下一次运行时，Python 会根据磁盘上的 `pyc` 文件中记录的编译结果直接在内存中建立 `PyCodeObject` 对象，而不用再一次对 Python 源码文件进行编译。

#### 3.1.1 PyCodeObject

需要理解 Python 虚拟机是如何解释执行字节码文件的过程，就必须首先了解 Python 编译器的编译出的与自字节码文件一一对应的 `PyCodeObject` 结构。

如图3.1所示，首先来介绍一下 Python 源码中文件 Include/code.h 中是如何声明 PyCodeObject 类型的。

```

1  /* Bytecode object */
2  typedef struct {
3  PyObject_HEAD
4  int co_argcount;      /* #arguments, except *args */
5  int co_nlocals;      /* #local variables */
6  int co_stacksize;    /* #entries needed for evaluation
7                      stack */
8  int co_flags;        /* CO_..., see below */
9  PyObject *co_code;    /* instruction opcodes */
10 PyObject *co_consts; /* list (constants used) */
11 PyObject *co_names;   /* list of strings (names used) */
12 PyObject *co_varnames; /* tuple of strings (local variable
13                      names) */
14 PyObject *co_freevars; /* tuple of strings (free variable
15                      names) */
16 PyObject *co_cellvars; /* tuple of strings (cell variable
17                      names) */
18 /* The rest doesn't count for hash/cmp */
19 PyObject *co_filename; /* string (where it was loaded
20                      from) */
21 PyObject *co_name; /* string (name, for reference) */
22 int co_firstlineno; /* first source line number */
23 PyObject *co_lnotab; /* string (encoding addr->lineno
24                      mapping) See Objects/lnotab_notes.txt for details. */
25 void *co_zombieframe; /* for optimization
26                      only (see frameobject.c) */
27 PyObject *co_weakreflist; /* to support weakrefs
28                      to code objects */
29 } PyCodeObject;

```

图 3.1 PyCodeObject 类型的定义

事实上，Python 编译器在将 Python 源码文件（.py）文件编译为字节码文件（.pyc）的过程中，对于源码中的每个 Code Block 都会创建一个与之对应的 PyCodeObject 对象。那 Python 编译器是怎么确定多少代码算一个 Code Block 的呢？那就是每当进入一个新的名字空间的时候，或者说进入一个新的作用域的时候，就算是进入一个新的 Code Block 了。

这里，需要对 Python 中一个重要概念名字空间进行介绍，名字空间是符号的上下文环境，符号的含义取决于其所在的名字空间。进一步说，也是就在 Python 中，当看到一个变量名称时，无法明确的确定该变量的类型值到底是什么，要确定该变量的值还要依赖于该变量所在的名字空间。

为了更好的理解 PyCodeObject 与 Python 虚拟机解释执行机制，这里需要对 PyCodeObject 对象中的各个域需要进行介绍，如图3.2所示，其中，位置参数可以理解为在调用函数时，函数的内部的参数的值是通过参数位置来确的参数；

co\_flags 对于理解 Python 虚拟机没有太多帮助，在表中没有详细介绍。

co_argcount	Code Block 的位置参数个数，比如一个函数的位置参数个数
co_nlocals	Code Block 中局部变量的个数，包括位置参数的个数
co_stacksize	执行该段 Code Block 所需要的栈空间
co_flags	N/A
co_code	Code Block 编译所得的字节码序列，以 PyStringObject 的形式存在
co_consts	PyTupleObject 对象，保存 Code Blocks 中的所有常量
co_names	PyTupleObject 对象，保存 Code Blocks 中的所有符号
co_varnames	Code Block 中局部变量名集合
co_freevarsa	Python 实现闭包所需要的东西
co_cellvars	Code Block 中内部嵌套函数所引用的局部变量名称
co_filename	Code Block 对应的.py 文件名称
co_name	Code Block 的名称，一般是函数名或者类名
co_firstlineno	Code Block 在对应的.py 文件中的起始行号
co_lnotab	字节码指令与.py 文件中的 source code 行号对应关系，以 Py-StringObject 的形式存在

图 3.2 PyCodeObject 中各个域的含义

3.1.2 co\_code 域

在上面的 PyCodeObject 中，需要特别注意其中的 co\_code 域。co\_code 域中存放的就是源码文件编译生成的字节码指令序列，也就是说 co\_code 规定了 Python 虚拟机拿什么对象执行什么操作的信息。下面将对 co\_code 域进行介绍，Py-CodeObject 中 co\_code 域其逻辑结构如图3.3所示，实际上 co\_code 对象的 HEAD 和 BODY 部分在内存中不是连续存储的，而是 HEAD 通过一个指针指向 BODY 所在的区域，图3.3仅为了表示方便将其画为连续存储的。

通过图3.3可以看到，PyCodeObject 中的 co\_code 域的逻辑结构主要由 HEAD 和 BODY 两大部分组成，其中 HEAD 是一个如图3.4所示 PyObject\_Header 结构体，而 BODY 是一个如图所示 4.2 的由操作码和参数构成的一维数组。

而事实上，co\_code 是一个 PyString 类型对象，HEAD 部分无非是记录了 co\_code 的引用计数、类型信息、对象的值（例对象 a 是一个整数类型，则该域记录对象的实际值 5）、对象的实际包含的元素个数（例如，对象可能是一个 string 类型，则该域记录对象中包含的元素个数，若对象是个 list 类型，则需要记录 list

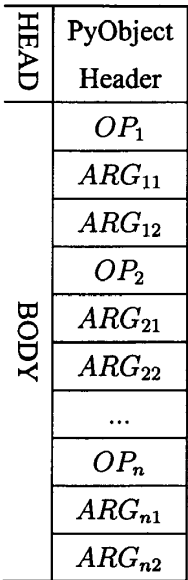


图 3.3 co\_code 的逻辑结构示意图

中元素的个数)、还有就是对象实际所在地址的指针等信息。也就是说操作码序列最有意义的地方就在这个 BODY 部分，BODY 部分保存了 Python 虚拟机需要执行的字节码指令和这些指令所需要的参数。Python 虚拟机对字节码指令解释执行的实质就是对 `co_code` 域中的 BODY 部分进行从头到尾的遍历，依次对 BODY 中的每个字节码指令进行解释执行。

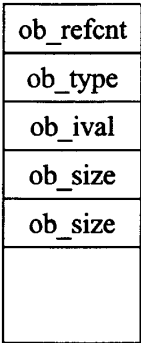


图 3.4 PyObject\_Header 的逻辑结构示意图

3.2 Python 字节码文件的解释执行

前面介绍了 Python 字节码文件和操作码序列的相关内容，下面将对 Python 虚拟机是如何解释执行字节码文件进行介绍。

前面一节介绍 `PyCodeObject` 中的 `co_code` 域实际上是个 `PyStringObject` 类型

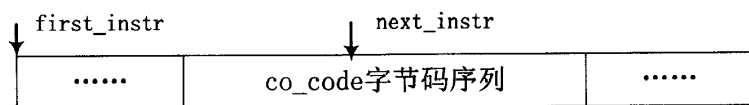


图 3.5 Python 遍历字节码指令序列状态示例

的对象，而其中的字符数组 BODY 部分才是最有意义的部分。也就是说，这个操作码序列最有意义的 BODY 的结构就是一个 C 语言中普通的字符数组。事实

```

1 PyObject *
2 PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
3 {
4     ...
5     why=WHY_NOT;
6     ...
7     for (;;) {
8         ...
9         fast_next_opcode:
10            f->f_lasti=INSTR_OFFSET();
11            opcode=NEXTOP(); //获取操作码
12            oparg=0;
13            //如果操作码需要参数，则去获取该操作码对应的参数
14            if(HAS_ARG(opcode))
15                oparg=NEXTARG();
16            dispatch_opcode:
17                switch(opcode) {
18                    case NOP:
19                        goto fast_next_opcode;
20                    case LOAD_FAST:
21                        ...
22                }
23            }
24 }

```

图 3.6 Python 虚拟机解释执行逻辑示意

上，在 Python 虚拟机内部就是利用 3 个变量来控制对操作码序列的遍历过程，因为操作码序列的实际上是个字符数组，所以这 3 个变量都是 `char *` 类型的变量：`first_instr` 变量永远指向字节码指令的起始位置（也就是 BODY 的起始指位置）；`next_instr` 永远指向下一条待执行的操作码码所在的位置；`f_lasti` 指向上一条已经执行过的操作码所在的位置，也就是 `next_instr` 所指向的前一个操作码所在的位置。如图 3.5 展示了这 3 个变量在 Python 虚拟机解释执行操作码序列某一时刻的状态。

Python 虚拟机是如何对图 4.2 所示的操作码序列进行解释执行的呢？可以在 `Python/ceval.c` 文件中来看看 Python 虚拟机解释执行操作码指令的整体架构，其实质就是一个 `for` 循环加上一个巨大而复杂的 `switch/case` 结构，图 3.6 对 Python 虚拟机的解释执行字节码的结构做了展示。图 3.6 中的代码只展示了一个极其简

化的之后的 Python 虚拟机的模型，更加具体和详细的内容可以从 Python/ceval.c 源码文件中获知。

在这个解释执行过程中对字节码的解释执行是通过图3.7中所示的3个宏来实现的，其中 INSTR\_OFFSET() 用来获取当前准备要执行的操作码距离操作码序列起始位置的偏移量；而 NEXTOP() 则是 Python 虚拟执行完一条操作码之后获取下一个要执行的操作码的值，在图3.6中的11行处调用，NEXTARG() 则是用于获取下一条欲执行的操作码的参数，在图3.6中的14行处调用，例如当操作码序列中的参数序列为0、1时则获得的值应为256，若恰好此时操作码对应的操作为 LOAD\_CONST，则 Python 虚拟机在执行 LOAD\_CONST 操作时定会将代码所示的 PyCodeObject 中 co\_const 列表中的第256个元素读取并压栈。

```

1
2 #define INSTR_OFFSET() (int)(next_instr-first_instr)
3 #define NEXTOP() (*next_instr++)
4 #define NEXTARG() (next_instr+2,(next_instr[-1]<<8)/
5                     +next_instr[-2])

```

图 3.7 Python 虚拟机中几个重要的宏定义

```

1 /* Status code for main loop (reason for stack unwind) */
2 enum why_code {
3     WHY_NOT = 0x0001, /*No error*/
4     WHY_EXCEPTION = 0x0002, /*Exception occurred*/
5     WHY_RERAISE = 0x0004, /*Exception re-raised by 'finally'*/
6     WHY_RETURN = 0x0008, /*'return' statement*/
7     WHY_BREAK = 0x0010, /*'break' statement*/
8     WHY_CONTINUE = 0x0020, /*'continue' statement*/
9     WHY_YIELD = 0x0040 /*'yield' operator*/
10 };

```

图 3.8 Python 虚拟机状态示例代码

前面在介绍 Python 操作码相关内容时强调，Python 操作码部分是带参数的，部分是不带参数的（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，其中每个参数占两个字节）。而判断操作码是否带参数是通过 HAS\_ARG() 这个宏来实现的。所以，对于 co\_code 域中不同的操作码，由于存在带参数和不带参数的区别，所以当执行完一条指令后，next\_instr 的偏移可能是不同的：对于不带参数的情形只需要执行 next\_instr++ 操作即可，而带参数的情形则需要执行 next\_instr=next\_instr+3 操作，但是无论如何 next\_instr 都始终指向下一条 Python 虚拟机需要执行的操作码，这一点与 x86 平台上的 PC 寄存器类似。

Python 虚拟机在获得了一个操作码和其所需要的参数后，会利用 switch 语句来查找操作码对应的 case 语句，并利用 case 语句包含的内容对操作码进行解



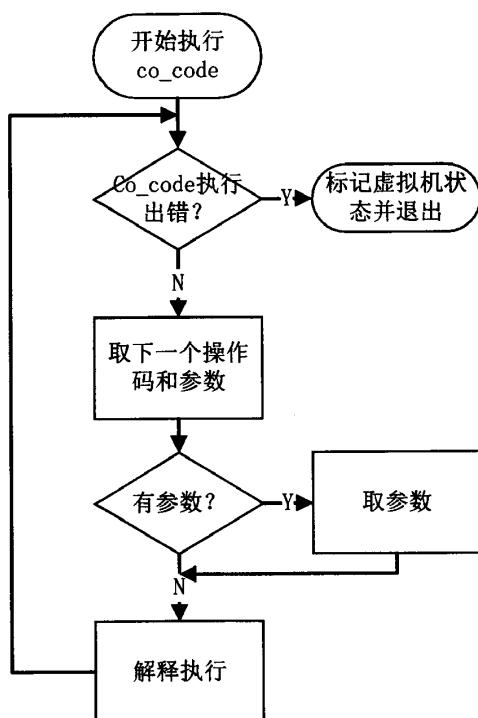


图 3.9 Python 虚拟机执行流程图

释执行。所以，每个操作码和每个 case 语句都是一一对应的关系，在 case 语句中就是 Python 对操作码解释执行的实行。在成功执行完 co\_code 域中的一条操作码之后，Python 的执行流程会跳转到 fast\_next\_opcode 处，或是 for 循环处，无论如何，Python 虚拟机接下来的动作都是去获取下一条需要解释执行的操作码和其对应的参数，并对其进行解释执行。

在图3.6所示的代码中需要注意的是 Python 虚拟机中名为“why”的变量，它指示了 Python 虚拟机在利用 for 循环代码块执行完 co\_code 操作码序列中的每一个操作码后 Python 虚拟机的状态。因为 for 循环中的代码块不能保证对 co\_code 操作码序列中的每一个操作码的执行都是顺利进行的，所以很有可能在执行 co\_code 操作码序列中的某个操作码时，发生了错误，也就是发生了“异常”（exception）时，导致 Python 虚拟机不能继续执行 for 循环对应的代码块，而不得不进入 for 以外的代码块。所以 Python 虚拟机在退出 for 循环的时候，就需要知道是什么原因导致对操作码的解释执行结束，是执行完所有的操作码正常结束？还是发生错误导致虚拟机退出？why 变量的功能就是用来标记 Python 虚拟机退出状态。变量 why 在虚拟中 ceval.c 中的定义如图3.8所示。

通过上面的介绍，可以知道了 Python 虚拟机对操作码的解释执行过程了，在 Python 虚拟机进入 Python/ceval.c 文件中的 PyEval\_EvalCodeEx() 函数的 for 循环之后，就会按照如图3.9流程图所示，Python 虚拟机将一条一条的对 co\_code 域

中的操作码进行解释执行，最终完成整个 Python 程序的执行任务。

### 3.3 本章小结

本章介绍了 Python 字节码文件和 PyCodeObject 的对应关系、PyCodeObject 的结构和内容、Python 字节码文件中的操作码序列的解构和 co\_code 域,和 Python 虚拟机解释执行字节码文件的架构进行了介绍，为下文在 Python 虚拟上设计基于操作码替换和合并的 Python 字节码文件防逆转策略打下基础。

## 第 4 章 操作码替换与合并的设计

通过前面第 2、3 章的介绍，知道 Python 的是首先将使用 Python 编程语言编写的源码文件（.py）利用编译模块（py\_compile 或 compileall）编译为字节码文件（.pyc 或 .pyo）再进行解释执行的，尤其是对字节码文件（.pyc）对应的 PyCodeObject 对象中的 co\_code 域包含的操作码逐一进行解释执行的。而 Python 字节码文件易被反编译的主要原因是因为字节码文件保留了 Python 源码文件的所有信息，并且字节码文件中的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，所以通过改变操作码序列的结构或隐藏操作码的含义是保护 Python 字节码文件的重要的方法，然而这些方法会可能对虚拟机执行字节码文件的结果造成影响。所以，为了防止 Python 字节码文件（.pyc）被他人反编译出其中包含的源码信息，本章要讨论两种方法，使得在不改变 Python 编译生成的字节码文件执行结果的前提下，改变字节码文件（.pyc）包含的操作码序列（co\_code）的内容和结构都发生变化，从而达到 Python 字节码文件防逆转的目的。

### 4.1 操作码序列

本文前面绍了 Python 字节码文件（.pyc）的结构、Python 字节码文件（.pyc）与 PyCodeObject 对象的对应关系，尤其是对 PyCodeObject 对象中用于指定 Python 虚拟机需要执行哪些操作的 co\_code 域进行了重点介绍，那么本节将对 Python 虚拟机到底有哪些操作和对这些操作组成的序列进行抽象介绍。

```
1      /* Instruction opcodes for compiled code */
2
3      #define STOP_CODE 0
4      #define POP_TOP 1
5      #define ROT_TWO 2
6      #define ROT_THREE 3
7      #define DUP_TOP 4
8      #define ROT_FOUR 5
9      #define NOP 9
10     ...
```

图 4.1 Python 操作码定义示例

**定义 1 操作码：**操作码就是 Python 虚拟机中的指令码，占一个字节的长度，它规定了 Python 虚拟机需要执行哪一条指令。在 Python 的 Include/opcode.h 文件中定义了一系列的可以从源码文件（.py）中编译生成的操作，这些操作同样也是虚拟机可以解释执行的操作。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，其中每个参数占两个字节。如图4.1所示展示了 Python 2.7.9 版本的 Include/opcode.h 中的部分操作码的定义。

从图4.1中的操作码定义中可以看出，每个具有特定语义的操作都是由一个正整数表示的，例如“POP\_TOP”代表将栈顶的元素出栈并返回是由“POP\_TOP”后面的“1”所表示的，“ROT\_TWO”代表将栈顶的两个元素调换位置由“ROT\_TWO”后面的“2”所表示的... 也就是说当 Python 虚拟机拿到操作码为“1”时，就知道了要执行将栈顶元素出栈并返回的操作，当 Python 虚拟机拿到操作码为“2”时，就知道了要执行将栈顶的两个元素调换位置的操作...

**定义 2 操作码序列：**一个字节码文件的操作码序列  $S$  是由操作码和其所带的参数构成的一个序列。其实质为 `co_code` 域中的 BODY 部分，为了表示方便，将其进行转换为如图4.2所示的结构，实际中可能有些操作码并不带参数，为了方便仅展示带参数的操作码构成的序列的情况。在操作码序列  $S$  中  $OP_i$  是来自 Include/opcode.h 中的定义的操作码（如图4.1中定义的 0、1、2...）是一个正整数， $ARG_{ij}$  为  $OP_i$  的参数代表  $OP_i$  需要对 PyCodeObject 中其他域（如 `co_nlocals`、`co_consts`）中的第几个对象执行操作也是正整数， $n$  为一个字节码文件所包含的操作码总数。

$OP_1$	$ARG_{11}$	$ARG_{12}$	$OP_2$	$ARG_{21}$	$ARG_{22}$	...	$OP_n$	$ARG_{n1}$	$ARG_{n2}$
--------	------------	------------	--------	------------	------------	-----	--------	------------	------------

图 4.2 操作码序列示意图

**定义 3 基本块：**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（例如 114 代表的 POP\_JUMP\_IF\_FALSE、115 代表的 POP\_JUMP\_IF\_TRUE、111 代表的 JUMP\_IF\_FALSE\_OR\_POP、112 代表的 JUMP\_IF\_TRUE\_OR\_POP）、绝对跳转操作（110 代表的 JUMP\_FORWARD、113 代表的 JUMP\_ABSOLUTE）、控制相关的操作（119 代表的 CONTINUE\_LOOP、120 代表的 SETUP\_LOOP、121 代表的 SETUP\_EXCEPT、122 代表的 SETUP\_FINALLY、80 代表的 BREAK\_LOOP），则这些连续出现的操作码则处于同一个基本块当中。

**定义 4 基本块信息：**一个操作码序列  $S$  的基本块信息  $B$  是一个长度与  $S$  中的操作码个数  $n$  相同的一个由正整数构成的单调不减序列， $B$  中的每个元素与  $S$  中的每个操作码一一对应，其值为其对应的操作码在所  $S$  中所在的基本块号。如表4.3所示，就是一个基本块信息，其代表在  $S$  中第 1~3 个操作码处于第一个基本块，第 4~7 个操作码处于第二个基本块...第  $n$  个操作码处于第  $k$  个基本块。

1	1	1	2	2	2	2	...	k	k
---	---	---	---	---	---	---	-----	---	---

图 4.3 基本块信息示意图

## 4.2 操作码替换的设计

操作码替换其实质为将字节码文件（.pyc）包含的操作码序列（co\_code）中的操作码用新的数值进行替换，使其包含新的操作码，从而改变操作码序列的内容，达到防逆转的目的。其形式化表示如下：

Python 虚拟机规范中定义的  $n$  个操作的集合为公式4.2所示的集合：

$$OP = \{op_1, op_2, \dots, op_n\} \quad (4.1)$$

公式中  $op_i$  是带有特定语义信息的字符序列，表示第  $i$  个虚拟机所支持的操作， $n$  为 Python 虚拟机所支持的操作总数， $1 \leq i \leq n$ 。

与集合  $OP$  对应的操作码集合为公式4.2所示的集合：

$$OP\_CODE = \{code_1, code_2, \dots, code_n\} \quad (4.2)$$

公式4.2中  $code_i$  与  $op_i$  一一对应，为  $op_i$  在的数值表示形式， $code_i \in \mathbb{N}^+$  且  $code_i \leq 255$ ， $1 \leq i \leq n$ 。即集合  $OP$  与集合  $OP\_CODE$  之间的关系可用如公式4.3所示  $f(x)$  表示：

$$f(OP_i) = code_i \quad (4.3)$$

其中  $OP_i \in OP$ ， $code_i \in OP\_CODE$ ， $1 \leq i \leq n$ 。

进行操作码替换的实质则为使用集合  $OP\_CODE' = \{code'_1, code'_2, \dots, code'_n\}$  来替换集合  $OP\_CODE$ ，使  $OP\_CODE'$  至少存在一个元素  $code'_i \neq code_i$ ， $1 \leq i \leq n$ 。即用  $g(x)$  来代替  $f(x)$ ，使得公式4.4成立：

$$g(OP_i) = code'_i \quad (4.4)$$

在公式4.4中  $OP_i \in OP$ ， $code'_i \in OP\_CODE'$ ， $1 \leq i \leq n$ 。

操作码替换过程则如图4.4，本文提出的操作码替换算法主要分为2个部分，如图4.5所示：

寻找操作码编译和解释执行规律是为了理清楚操作码编译和解释执行之间存在的关系，不能盲目随意的对操作码进行替换。打乱操作码映射关系则是使用新的操作码值集合来替换原来的操作码值集合，达到改变操作码序列内容的目的。

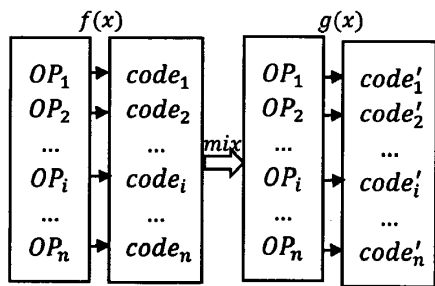


图 4.4 操作码替换示意图

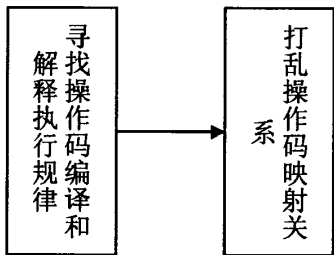


图 4.5 操作码替换设计

4.2.1 寻找操作码编译和解释执行规律

欲通过对字节码文件中包含的操作码的值进行替换来达到防逆转的目的，就必须弄清楚操作码编译和解释执行之间存在的关系。通过对 Python 操作码编译的解释执行规律的跟踪与分析，发现在 Python 版本 2.7.9 所支持的操作码 (Include/opcode.h) 集中，并不是所有的操作码都是可以进行随机替换的（因为操作码只占一个字节的大小，所以其替换前后的操作码值 *code* 都属于 0~255）。其规律如图4.6所示：

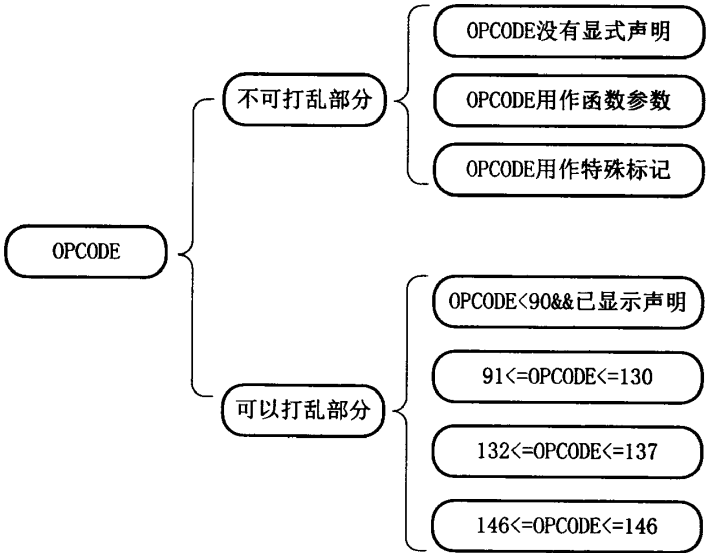


图 4.6 操作码替换的分类

从图4.6中可以看到，操作码分为可替换部分和不可替换部分两类。从原理上讲，Python 所支持的所有操作码都是可以进行替换的，但若将某些操作码替换之后还要对 Python 虚拟机进行较大的调整，耗时耗力，很不可取。所以们这里将操作码分为不可替换和可以替换两类。

不可打乱部分的操作码又分为一下 4 类：

(1) 操作没有显示声明：这些操作没有在 Python 操作码定义表中显示声明

出来,但实际中会在字节码文件编译过程中生成,并在解释阶段执行。这些操作和其对应的操作码分别为 (SLICE +1 31), (SLICE +2 32), (SLICE +3 33); (STORE\_SLICE +1 41), (STORE\_SLICE +2 42), (STORE\_SLICE +3 43); (DELETE\_SLICE +1 51), (DELETE\_SLICE +2 52), (DELETE\_SLICE +3 53)。

(2) 用作函数参数: Python 中的部分操作对应的操作码只差有用作函数调用时的参数,这些参数会直接决定函数在栈空间上读取到的数据。所以最好不要对这些操作码值进行替换。这些操作和其对应的操作码分别为 (CALL\_FUNCTION 131), (CALL\_FUNCTION\_VAR 140), (CALL\_FUNCTION\_KW 141), (CALL\_FUNCTION\_VAR\_KW 142)。

(3) 用作特殊标记: 还有部分操作对应的操作码是用作特殊标记的,所以也不应该对其进行替换。它是 (EXTENDED\_ARG 145)。

在保持上面不可打乱的操作和操作码值得对应关系的前提下,可以打乱部分的操作码可以分为 4 个部分,每个部分内部的操作其对应的操作码可以和同一类的其他操作对应的操作码两两随机替换。

(1)  $OPCODE < 90$  操作码已经显式声明: 都是简单的不带参数的操作,同一类内部可以随机打乱,对其值进行打乱不会影响 Python 程序的执行。

(2)  $90 \leq OPCODE \leq 131$ : 带参数的操作码,和函数调用无关,内部可以打乱。

(3)  $131 \leq OPCODE \leq 137$ : 带参数的操作码,和函数调用相关的部分,内部可以随机打乱。

(4)  $146 \leq OPCODE \leq 147$ : 带参数的操作码,两个扩展操作码内部可以随机打乱。

## 4.2.2 打乱操作码映射关系

对操作码映射关系进行打乱,如图4.4所示,即使用新的映射关系  $g(x)$  来代替  $f(x)$  从而使操作码序列发生如图4.7所示的变换。这里需要分 3 不进行:

$$S = [code_1, arg_{11}, arg_{12}, code_2, arg_{21}, arg_{22}, \dots, code_n, arg_{n1}, arg_{n2}]$$



$$S' = [code'_1, arg_{11}, arg_{12}, code'_2, arg_{21}, arg_{22}, \dots, code'_n, arg_{n1}, arg_{n2}]$$

图 4.7 操作码替换前后操作码序列变化示意

(1) 改变 Python 操作码映射表: 在上面可以打乱部分的 4 类操作码在同一类中对 Python 操作码映射表 (Include/opcode.h) 进行随机替换 (可以使用随机

```

1  /* Instruction opcodes for compiled code */
2
3  #define STOP_CODE 74
4  #define POP_TOP 16
5  #define ROT_TWO 12
6  #define ROT_THREE 18
7  #define DUP_TOP 4
8  #define ROT_FOUR 37
9  #define NOP 28
10 ...

```

图 4.8 操作码替换示例

洗牌的方法)。例如对图 4.1 中所示的操作码映射关系可以进行如图 4.8 所示的替换：

(2) 生成新的 Python 运行环境：为能够将新的操作码映射关系写入到字节码文件中，需要对操作码进行替换后需要重新编译 Python 源码文件生成新的 Python 编程语言开发的源码文件（.py）运行环境 new\_python。

(3) 使用新的 Python 运行环境 new\_python 编译使用 Python 编程语言开发的源码文件（.py）生成字节码文件（.pyc）。

这里生成字节码文件（.pyc）即包含新的操作码映射关系，也只能被运行环境 new\_python 所识别和运行。其他环境无法运行，反编译工具也无法对其进行反编译。

### 4.3 操作码合并的设计

本文提出的操作码合并<sup>[32]</sup>算法主要分为 4 个部分，如图 4.9 所示。图中，提

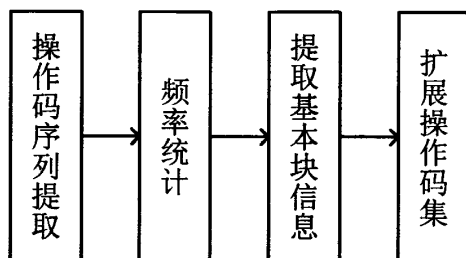


图 4.9 操作码合并算法流程

取操作码序列与统计频率的目的是寻找待合并的操作码序列。提取基本块信息是为了从大量的待合并操作码序列中选出可以合并的操作码序列，减少对扩展操作码集的干扰。扩展操作码集是添加新操作码的定义、与解释执行。



### 4.3.1 提取操作码信息

由于欲使采取操作码合并算法需要适用于不同的 Python 应用程序，使它们在合并之后的操作码序列结构较合并前都发生较大变化，所以为了算法的通用性，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是合理的做法。实际中需要从下面 3 步进行：

(1) 寻找大量的 Python 编程语言编写的应用程序源码文件并组成公式4.5所示的  $SET_{source\_file}$  集合：

$$SET_{source\_file} = \{file^1.py, file^2.py, \dots, file^k.py\} \quad (4.5)$$

(2) 将集合  $SET_{source\_file}$  中的每个源码文件利用 Python 编译模块进行编译，编译生成的字节码文件组成公式4.6所示的  $SET_{bytecode\_file}$  集合：

$$SET_{bytecode\_file} = \{file^1.pyc, file^2.pyc, \dots, file^k.pyc\} \quad (4.6)$$

(3) 从公式4.6所示的集合  $SET_{bytecode\_file}$  中提取每个字节码文件  $file^i.pyc$  包含如公式4.7的操作码序列  $S^x$ ：

$$S^x = [OP_1^x, ARG_{11}^x, ARG_{12}^x, \dots, OP_n^x, ARG_{n1}^x, ARG_{n2}^x] \quad (4.7)$$

在上面 3 步中的变量  $x$  和  $k$  满足  $1 \leq x \leq k$  且  $100 \ll k$ ， $n$  为操作码序列  $S^x$  包含的操作码个数。

本文提取的操作码序列为 Python/Lib 下的 1950 个库文件源程序对应的 1950 个操作码序列。这些程序通常提供接口给其他模块导入，功能多样且不会随意发生改变，所以选它们进行操作码序列提取较为合理，其中最长的操作码序列包含 10000 多个操作码，最短的也包含 60 个操作码，平均包含为 200 个的操作码，共出现操作码 240263 个。

### 4.3.2 频率统计

统计频率的目的是获取候选合并对象。在上一小节获取的多个操作码序列  $SET_{bytecode\_file}$  的基础上，统计出  $SET_{bytecode\_file}$  中的每个操作码序列  $S^x$  包含的所有长度为  $len$  的操作码子序列出现的次数，并按出现次数从高到底排序，其中  $len$  满足公式4.8。

$$len \in \{len \in \mathbf{N}^+ \cap 2 \leq len\} \quad (4.8)$$

表 4.1 部分长度为 2 的操作码子序列以及出现次数

$OP_1$	$OP_1$ 的值	$OP_2$	$OP_2$ 的值	成对出现次数
LOAD_CONST	100	MAKE_FUNCTION	132	9269
LOAD_CONST	100	IMPORT_NAME	108	7762
BUILD_CLASS	89	STRORE_NAME	90	6285
IMPORT_FROM	109	STORE_NAME	90	3732
CALL_FUCNTION	131	POP_TOP	1	2871

### 4.3.3 提取基本块信息

基本块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。从上一小节获取的结果其 TOP N 中的结果可能并不都是可以合并的操作码子序列，因为有些子序列中的操作码可能并不是处于同一个基本块中，如果将处于不同基本块中的操作码合并将会破坏程序的执行逻辑。所以需要进一步在窥孔优化模块的基础上（peephole 模块）根据程序执行逻辑提取公式 4.7 所示操作码序列  $S^x$  的如公式 4.9 基所示的本块信息  $B^x$ ：

$$B^x = [VAL_1^x, VAL_2^x, \dots, VAL_n^x] \quad (4.9)$$

其中  $n$  为操作码序列  $S^x$  包含的操作码个数。

若  $B^x$  中存在满足公式 4.10 的情况：

$$VAL_i^x = VAL_{i+1}^x = VAL_{i+2}^x = \dots = VAL_{i+j}^x \quad (4.10)$$

则在  $S^x$  中  $OP_i^x, OP_{i+1}^x, OP_{i+2}^x, \dots, OP_{i+j}^x$  是处于同一个基本块中，是一条可以合并的操作码子序列。

如表 4.1 所示，是上述 1950 个操作码序列中部分长度为 2 且处于同一个基本块中的操作码子序列以及出现的次数。

### 4.3.4 扩充操作码集

扩充操作码集就是利用窥孔优化将出现频率较高且处于同一个基本块中的操作码子序列  $(OP_1, OP_2, \dots, OP_{len})$  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  $OP_1, OP_2, \dots, OP_{len}$  的效果等价，完成对  $OP_{new}$  的解释执行。如图 4.10 所示，利用操作码合并算法可以将上表中的两个操作码  $OP_1$  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  $S_{new}^x$ ，图 4.10 中仅展示了对两个

带参数的操作码的合并过程，若要对多个操作码进行合并也是采用类似的方法，不过需要新的操作码带跟多的参数。此时在  $S_{new}^x$  中  $OP_{new}$  将带 2 个参数，所以在解释  $OP_{new}$  时需要一次将 2 个参数都读出来。因为在 Python 字节码序列中任

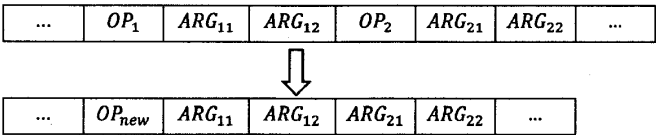


图 4.10 操作码合并示意图

意一个操作码仅占一个字节的大小，所以在不改变操作码所占空间大小的情况下最多可以定义 256 个操作码。而在 Python 2.7.9 版本的虚拟机规范已经使用了 256 个中的 110 个，所以可以利用剩余 146 个值来定义新操作码，从利用而用操作码合并缩短操作码序列的同时隐藏操作码序列的含义。本文就是对表 4.1 中出现的 5 个子序列进行了合并，并利用剩余 146 个值中的 5 个值来表示表 4.1 中出现的 5 个子序列合并之后的  $OP_{new}$ 。

而 Python 3 仅在 Python 2 的基础上对某些语法和模块做了调整和改动，同样遵循操作码占一个字节的特性和依次对操作码序列进行解释执行的机制，所以不会影响操作码合并算法在 Python3 上的实现。例如 Python 3.6.1 使用了 256 个值中的 119 个，所以只能利用剩余的 137 个值来定义新操作码。

在操作码仅占一个字节的基础上进行操作码合并，会大大限制操作码合并的扩展的空间。所以若要生成和使用更多的操作码，则必须通过增加操作码所占存储空间的大小（将操作码大小全部设为 2 个字节或将部分操作码的大小设为 2 个字节），但这样在执行阶段可能会因多次读取操作码导致程序执行效率下降。

4.4 操作码合并示例

这一小节中，会利用到第 3 章中对字节码文件介绍的部分内容，这一小节将针对具体代码，展示操作码合并算法的具体实现过程。

例如有如代码 4.11 所示的使用 Python 编程语言编写的 test.py 源码文件，下面本小节将以 test.py 为源码为示例，按图 4.9 中展示的 4 个步骤进行操作码合并转换演示。

4.4.1 获取段代码的操作码信息

利用 Python 编译模块（py\_compile 或者 compileall）将上述源代码文件 test.py 编译为字节码文件 test.pyc。提取出 test.pyc 文件对应的 PyCodeObject 对

```

1
2 import sys
3
4 def func1():
5     print sys.path
6 if __name__=='__main__':
7     func1()

```

图 4.11 test.py 示例代码

象中 `co_code` 域（实质为 `PyStringObject` 对象）包含的字符串，这个字符串就是操作码合并需要的 `test.py` 源码文件包含的操作码序列  $S^{test.pyc}$ 。

$$\begin{aligned}
 S^{test.pyc} = [ & 100\ 0\ 0\ \mathbf{100}\ \mathbf{1}\ 0\ \mathbf{108}\ 0\ 0\ 90\ 0\ 0\ \mathbf{100}\ \mathbf{2}\ 0\ \mathbf{132}\ 0\ 0\ 90\ 1\ 0 \\
 & 101\ 2\ 0\ 100\ 3\ 0\ 107\ 2\ 0\ 114\ 43\ 0\ 101\ 1\ 0\ 131\ 0\ 0\ 1\ 110\ 0\ 0 \\
 & 100\ 1\ 0\ 83 ]
 \end{aligned} \quad (4.11)$$

#### 4.4.2 频率统计

因为这里只是对单个文件的操作码序列进行合并，其中仅包含一条操作码序列，所以其操作码子序列的频率统计没有参考价值，更没有必要对其进行统计。但可以根据在其他字节码文件中出现的次数较高的操作码子序列，表4.1中已经统计好的结果，找到其中可以用于合并的操作码子序列  $S_{sub1}^{test.pyc}$  和  $S_{sub2}^{test.pyc}$ 。如图4.12和图4.13所示，即是图4.11中所示的代码编译生成的字节码文件包含且在表4.1中出现的2个的操作码子序列，在公式4.11中已加粗表示。

...	100	1	0	108	0	0	...
-----	-----	---	---	-----	---	---	-----

图 4.12 频率较高的操作码序列 1

...	100	2	0	132	0	0	...
-----	-----	---	---	-----	---	---	-----

图 4.13 频率较高的操作码序列 2

#### 4.4.3 获取基本块信息

在 Python 窥孔优化模块（`peephole` 模块）的基础上，提取出  $S^{test.pyc}$  对应的基本信息  $B^{test.pyc}$ 。

$$B^{test.pyc} = [00\ 000\ 000000000000011] \quad (4.12)$$

从公式4.12所示的基本块信息与公式4.11所示的操作码序列的对应关系（即公式4.11与公式4.12中加粗的部分的对应关系）中可以发现操作码子序列  $S_{sub1}^{test.pyc}$  和  $S_{sub2}^{test.pyc}$  在字节码文件 test.pyc 中是同处于第0个基本块中的。也就是说,  $S_{sub1}^{test.pyc}$  包含的两个操作 LOAD\_CONST 和 IMPORT\_NAME 在解释执行是按照前后顺序依次执行的，是一对可以合并的操作码；同样  $S_{sub2}^{test.pyc}$  也是一对可以合并的操作码子序列，因为它包含的两个操作 LOAD\_CONST 和 MAKE\_FUNCTION 在解释执行也是按照前后顺序依次执行的。

到目前为止，已经在图4.11中所示的代码编译生成的字节码文件 test.pyc 中找到了两对可以合并的操作码子序列 (LOAD\_CONST 和 IMPORT\_NAME), (LOAD\_CONST 和 MAKE\_FUNCTION)。

#### 4.4.4 扩充操作码集

这里需要对上面选定的操作码子序列  $S_{sub1}^{test.pyc}=[100\ 1\ 0\ 108\ 0\ 0]$  和  $S_{sub2}^{test.pyc}=[100\ 2\ 0\ 132\ 0\ 0]$  进行合并。

如图4.14所示，可以将  $S_{sub1}^{test.pyc}=[100\ 1\ 0\ 108\ 0\ 0]$  合并为  $S_{new1}^{test.pyc}=[181\ 1\ 0\ 0\ 0]$ ，将  $S_{sub2}^{test.pyc}=[100\ 2\ 0\ 132\ 0\ 0]$  合并为  $S_{new2}^{test.pyc}=[180\ 2\ 0\ 0\ 0]$ 。此时  $S_{new1}^{test.pyc}$  和  $S_{new2}^{test.pyc}$  将都带2个参数（每个参数2个字节）。

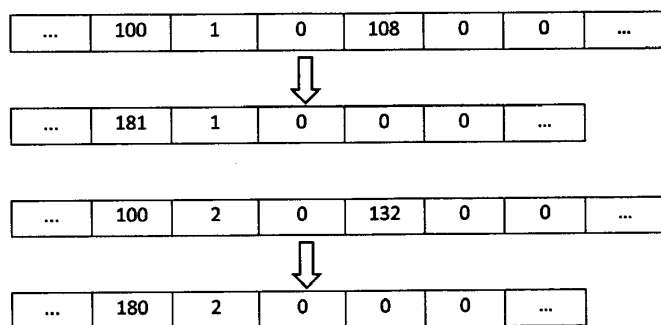


图 4.14 操作码合并实例图

接着在在 Python 虚拟机中添加对  $S_{new1}^{test.pyc}=[181\ 1\ 0\ 0\ 0]$ ，使得对  $S_{new1}^{test.pyc}=[181\ 1\ 0\ 0]$  的解释执行效果与依次执行  $S_{sub1}^{test.pyc}=[100\ 1\ 0\ 108\ 0\ 0]$  的效果等价；添加对  $S_{new2}^{test.pyc}$  的解释执行过程，使得对  $S_{new2}^{test.pyc}=[180\ 2\ 0\ 0\ 0]$  的解释执行效果与依次执行  $S_{sub1}^{test.pyc}=[100\ 1\ 0\ 108\ 0\ 0]$  的效果等价，即可完成对源码文件 test.py 的操作码合并过程。

#### 4.4.5 操作码序列合并前后比较

通过以上4个步骤,即可完成对代码4.11对应的字节码文件 test.pyc 操作码合并过程,此时重新编译代码4.11生成的字节码文件的z中包含的新的操作码序列和对应的基本块信息分别如公式4.13和4.14所示。

$$S_{new}^{test.pyc} = [100\ 0\ 0\ 181\ 1\ 0\ 0\ 0\ 90\ 0\ 0\ 180\ 2\ 0\ 0\ 0\ 90\ 1\ 0\ 101\ 2\ 0\ 100\ 3\ 0\ 107\ 2\ 0\ 114\ 43\ 0\ 101\ 1\ 0\ 131\ 0\ 0\ 1\ 110\ 0\ 0\ 100\ 1\ 0\ 83] \quad (4.13)$$

$$B_{new}^{test.pyc} = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1] \quad (4.14)$$

比较公式4.13和公式4.11中的操作码序列可以看出,通过操作合并使得图4.11中的代码对应的操作码序列比原来减少了两个操作码,同时长度较原来缩短了两个字节的。从第5章的分析和实验中将可以看到,通过操作码合并减少操作码序列中操作码的个数不仅可以达到防逆转的目的,而且有利于减小字节码文件的大小节约存储空间,更有利于提高 Python 字节码文件的执行效率。此时,在  $S_{new1}^{test.pyc}$  和  $S_{new2}^{test.pyc}$  的操作码 181 和 180 都将带 2 个参数(每个参数 2 个字节),所以 Python 虚拟机在每次执行在解释  $S_{new1}^{test.pyc}$  和  $S_{new2}^{test.pyc}$  时中的操作码时需要一次将 2 个参数都读出来。

### 4.5 本章小结

本章对基于操作码替换策略和操作码合并策略的 Python 字节码文件防逆转方式的设计与实现结合特定的代码为实例分别做详细介绍,并指出在操作码替换时需要保留部分操作码映射关系不能随意替换负责会对程序的执行结果造成严重的影响;在进行操作码合并时,需要在同一个基本块内进行合并不能跨基本块合并,否则也会对程序的执行逻辑和结果造成严重的影响。

## 第5章 操作码替换与合并方式的性能评测

在本章中，将对操作码替换与合并所能达到的理论安全性与执行效率进行分析与实验。

这里假设 Python 源码文件 *file.py* 在操作码替换与合并之前编译生成的字节码文件为 *file.pyc*，其中包含的操作码序列为  $S$ ， $n$  和  $m$  分别表示  $S$  中共包含的操作码个数和参数个数， $N$  表示 Python 虚拟机规范中定义的基础操作码个数。在操作码替换时，总共替换了  $repalce$  个操作码；此时重新编译生成的字节码文件为 *file<sub>r</sub>.pyc*，其中包含的操作码序列为  $S_r$ ， $n_r$  和  $m_r$  分别表示  $S_r$  中共包含的操作码个数和参数个数。在进行操作码合时，总共合并了  $folder$  个操作码序列，也就是重新生成了  $folder$  个新操作码；此时重新并后编译生成的字节码文件 *file<sub>f</sub>.pyc*，其中包含的操作码序列为  $S_f$ ， $n_f$  和  $m_f$  分别表示  $S_f$  中共包含的操作码个数和参数个数； $g(folder)$  表示  $S$  与  $S_f$  的长度之差，也就是因操作码合并导致  $S$  减小的长度； $h(folder)$  表示  $S_f$  包含的新操作码个数。

### 5.1 操作码替换性能评测

#### 5.1.1 操作码替换安全性分析

由于操作码替换技术是对每个操作码值用另一个操作码的值进行替换，来对 Python 编译器和解释同时进行订制，使得编译生成的字节码文件对于标准的解释器来说包含非法的操作码序列，来达到防逆转的目的，该技术已分别被基于虚拟机 Dropbox<sup>[33]</sup> 的应用，和基于 PC 的应用通过操作码随机化的安全<sup>[34]</sup> 中实现，并且网络上有现成的工具 python-obfuscation<sup>[35]</sup> 来对 Python 操作码进行随机替换。所以操作码替换技术其实质为利用密码学中的单表代替密码（也称单表置换密码）来对  $S$  中的操作码进行加密，从而将  $S$  转换为  $S_r$ 。

由于一般包含  $k$  个元素的集合的置换有  $k!$  个。例如，当操作码集合  $S = \{\text{LOAD}, \text{MOVE}, \text{ADD}\}$  时（分别代表数据读取、数据移动、数据求和操作）， $S$  此时有 3 个元素，则  $S$  的 6 个置换分别为：

```
{LOAD,MOV,ADD}
{LOAD,ADD,MOV}
{MOV,LOAD,ADD}
{MOV,ADD,LOAD}
{ADD,LOAD,MOV}
```

{ADD,MOV,LOAD}

所以，如果操作码替换之后的密文是 110 个操作码（Python2.7.9 有 110 个操作码）的任意置换，那么就有 110! 或大于  $1.58 \times 10^{178}$  种可能的密钥，这比 DES（Data Encryption Standard）的密钥空间（DES 64 位的密钥，强度大约为  $1.84 \times 10^{19}$ ）还要大 160 个数量级。

5.1.2 操作码替换实验检验

表面上看，由于 Python 所支持的操作码的个数  $N$  很多，导致操作码替换技术能够对字节码文件提供的密钥空间非常大。似乎操作码替换技术可以为字节码文件在解释器层次上提供一定的保护，对逆向分析增加了一定的难度。但是它的缺点也很明显，如果攻击者了解本文对字节码文件采取了基于操作码替换的保护方式，他就可以利用操作码的一些规律进行分析，采用统计攻击与比较分析。例如攻击者可以利用统计攻击只需要 4 步即可从替换后的字节码文件中提取出源码。

(1) 统计操作码使用频率：寻找大量的字节码文件，统计其中每个操作码的使用频率。如图5.1所示，是 Python2.7.9/Lib 下 1900 个源码文件（.py）经过编译之后的字节码文件（.pyc）中包含的操作码相对使用频率曲线：

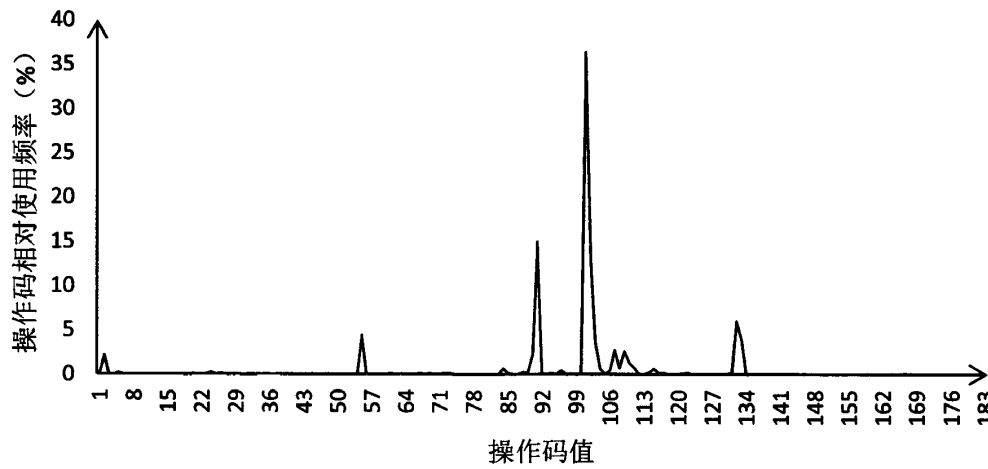


图 5.1 操作码相对使用频率曲线

(2) 统计欲破译的字节码文件中包含的操作码使用频率：统计欲破译的目标字节码文件中包含的操作码使用频率。如图5.2所示，是对 Django 下 800 个使用 Python 编程语言开发的源码文件（.py）使用某种操作码替换方案编译生成的字节码文件（.pyc）包含的操作码相对频率曲线：



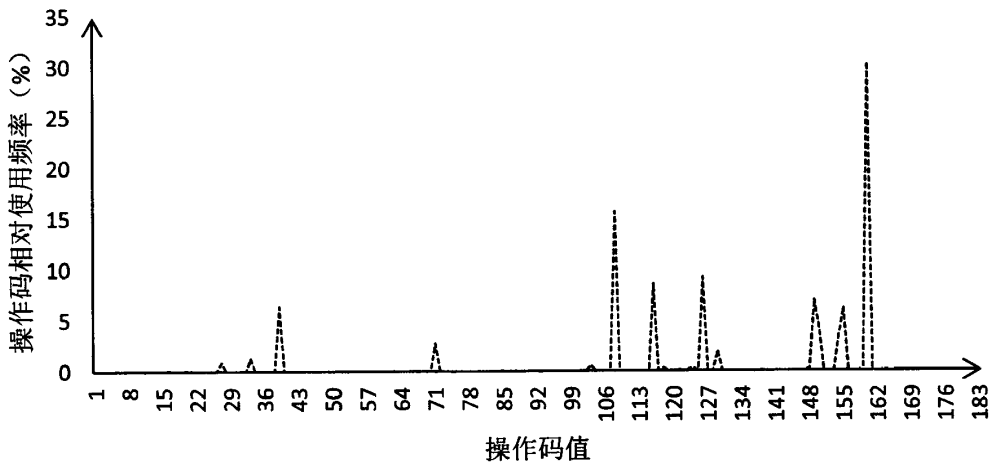


图 5.2 破译目标操作码相对使用频率曲线

(3) 进行比较分析：利用上面两步得到的数据进行比较，以期从中分析操作码替换关系，也就求解操作码替换时对  $S$  所采用的置换方案。如图5.3所示，可以对操作码替换之后的频率曲线与相对频率进行分析比较：

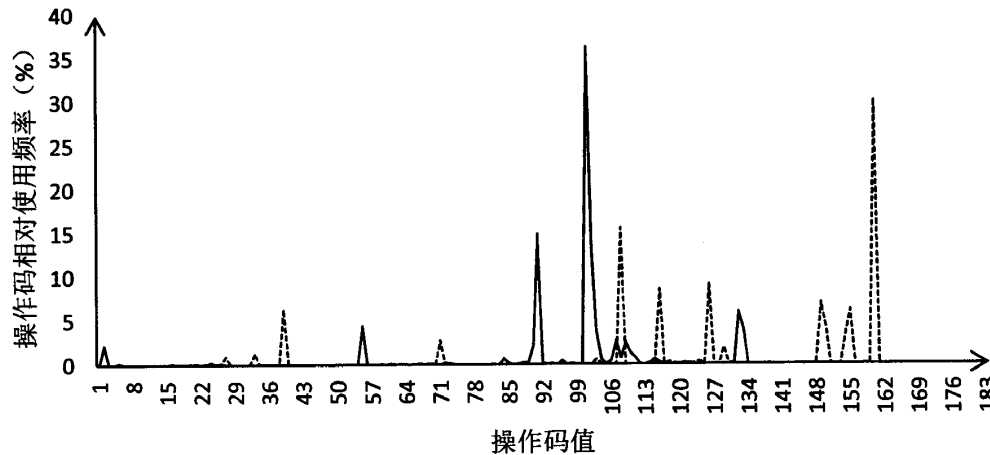


图 5.3 相对频率比较曲线

从图5.3中，可以看出，这种操作码映射关系将操作码 100 替换为 159，将操作码 90 替换为 107...

(4) 将目标中操作码值按照上述关系进行反向替换，然后使用反编译工具，既可以破译出目标字节码应用中包含的 python 源码。

所以，虽然操作码替换方案可以为 Python 字节码文件提供一定程度的保护，但其仍然较容易被攻破，因为它带有原始操作码使用频率的一些统计学特性。

因为该技术对 Python 的操作码映射关系进行了修改，需要在发布应用时同时将替换之后生成 Python 解释器发布，所以攻击者除了可以利用统计分析之外，

还可以对替换之后的解释器进行逆向分析,来获得操作码替换关系。攻击者可以通过比较分析标准的 Python 解释器与操作码替换之后的解释器的反汇编之后的内容来获取操作替换方案,然后再重复上面第4步内容也可以获取到操作码替换之后编译生成的字节码文件中包含的源码信息。很显然这个方法将会是非常耗时且不是一个一劳永逸的方法,如果下次采用不同的操作码映射方式,则又要重新进行比较分析。目前在 PyREtic<sup>[36]</sup>中提到一种破译操作码替换的方法,并且在 dropboxdex<sup>[37]</sup>中已部分采用。

## 5.2 操作码合并性能评测

### 5.2.1 安全性分析

与操作码替换相比,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,不仅削弱了字节码的统计学特性而且改变了操作码序列的长度和结构 (codesize),所以安全性比操作码替换高。如图4.10所示,因为操作码合并算法将操作码序列  $S$  中的连续出现的多个操作码合并为一个新操作码  $OP_{new}$ ,从而将  $S$  转换为  $S_f$ ,此时  $S_f$  隐藏了  $S$  中的多个操作码和其频率特征,取而代之的则是新操作码  $OP_{new}$  的值和频率特征。若要从  $S_f$  中破译出对应的操作信息,除了要分析  $S_f$  的逻辑结构之外,还有统计  $OP_{new}$  的频率特征,而  $OP_{new}$  的频率既不是  $S$  中原来多个操作码的频率之和,也不是多个操作码的频率的最小值,而是连续出现的操作码作为一个“单词”的相对频率,所以攻击者除了要知道我们合并的操作码序列外,还需要知道整个“单词”的相对频率,安全性大大提高。

理论上如果有耐心编写合并过程和解释过程支持  $OP_{new}$  的生成与执行的话,可以对操作码序列中的任意多个操作码进行合并,共有公式5.1中  $total(N)$  种可能的合并方式,也就说操作码合并的密钥空间大小为  $total(N)$ 。

$$total(N) = A_N^2 + A_N^3 + \dots + A_N^N \quad (5.1)$$

所以可能的操作码合并方式总数远大于指数函数,在 Python2.7.9 中  $N = 110$ 。攻击者要确定新操作码序列中包含多少个新操作码的复杂度为指数函数,安全性足够高。但一方面由于一些操作码子序列的排列根本不会出现,即使出现的操作码子序列若其中操作码处于不同基本块也不能合并,所以达不到这么多中可能的合并方式;另一方面本文利用了操作码仅占一个字节的特性,所以这里仅能使用 256 个中剩余的 146 个来定义新操作码。

公式5.2给出了操作码合并的强度  $O$ ，也就是利用穷举攻击从  $S_f$  恢复出  $S$  的需要穷举的操作码序列个数：

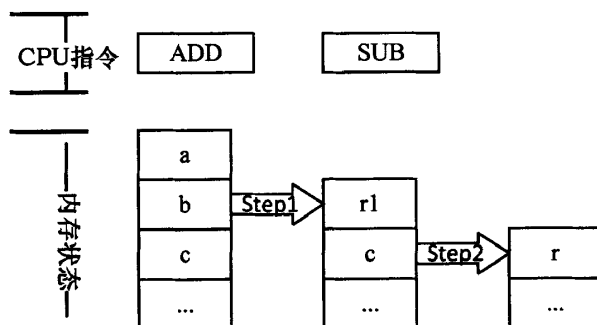
$$O = \min(256^{n+2m}, f) \quad (5.2)$$

其中  $f = N_{n+2m-g(folder)}^{C_g(folder)} \times N_{n+2m}^{C_h(folder)}$ ， $1 \leq folder \leq 146$ 。

## 5.2.2 执行效率分析

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。

操作码合并之前的执行过程



操作码合并之后执行过程

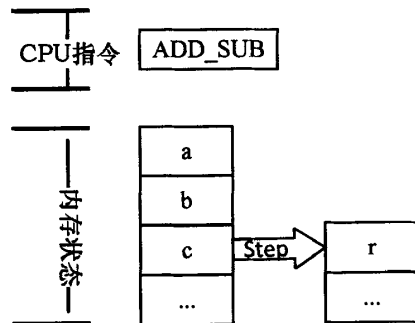


图 5.4 操作码合并前后执行过程对比图

如图5.4上部分所示，当在操作码序列中包含 ADD、SUB 列操作是，则需要读取两次操作码和参数来对栈顶的两个元素  $a$  和  $b$  出栈、求和、将结果  $r_1$  压栈，然后将栈顶元素  $r_1$  与  $c$  出栈、求差、将结果  $r$  再压栈。公式5.3给出了操作码合并之前 Python 虚拟机执行源码文件  $file.py$  的时间，其中  $C$  表示文件  $file.py$  编译所花费的时间， $read_i$  和  $exec_i$  分别表示虚拟机读取和解释操作码序列  $S$  中的

每个操作码所花费的时间，其中  $1 \leq i \leq n$ 。

$$t_b = C + \sum_{i=1}^n (read_i + exec_i) \quad (5.3)$$

操作码合并之后，如图5.4下部分所示，用新操作码 `ADD_SUB` 来代替上面的两个操作码，用它完成一次将栈顶的 3 个元素  $a$ 、 $b$ 、 $c$  都出栈，并将  $a + b - c$  的结果  $r$  运算完毕之后再再将  $r$  压栈。这里虽然需要以增加编译源码文件的时间为代价，但新的运算方式减少了冗余操的读取操作码的次数和修改 Python 虚拟机内部状态的次数，并且编译之后的字节码文件以后再次运行时，无需再次编译，提升了字节码文件的执行效率。

公式5.4给出了操作码合并之后 Python 源码文件 *file.py* 的执行时间，其中  $C_\Delta$  表示合并操作码在编译阶段额外花费的时间， $\overline{read}_i$  和  $\overline{exec}_i$  分别为操作码合并之后每个操作码读取与执行时间。

$$t_a = C + C_\Delta + \sum_{i=1}^{n-g(folder)} (\overline{read}_i + \overline{exec}_i) \quad (5.4)$$

因为防逆转的目的最终是发布字节码文件 (.pyc) 给用户，所以这里只对字节码文件的执行效率进行讨论。操作码合并的整体性能与合并前后的字节码文件执行时间有关。公式5.5给出了操作码合并后字节码文件执行的变化率，分母表示操作码合并之前字节码文件的执行时间，分子表示操作码合并之后较合并前字节码文件执行时间的变化量。

$$\begin{aligned} t_\Delta &= \frac{(t_a - (C + C_\Delta)) - (t_b - C)}{t_b - C} \times 100\% \\ &= \left( \frac{t_a - (C + C_\Delta)}{t_b - C} - 1 \right) \times 100\% \\ &= \left( \frac{\sum_{i=1}^{n-g(folder)} (\overline{read}_i + \overline{exec}_i)}{\sum_{i=1}^n (read_i + exec_i)} - 1 \right) \times 100\% \end{aligned} \quad (5.5)$$

由公式 5.5 可以看出，当操作码合并前字节码文件的执行时间  $t_b - C$  恒定的情况下，合并之后的执行效率的变化率  $t_\Delta$  取决于新生成的字节码文件中操作码序列  $S_f$  的执行的的时间，而新的操作码序列的执行时间与操作码序列  $S_f$  中的操作码的数量成反比，与  $S_f$  中每个操作码的执行时间成正比，所以通过合并操作码来减少  $S_f$  中的操作码数量，或者缩短每个操作码的解释执行时间是提高程序

运行速度的重要方法。虽然要在不影响操作码序列执行结果的前提下,通过缩短操作码的执行时间来完成同样的任务难度非常大,但是通过合并操作码来减少操作码序列中的操作码个数是可以实现的。本文正是借此在实现防逆转的同时提高了字节码文件的运行效率。

### 5.2.3 存储空间分析

操作码合并将操作码序列  $S$  中的连续出现的多个操作码合并为一个操作码,在不减少  $S$  中参数的同时,减少了  $S$  中操作码的个数,缩短了操作码序列的长度。公式5.6给出了操作码合并后,重新生成的字节码文件的大小变换率。

$$\begin{aligned} s_{\Delta} &= \frac{S_r - S}{S} \times 100\% \\ &= \left( \frac{S_r}{S} - 1 \right) \times 100\% \\ &= -\frac{g(folder)}{n + 2m} \times 100\% \end{aligned} \quad (5.6)$$

从公式5.6中可以看出,操作码合并使得原本需要占用  $n + 2m$  个字节的操作码序列  $S$ ,变为只需要占用  $n + 2m - g(folder)$  个字节的  $S_f$ ,使编译生成的字节码文件的大小缩小了  $g(folder)$  个字节,所以通过增加操作码合并的数量或长度可以大大减小新生成的字节码文件的大小。所以操作码合并策略可以在需要存储和导入大量模块时有利于节约计算机磁盘和内存。

### 5.2.4 操作码合并实验

本节基于 Python2.7.9 版本,实验测试平台为 Intel Xeon E5-2690、主频 2.6GHz, 252G 内存,操作系统为 CentOS 6.7 64 位,合并的操作码为表 4.1 中出现的 5 对操作码。采用 microbenchmark<sup>[38]</sup> 来对操作码合并策略的安全性、执行效率、文件大小进行了检验。microbenchmark 包含对行数字运算,字符处理等多个方面的 Python 源码文件,是一个比较好的测试集。

(1) 安全性:由于通过操作码合并改变了操作码序列的结构,对原始的操作码序列中的操做序列信息进行了很好的隐藏。所以传统的反编译工具 uncompile2、Decompyle++ 等工具无法识别合并之后的操作码序列的结构,更无法理解新操作码包含的语义信息,所以无法对新的字节码文件进行反编译,如图5.5所示是使用 uncompile2 工具对图 4.11所示的代码编译生成的字节码文件反编译结果。

(2) 执行效率:操作码合并策略使用一个操作码来完成原来多个操作码完成的任务,减少了冗余的读取操作与修改虚拟机状态的次数,从而提升了运算效率。表5.1是操作码合并前后 microbenchmark 下的部分测试脚本编译后的运行时间和

```
# 2017.03.30 11:40:59 CST
### Can't uncompile test.pyc
Traceback (most recent call last):
  File "/usr/local/python-original/lib/python2.7/site-packages/uncompyle/__init__.py", line 188, in main
    uncompyle_file(infile, outstream, showasm, showast)
  File "/usr/local/python-original/lib/python2.7/site-packages/uncompyle/__init__.py", line 125, in uncompyle_file
    uncompyle(version, co, outstream, showasm, showast)
  File "/usr/local/python-original/lib/python2.7/site-packages/uncompyle/__init__.py", line 97, in uncompyle
    ast = walker.build_ast(tokens, customize)
  File "./build/lib/uncompyle/Walker.py", line 1234, in build_ast
AssertionError
# decompiled 0 files: 0 okay, 1 failed, 0 verify failed
# 2017.03.30 11:40:59 CST
```

图 5.5 反编译失败示例

变化率数据，这里仅列举了 microbenchmark 下的部分文件执行时间变化情况，test.pyc 图 4.11 中所示的代码编译所得的字节码文件（其中  $t_{\Delta} = \frac{t_a - t_b}{t_b} \times 100\%$ ）。

表 5.1 操作码合并前后字节码文件执行时间比较

文件名	合并前执行时间 $t_b$ (ms)	合并后执行时间 $t_a$ (ms)	执行时间变化率 $t_{\Delta}$
test.pyc	606	513	-15.35
attribute_lookup.pyc	1306	1210	-7.56
attrs.pyc	16505	16375	-0.79
closures.pyc	10610	10597	-0.12
contains_ubench.pyc	10953	10513	-4.02

从表5.1的执行的执行时间变化数据中可以看出，使用操作码合并策略可以使字节码文件中的操作码序列，在参数个数不变的前提下，减少了操作码的个数，从而减少了 CPU 读取操作码的次数，减少了多个操作码执行之间的跳转次数和修改虚拟机状态的次数，最终可以达到提升了程序的执行效率的目的。

从表5.1中可以看到通过操作码合并之后 test.pyc 和 pydigits.pyc 文件较之前有明显的减少。通过分析，发现原因为 test.pyc 中操作码序列中包含的操作码个数较少，只需要合并几个操作码，就可以使操作码个数有明显的减少；而 pydigits.pyc 运行时间明显减少，的原因是合并之前的操作码对包含在循环中，循环中的操作码被多次执行，使得新的操作码序列较整个执行过程有明显减少。

(3) 字节码文件大小：操作码合并策略用一个操作码代替原来两个或更多的操作码的语义信息，生成更短的操作码序列，更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更小。表5.2是操作码合并前后 microbenchmark 下的部分测试脚本编译后的文件大小和变化率数据，表5.2中仅列举了 mi-

crobenchmark 下的部分文件大小变化情况，test.pyc 为代码 4.11 编译所得的字节码文件。（其中  $s_{\Delta} = \frac{s_a - s_b}{s_b} \times 100\%$ ）。

表 5.2 操作码合并前后字节码文件大小比较

文件名	合并前文件大小 $s_b$ (byte)	合并后文件大小 $s_a$ (byte)	文件大小变化率 $s_{\Delta}$
test.pyc	1508	1498	-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

5.3 本章小结

本章对基于操作码替换与合策略的 Python 字节码文件放逆转方式进行了分别实验与分析，得出以下结论：

- （1）操作码替换与合并策略可以适用于 Python 字节码文件的防逆转；
- （2）通过操作码替换策略来改变 Python 字节码文件中操作码序列的内容，虽然可以达到防逆转的目的，密钥空间也很大（Python2.7.9 中约为 110!），但其保留了 Python 操作码的使用频率特征，因此安全性还有待加强；
- （3）通过操作码合并方式改变 Python 字节码文件中操作码序列的解构和内容，在达到防逆转目的的同时，大大缩短了字节码文件的长度，而这会带来提升 Python 字节码文件的执行效率，和减小字节码文件大小的优点。





## 第6章 全文总结

随着脚本编程语言的发展和普及,越来越多的个人和企业在进行编程或软件开发时都开始选择一种脚本编程语言作为主要开发语言。Python 编程语言做为众多脚本编程语言之一,已被广泛的应用于系统管理任务和 Web 编程,使用 Python 编程语言开发的程序源码文件首先需要经过 Python 编译模块将其编译为 Python 字节码文件才能够在 Python 虚拟机上运行。

但 Python 程序编译生成的字节码文件中包含了源码文件名、路径、常量、所有的符号、操作码序列等诸多信息导致字节码文件很容易被逆向工具反编译出为源码文件。而传统的代码混淆技术、文件加密技术、本地编译技术、数字水印技术等防逆转方法存在着处理后的字节码文件安全性不足、造成程序运行效率降低、或字节码文件存储空间增加等问题。

### 6.1 研究工作与成果

为此本文围绕基于操作码替换与合并的 Python 字节码文件防逆转策略展开研究工作,采用理论分析与实验验证相结合的方法,一方面从理论上分析了基于操作码替换与合并 Python 字节码文件防逆转策略会对字节码文件造成的影响,另一方面对这些影响建立有效的实验评价方案和实验环境,并进行实验验证。本文的主要的研究内容和成果包括以下三个方面:

(1) 通过对 Python 运行框架和 Python 字节码文件编译生成、执行执行机制的分析,根据 Python 虚拟机对字节码文件中的操作码逐一进行解释执行的特点,将 Python 字节码文件的核心内容 `co_code` 域的执行流程进行简化抽象,建立字节码文件的操作码序列与其基本块模型。

(2) 针对现有的代码混淆技术和数字水印技术安全性不足的问题,本文以字节码文件中的操作码序列为基础,结合单表替换密码,在不改变 Python 字节码文件执行正确性的前提下,设计出了一种适用于 Python 操作码的操作码替换策略。该策略通过操作码替换来改变操作码序列中操作码的值来达到改变操作码序列内容,最终达到防逆转的目的。最后对操作码替换在 Python 2.7.9 中予以实现,并根据单表替换密码的特性,利用操作码的统计学特性,对操作码替换前后的字节码文件中的操作码频率进行统计、比较分析,评估操作码所替换策略的安全性进行与实验。

(3) 针对文件加密技术易对字节码文件的执行效率造成影响和本地编译技术目标程序体积增加的问题,本文设计出了操作码合并策略。该策略对 Python

操作码集进行扩充，在不改变 Python 字节码文件执行结果和逻辑的前提下，以字节码文件中的操作码序列的基本块为基础，利用窥孔优化技术将处于同一个基本块中连续出现的多个操作码进行合并，使用新的操作码来代替原来操作码序列中连续出现的多个操作码。通过操作码合并改变了操作码序列的结构和内容，大大缩短了操作码序列的长度，达到防逆转的目的。最后对操作码合并并在 Python 2.7.9 中予以实现，并操作码合并策略产生的字节码文件的安全性、执行效率、以及文件大小进行评估与实验。

## 6.2 下一步工作

本文在基于操作码替换与合并的 Python 字节码文件防逆策略的设计上取得了一定的研究成果，但仍有一些工作可以进行改进或扩展。

基于操作码替换的 Python 字节码文件防逆转策略，保留了 Python 字节码文件操作码使用频率的特性，任然不是一个安全的保护策略，因此下一步工作将操作码替换策略进行改进，消除其频率特性，增加操作码替换策略的安全性。

操作码合并策略大大减少了操作码序列中操作码的个数，缩短了操作码序列的长度，在缩小了字节码文件大小的同时，减少了 Python 虚拟机读取操作码和修改虚拟机状态的次数，提升了 Python 字节码文件的执行效率的。然而目前操作码合并策略只能使用与同一个基本块中，不能在多基本块间进行合并，操作码合并策略仍然有很大的提升空间，下一步将考虑通过抽象语法树变换和优化，对多基本块之间的操作码合并进行实验。

## 参考文献

- [1] 脚本语言[EB/OL]. <https://zh.wikipedia.org/wiki/%e8%84%9a%e6%9c%ac%e8%af%ad%e8%a8%80>.
- [2] Python[EB/OL]. <https://zh.wikipedia.org/wiki/Python>.
- [3] Python 2.7 decompiler[EB/OL]. <https://github.com/wibiti/uncompyle2>.
- [4] python bytecode disassembler and decompiler[EB/OL]. <https://github.com/wibiti/uncompyle2>.
- [5] bytecode decompiler[EB/OL]. <https://sourceforge.net/projects/easypythondecompiler/>.
- [6] Collberg C S, Thomborson C. Watermarking, tamper-proofing, and obfuscation-tools for software protection[J]. IEEE Transactions on software engineering. 2002, 28 (8): 735–746.
- [7] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations[R]. [S.l.]: [s.n.], 1997.
- [8] A python package could import/run encrypted python scripts[EB/OL]. <https://pypi.python.org/pypi/pyarmor>.
- [9] A Python Distutils extension which converts Python scripts into executable Windows programs[EB/OL]. <http://www.py2exe.org/>.
- [10] J X. Digital watermarking and its application in image copyright protection[M]//[S.l.]: [s.n.].
- [11] Hamilton J D S. An evaluation of static java bytecode watermarking[M]//[S.l.]: [s.n.].
- [12] Kumar K K P, Kehar V. An evaluation of dynamic java bytecode software watermarking algorithms[M]//[S.l.]: [s.n.].
- [13] Myles G, Collberg C. Software watermarking via opaque predicates: Implementation, analysis, and attacks[J]. Electronic Commerce Research. 2006, 6 (2): 155–171.
- [14] 陈明奇, 钮心忻, 等. 数字水印的研究进展和应用[J]. 通信学报. 2001, 22 (5): 71–79.
- [15] Kuzurin N, Shokurov A, Varnovsky N, et al. On the concept of software obfuscation in computer security[C]//Springer, International Conference on Information Security. [S.l.]: Springer, 2007: 281–298.
- [16] 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.
- [17] Diffie W, Hellman M. New directions in cryptography[J]. IEEE transactions on Information Theory. 1976, 22 (6): 644–654.
- [18] 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.
- [19] 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.
- [20] 苏庆, 吴伟民, 李忠良, 等. 混沌不透明谓词在代码混淆中的研究与应用[J]. 计算机科学. 2013, 40 (6): 155–159.
- [21] Chan J T, Yang W. Advanced obfuscation techniques for java bytecode[J]. Journal of Systems and Software. 2004, 71 (1): 1–10.
- [22] Stallings W. 密码编码学与网络安全——原理与实践[M]. 北京: 电子工业出版社, 2014.
- [23] 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, 6: 034.
- [24] Weiss M, De Ferriere F, Delsart B, et al. Turboj, a java bytecode-to-native compiler[C]//Springer, Languages, Compilers, and Tools for Embedded Systems. [S.l.]: Springer, 1998: 119–130.
- [25] Chow S, Eisen P, Johnson H, et al. A white-box des implementation for drm applications[C]//Springer, ACM Workshop on Digital Rights Management. [S.l.]: Springer, 2002: 1–15.
- [26] Xuehua J. Digital watermarking and its application in image copyright protection[C]//IEEE, Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on: volume 2. [S.l.]: IEEE, 2010: 114–117.
- [27] Collberg C, Thomborson C. Software watermarking: Models and dynamic embeddings[C]//ACM, Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. [S.l.]: ACM, 1999: 311–324.
- [28] Zaidi S J H, Wang H. On the analysis of software watermarking[C]//IEEE, Software Technology and Engineering (ICSTE), 2010 2nd International Conference on: volume 1. [S.l.]: IEEE, 2010: V1–26.
- [29] 孙圣和, 陆哲明. 数字水印处理技术[J]. 电子学报. 2000, 28 (8): 85–90.
- [30] 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.
- [31] Ichisugi Y. Watermark for software and its insertion, attacking, evaluation and implementation methods[C]//Summer Symposium on Programming. [S.l.], 1997:

- 57–64.
- [32] Building an obfuscated Python interpreter: we need more opcodes[EB/OL]. <http://blog.quarkslab.com/building-an-obfuscated-python-interpreter-we-need-more-opcodes.html>.
  - [33] Kholia D, Węgrzyn P. Looking inside the (drop) box[C]//Presented as part of the 7th USENIX Workshop on Offensive Technologies. [S.l.], 2013.
  - [34] 微软公司. 通过操作码随机化的安全[P].
  - [35] python-obfuscation[EB/OL]. <https://github.com/citrusbyte/python-obfuscation>.
  - [36] Smith R. Pyretic: in memory reverse engineering for obfuscated python bytecode[C]//BlackHat/Defcon security conference. [S.l.], 2010.
  - [37] Dropbox Bytecode Decryption Tool[EB/OL]. <https://github.com/rumpeltux/dropboxdec>.
  - [38] microbenchmark[EB/OL]. <https://github.com/dropbox/pyston/tree/master/microbenchmarks>.



## 致 谢

在论文完成之际，三年的研究生生涯也将即将结束，心中也充满了无尽的感慨。深切的感受到个人的学习和成长离不开身边无数良师益友的教导与帮助，心中的感激之情实在是难以言表。

首先感谢我的导师顾乃杰教授三年来的循循善诱和殷殷教诲，能够顺利完成学业离不开顾老师的教导有方。顾老师忘一丝不苟的工作精神以及严谨负责的治学态度，让我受益匪浅。在学习和科研道路上，顾老师给了我很多锻炼的机会，使我的专业水平和科研能力有了很大的提高。在此，谨向尊敬的导师致以深深的敬意和感谢。

接着要感谢黄刘生、许胤龙、吴敏俊、熊焰、郑启龙、苗付友、陈恩红、李曦、张信明等老师，在我研一的时候教导我并传授知识，为我在日后的研究和工作的打下了良好的基础。

感谢实验室的任开新老师、黄章进老师以及曹华雄、杜云开、张旭、黄增士、张孝慈、林传文、陈雅莉等师兄的指点和照顾；感谢江国荐、Robail Yasrab、黄增士、王裕民、周文博、胡海强、刘帅、尹军、祝璞、陈露、刘思婷、王少萍、章润如、苏俊杰、谷德贺、刘博文、郑晓松、陈悟、李燚、王岩、郝建林、候津、王芬、陈思润、秦波、朱方祥、丁世举、刘廷建、浦丹、邵祎康、陈瑞、郑海波等师弟师妹，我们在网络计算与高效算法实验室共同学习生活，一起走过了这段难忘的岁月。

衷心的感谢远在家乡的父母和每一位亲人，感谢你们一直对我的支持和付出，是你们的爱与陪伴是鼓励我前行的坚强的后盾，仅在本文的最后向我远在家乡的每一位亲人表示感谢。

2017年3月31日





## 在读期间发表的学术论文与取得的研究成果

### 已发表论文

1. 王小强, 顾乃杰: 基于操作码合并的 Python 程序防逆转方法的研究与实现, 计算机工程, 已录用。