



(12) 发明专利申请

(10) 申请公布号 CN 105046117 A

(43) 申请公布日 2015. 11. 11

(21) 申请号 201510375384. X

(22) 申请日 2015. 06. 30

(71) 申请人 西北大学

地址 710069 陕西省西安市太白北路 229 号

(72) 发明人 李光辉 房鼎益 汤战勇 匡开圆  
陈晓江 郝朝辉 祁生德 樊如霞  
任庆峰 王蕾

(74) 专利代理机构 西安恒泰知识产权代理事务  
所 61216

代理人 王芳

(51) Int. Cl.

G06F 21/14(2013. 01)

G06F 21/12(2013. 01)

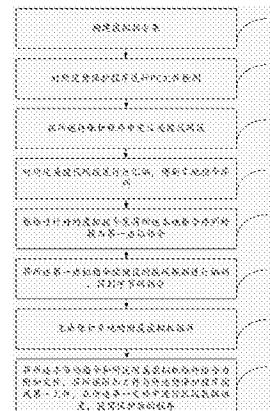
权利要求书2页 说明书13页 附图3页

(54) 发明名称

一种具有指令集随机化的代码虚拟化软件保  
护系统

(57) 摘要

本发明公开了一种具有指令集随机化的代码  
虚拟化软件保护系统，属于计算机软件安全领域。  
所述发明包括在该系统中构建虚拟指令集，将待  
保护程序中的关键代码进行反汇编得到本地指令  
序列，接着根据以构架的虚拟指令集将本地指令  
序列转换为第一虚拟指令，并将第一虚拟指令进  
行编码得到字节码指令，将字节码指令与生成的  
附属虚拟机组件组合为附加文件，将附加文件与  
待保护程序组合为第一文件，在对第一文件进行  
垃圾数据填充后得到最终的保护程序。本发明能  
够提高待保护程序的复杂性，延长了分析者对待  
保护程序的分析时间，从而增强了待保护程序的  
保护能力。



1. 一种具有指令集随机化的代码虚拟化软件保护系统, 其特征在于, 所述具有指令集随机化的代码虚拟化软件保护系统, 包括 :

构建虚拟指令集;

对所述待保护程序进行 PE 文件检测;

在所述待保护程序中定位关键代码段;

对所述关键代码段进行反汇编, 得到本地指令序列;

根据设计好的虚拟指令集将所述本地指令序列转换为第一虚拟指令;

将所述第一虚拟指令按预设的编码规则进行编码, 得到字节码指令;

生成保护系统的附属虚拟机组件;

将所述字节码指令和所述附属虚拟机组件组合为附加文件, 将所述附加文件与所述待保护程序构成第一文件, 在所述第一文件中进行垃圾数据填充, 获得保护后的程序。

2. 根据权利要求 1 所述的具有指令集随机化的代码虚拟化软件保护系统, 其特征在于, 所述构建虚拟指令集, 包括 :

确定待构建的虚拟指令;

确定与所述虚拟指令对应的解释程序;

其中, 所述确定待构建的虚拟指令包括数据传输指令、算数和逻辑运算指令、控制转移指令。

3. 根据权利要求 1 所述的具有指令集随机化的代码虚拟化软件保护系统, 其特征在于, 所述在所述待保护程序中定位关键代码段, 包括 :

在所述待保护程序中的关键代码段的段首添加首标记;

在所述待保护程序中的关键代码段的段尾添加尾标记。

4. 根据权利要求 1 所述的具有指令集随机化的代码虚拟化软件保护系统, 其特征在于, 所述对所述关键代码段进行反汇编, 得到本地指令序列, 包括 :

获取所述关键代码段的起始地址和结束地址;

使用反汇编工具将所述起始地址与所述结束地址中的二进制代码进行反汇编, 得到本地指令序列;

其中, 所述本地指令序列中的指令按照地址顺序进行排列。

5. 根据权利要求 1 所述的具有指令集随机化的代码虚拟化软件保护系统, 其特征在于, 所述根据设计好的虚拟指令集将所述本地指令序列转换为第一虚拟指令, 包括 :

将所述本地指令序列的操作数压入栈中;

执行所述本地指令序列的目标操作, 获取执行结果, 并存放于所述栈中;

将所述执行结果存入虚拟寄存器环境或内存中。

6. 根据权利要求 1 所述的具有指令集随机化的代码虚拟化软件保护系统, 其特征在于, 所述将所述字节码指令和所述附属虚拟机组件组合为附加文件, 将所述附加文件与所述待保护程序构成第一文件, 在所述第一文件中进行垃圾数据填充, 获得保护后的程序, 包括 :

将所述字节码指令和所述附属虚拟机组件组合为附加文件, 将所述附加文件附在所述待保护程序后, 构成第一文件;

在所述第一文件中的所述关键代码段的起始位置处添加指向所述附加文件中虚拟机

初始化入口起始地址的跳转指令，在所述第一文件中的所述关键代码段的剩余内容中随机填充垃圾数据，填充完毕后，获得保护后的程序。

## 一种具有指令集随机化的代码虚拟化软件保护系统

### 技术领域

[0001] 本发明属于计算机软件安全领域,特别涉及一种具有指令集随机化的代码虚拟化软件保护系统。

### 背景技术

[0002] 软件的广泛应用在为社会带来巨大利益的同时,关于软件安全的问题也变得日益突出,软件防恶意逆向成为急需解决的问题。当前软件攻击技术和逆向工具的发展方便了攻击者对软件逆向分析。另外,软件一般运行在“白盒攻击”环境当中,攻击者可以完全控制软件的执行过程,查看指令执行的序列。因此,只要给定攻击者足够的时间,应该能够成功逆向分析该软件。

[0003] 人在一定时间内处理信息量是有限的,如果攻击者在攻击时付出的代价超过他所获得的信息的价值或者分析过程的难度和复杂度可以导致攻击者放弃攻击,则采用的保护方法是有效的。因此软件保护的目的在于提高攻击者逆向分析的难度,增加攻击者的攻击开销。

[0004] 为了阻碍恶意的逆向工程和保护软件中的智力成果,产业界和学术界提出了多种软件保护技术,主要包括:①注入垃圾指令来阻止反汇编,②使用程序加壳技术压缩和加密关键代码和数据,③应用代码混淆技术使攻击者更难从代码中提取有用的语言信息,等等。针对二进制代码的保护方法不受编程语言的限制,相比源代码级别应用范围更广。然而这些保护技术也存在着缺点,因为,注入的垃圾指令在程序运行时不会执行,而加壳技术中压缩和加密的代码在程序运行的时候也需要在解压和解密之后才能执行。所以,垃圾指令注入和加壳技术只能阻止攻击者的静态分析(在程序没有运行的状态下分析),但是不能阻止攻击者的动态分析(当程序运行在调试或者仿真环境下时分析)。当前常用的分析工具有OllyDbg和IDA Pro等。代码混淆的原理是让一个程序转化为功能上等价,但静态表示或执行过程不同的等价程序,主要是通过指令替换和顺序重排,改变程序的控制流,但是这些简单的指令压缩变换是容易被理解的。并且当前应用于二进制代码的混淆技术主要是通过花指令或垃圾指令进行保护,保护效果差,容易被发现和去除。

### 发明内容

[0005] 为了解决现有技术的问题,本发明提供了一种具有指令集随机化的代码虚拟化软件保护系统,所述具有指令集随机化的代码虚拟化软件保护系统,包括:

[0006] 构建虚拟指令集;

[0007] 对所述待保护程序进行PE文件检测;

[0008] 在所述待保护程序中定位关键代码段;

[0009] 对所述关键代码段进行反汇编,得到本地指令序列;

[0010] 根据设计好的虚拟指令集将所述本地指令序列转换为第一虚拟指令;

[0011] 将所述第一虚拟指令按预设的编码规则进行编码,得到字节码指令;

- [0012] 生成保护系统的附属虚拟机组件；
- [0013] 将所述字节码指令和所述附属虚拟机组件组合为附加文件，将所述附加文件与所述待保护程序构成第一文件，在所述第一文件中进行垃圾数据填充，获得保护后的程序。
- [0014] 可选的，所述构建虚拟指令集，包括：
  - [0015] 确定待构建的虚拟指令；
  - [0016] 确定与所述虚拟指令对应的解释程序；
  - [0017] 其中，所述确定待构建的虚拟指令包括数据传输指令、算数和逻辑运算指令、控制转移指令。
- [0018] 可选的，所述在所述待保护程序中定位关键代码段，包括：
  - [0019] 在所述待保护程序中的关键代码段的段首添加首标记；
  - [0020] 在所述待保护程序中的关键代码段的段尾添加尾标记。
- [0021] 可选的，所述对所述关键代码段进行反汇编，得到本地指令序列，包括：
  - [0022] 获取所述关键代码段的起始地址和结束地址；
  - [0023] 使用反汇编工具将所述起始地址与所述结束地址中的二进制代码进行反汇编，得到本地指令序列；
  - [0024] 其中，所述本地指令序列中的指令按照地址顺序进行排列。
- [0025] 可选的，所述根据设计好的虚拟指令集将所述本地指令序列转换为第一虚拟指令，包括：
  - [0026] 将所述本地指令序列的操作数压入栈中；
  - [0027] 执行所述本地指令序列的目标操作，获取执行结果，并存放于所述栈中；
  - [0028] 将所述执行结果存入虚拟寄存器环境或内存中。
- [0029] 可选的，所述将所述字节码指令和所述附属虚拟机组件组合为附加文件，将所述附加文件与所述待保护程序构成第一文件，在所述第一文件中进行垃圾数据填充，获得保护后的程序，包括：
  - [0030] 将所述字节码指令和所述附属虚拟机组件组合为附加文件，将所述附加文件附在所述待保护程序后，构成第一文件；
  - [0031] 在所述第一文件中的所述关键代码段的起始位置处添加指向所述附加文件中虚拟机初始化入口起始地址的跳转指令，在所述第一文件中的所述关键代码段的剩余内容中随机填充垃圾数据，填充完毕后，获得保护后的程序。
- [0032] 本发明提供的技术方案带来的有益效果是：
- [0033] 相对于现有技术，能够提高待保护程序的复杂性，延长了分析者对待保护程序的分析时间，从而增强了待保护程序的防逆向分析能力。

## 附图说明

- [0034] 为了更清楚地说明本发明的技术方案，下面将对实施例描述中所需要使用的附图作简单地介绍，显而易见地，下面描述中的附图仅仅是本发明的一些实施例，对于本领域普通技术人员来讲，在不付出创造性劳动的前提下，还可以根据这些附图获得其他的附图。

- [0035] 图 1 是本发明提供的一种具有指令集随机化的代码虚拟化软件保护系统的流程示意图；

[0036] 图 2 是本发明提供的一种具有指令集随机化的代码虚拟化软件保护系统的控制跳转指令中内部跳转和外部跳转的示例图；

[0037] 图 3 是本发明提供的一种具有指令集随机化的代码虚拟化软件保护系统的三种虚拟指令的两种不同编码结果的示例图；

[0038] 图 4 是本发明提供的一种具有指令集随机化的代码虚拟化软件保护系统中保护后程序的详细结构示意图。

## 具体实施方式

[0039] 为使本发明的结构和优点更加清楚，下面将结合附图对本发明的结构作进一步地描述。

[0040] 实施例一

[0041] 本发明提供了一种具有指令集随机化的代码虚拟化软件保护系统，所述具有指令集随机化的代码虚拟化软件保护系统，如图 1 所示，包括：

[0042] 01、构建虚拟指令集。

[0043] 02、对所述待保护程序进行 PE 文件检测。

[0044] 03、在所述待保护程序中定位关键代码段。

[0045] 04、对所述关键代码段进行反汇编，得到本地指令序列。

[0046] 05、根据设计好的虚拟指令集将所述本地指令序列转换为第一虚拟指令。

[0047] 06、将所述第一虚拟指令按预设的编码规则进行编码，得到字节码指令。

[0048] 07、生成保护系统的附属虚拟机组件。

[0049] 08、将所述字节码指令和所述附属虚拟机组件组合为附加文件，将所述附加文件与所述待保护程序构成第一文件，在所述第一文件中进行垃圾数据填充，获得保护后的程序。

[0050] 在实施中，为了实现防止程序被逆向分析，特此提出了通过将本地 x86 指令转换为虚拟指令，在运行时利用嵌入的虚拟解释器对虚拟指令进行解释执行以达到和原始本地指令相同的功能。在该保护方法下，将原始指令进行复杂化处理，加大分析者理解的难度，从而起到对程序进行保护的效果。

[0051] 具体的，本发明中提供的方法的主要思想为：

[0052] 首先构建虚拟指令集和 handler，这里的虚拟指令集的设计，需要能够替代任何的本地指令，要保证语义的完备性。Handler 是指虚拟指令的解释程序，本地指令替换成虚拟指令，虚拟指令最终需要用 handler 来解释执行，handler 由本地指令写成。

[0053] 接着，对待保护程序进行 PE 检测，这里的 PE 为可执行文件 Portable Executable。所有 Windows 下的 32 位或 64 位可执行文件都是 PE 文件格式，其中包括 DLL、EXE、FON、OCX、LIB 和部分 SYS 文件。在 Windows 系统下的可执行文件的一种（还有 NE、LE），是微软设计、TIS（Tool Interface Standard，工具接口标准）委员会批准的一种可执行文件格式。通过对待保护程序是否为可执行文件的检测，只有在确定待保护程序为可执行类型后，执行后续处理步骤。

[0054] 后续的，在待保护程序中确定关键代码段的位置，将关键代码段进行反汇编得到本地指令序列，进而结合之前构建好的虚拟指令集将本地指令序列进行转换得到第一虚拟

指令,进而得到对应的字节码指令和附属虚拟机组件,最后将字节码指令和附属虚拟机组件组合为附加文件,将附加文件与待保护程序构成第一文件,对第一文件进行处理,最终得到保护后的程序。

[0055] 本申请提供一种具有指令集随机化的代码虚拟化软件保护系统,在该系统中构建虚拟指令集,将待保护程序中的关键代码进行反汇编得到本地指令序列,接着根据已构建的虚拟指令集将本地指令序列转换为第一虚拟指令,并将第一虚拟指令进行编码得到字节码指令,将字节码指令与生成的附属虚拟机组件组合为附加文件,将附加文件与待保护程序组合为第一文件,在对第一文件进行垃圾数据填充后得到最终的保护后的程序。相对于现有技术,能够提高待保护程序的复杂性,延长了分析者逆向分析程序的时间,从而增强了待保护程序的防逆向分析能力。

[0056] 可选的,所述构建虚拟指令集,即步骤 01 包括:

[0057] 确定待构建的虚拟指令;

[0058] 确定与所述虚拟指令对应的解释程序;

[0059] 其中,所述确定待构建的虚拟指令包括数据传输指令、算数和逻辑运算指令、控制转移指令。

[0060] 在实施中,这里的虚拟指令集被设计为需要能够替代任何的本地指令,同时还要保证语义的完备性。

[0061] 虚拟指令的设计,以及从本地 x86 汇编指令到虚拟指令的翻译过程和所选用的虚拟机的架构是密切相关的(虚拟机的架构主要有两种,一种是基于栈的,一种是基于寄存器的)。本发明选用的虚拟机架构是基于栈的,一旦虚拟机架构确定下来,虚拟指令的设计原则和本地 x86 指令到虚拟指令的翻译过程也就确定了。

[0062] 在基于栈的实现中,对指令的解释一般经历三个阶段:

[0063] 首先,本地 x86 指令的操作数被压入到栈中。

[0064] 然后,针对栈顶的操作数进行预期的操作(比如求和)并把结果放到栈顶。

[0065] 最后,把栈顶的数据进行保存(保存到虚拟寄存器环境或内存中)。

[0066] 上述过程实际上描述了 x86 指令到虚拟指令的翻译过程,同时也指明了虚拟指令集中应该包括的虚拟指令(load 用于将参数压栈,store 用于弹栈并保存运算结果,另外还有其他的指令用于执行常见的操作,如加、减、逻辑与运算等)。所以,在本发明中虚拟指令设计这部分是和虚拟机架构相关的,一旦虚拟指令确定,如果再结合虚拟寄存器环境的内容,那么与虚拟指令对应的解释程序即 handler 的实现就可以确定,因为 handler 的目的就是实现虚拟指令所定义的对虚拟寄存器环境 VMcontext 或内存的操作。

[0067] 为了实现本地指令向虚拟指令集的映射,这里将需要解释的本地指令分为三类,分别为数据传输指令、算数和逻辑运算指令、控制转移指令。

[0068] (1) 数据传输指令:典型的为“load”和“store”指令。“load”指令是将操作数压入栈顶,“store”指令从栈顶取出结果存入虚拟环境当中。主要用在虚拟化本地指令的第一和第三步操作中,“load”指令的操作对象是虚拟寄存器、内存地址或者立即数;“store”指令操作对象是虚拟寄存器或者是内存地址。

[0069] 其中,虚拟寄存器是存储在虚拟环境中的,是本地环境中寄存器在虚拟环境中的映射。除了操作数的类型,还需要考虑操作数的大小,8 位、16 位或者 32 位。在 x86 指令结

构中,对栈的“pop”和“push”操作不支持8位的操作数,本系统的虚拟指令集将操作数的大小考虑过程放到虚拟化本地指令的第二步即执行目标操作中来实现。

[0070] 表1 给出了“load”和“store”的虚拟指令和相应的 handler :

[0071]

虚拟指令	处理程序
load_r reg	...; 将虚拟寄存器的索引值放入寄存器 eax 中 push dword[VMcontext+eax*4]
load_r8h reg	...; 将虚拟寄存器的索引值放入寄存器 eax 中 mov ah, byte[VMcontext+eax*4+1] movzx eax, ah push eax
load_m mem	...; 将内存地址放入寄存器 eax 中 push dword[eax]
load_ms	pop eax push dword[eax]
load_i imm	...; 将立即数的值放入寄存器 eax 中 push eax
store_r reg	...; 将虚拟寄存器的索引值放入寄存器 eax 中

[0072]

	pop dword[VMcontext+eax*4]
store_r8h reg	...; 将虚拟寄存器的索引值放入寄存器 eax 中 pop edx mov byte[VMcontext+eax*4], dl
store_m mem	...; 将内存地址放入寄存器 eax 中 pop dword[eax]
store_ms	pop eax pop ebx mov dword[eax], ebx

[0073] 表1 :“load”和“store”的虚拟指令和相应的 handler

[0074] (2) 算数和逻辑运算指令。这类虚拟指令相对于本地指令变化不大,区别在于寻址方式是固定的,即基于堆栈的;算数和逻辑运算虚拟指令不需要考虑操作数的问题,它的操

作数已经通过“load”指令压栈；同时，在上一部分中说到将操作数的大小区分通过这类指令来实现，在32位系统中，操作数的大小可以是8位、16位或者32位，本系统针对每一种情况设计了对应的虚拟指令。表2给出了“add”操作的虚拟指令和对应handler：

[0075]

虚拟指令	处理程序
add8	pop eax add byte[esp], al
add16	pop eax add word[esp], ax
add32	pop eax add dword[esp], eax

[0076] 表2：“add”操作的虚拟指令和handler

[0077] (3) 控制跳转指令。该类指令用于改变字节码程序的控制流。在本地指令中，常用控制跳转指令包括“jmp”，“jcc”（条件跳转），“call”和“retn”。这类指令有不同的形式，每种形式都需要有对应的虚拟指令。

[0078] 根据控制跳转指令的目的地址判断，如果目的地址仍然在关键代码段内部，称之为内部跳转；否则称为外部跳转。图2给出这两种跳转的示例。

[0079] 根据目的指令的地址定位，需要考虑控制跳转指令的目的地址是否是静态可计算的。基于这一点，控制跳转指令可以分成两类：直接跳转和间接跳转。直接跳转，指通过偏移地址来计算跳转的目的地址，在静态时（即没有运行程序的情况下）能够计算出目的地址；间接跳转，指目的地址存储在寄存器或者内存当中，目的地址不是静态定义而是在程序运行过程中确定。表3给出了关于上述两种情况的不同指令的形式，表中“rel”指偏移地址。

[0080]

直接控制跳转	jmp rel8/16/32 jcc rel8/16/32 call rel16/32
间接控制跳转	jmp reg32/mem32 call reg32/mem32 retn

[0081] 表3：直接跳转和间接跳转指令示例

[0082] “jmp”指令是所有控制跳转指令的基础，表4给出了关于“jmp”指令的虚拟指令和其对应的处理程序的详细介绍来做示例。

[0083] 其中：直接“jmp”指令的目的地址是可以静态计算的。如果目的指令是在关键代

码段内部，则为直接内部“jmp”，否则为直接外部“jmp”。本系统的指令集针对以上两种情况设计相应的虚拟指令，“jmp\_di”处理第一种情况，“jmp\_do”处理第二种情况。对于“jmp\_di”指令，目的字节码指令地址可在保护代码段内获得，用“jmp\_di”的操作数来设置字节码指令地址，利用“load\_i”指令将操作数压入栈中，然后“jmp\_di”的处理程序从栈中获取地址并分配给VPC（虚拟程序计数器）。对于“jmp\_do”指令，可直接跳转到本地程序的目的地址，但需要在跳转操作之前恢复本地寄存器环境；“jmp\_do”的操作数就是本地目的指令的地址。

[0084] 间接“jmp”指令同样需要做内部和外部“jmp”区分，对于在静态的程序保护过程中无法获得目的指令的地址，因此不能直接判断是内部跳转还是外部跳转，本系统的解决方案是在程序运行时进行判断。因此，该类指令只需要定义一个虚拟指令，即“jmp\_in”，它的操作数是本地目的指令的地址。在程序运行过程中，处理程序从栈中获取目的地址并与关键代码段的起始地址比较。如果目的地址在关键代码段中，为内部“jmp”，否则为外部“jmp”。对于第一种情况，首先查找记录本地指令和字节码指令对应关系的映射表，找到本地目的指令对应字节码指令的地址分配给VPC。对于后一种情况，首先恢复本地寄存器环境，然后直接跳转到本地指令目的地址执行。

[0085]

虚拟指令	处理程序
jmp_di	... ; 操作数：目标字节码指令的地址 ... ; 将获取的操作数放入寄存器 eax 中 mov VPC, eax
jmp_do	... ; 操作数：目标本地指令的地址 ... ; 获取操作数并且存入“mem”中 ... ; 恢复本地环境 jmp dword [mem]
jmp_in	... ; 目标地址从栈顶获取 Label_inner: ... ; 查找映射表找到目标字节码指令地址 ... ; 获取目标字节码指令地址并赋给 VPC Label_outer: ... ; 恢复本地寄存器环境，跳转到目标指令处

[0086] 表 4：“jmp”指令的虚拟地址和 handler

[0087] 至于其他控制跳转指令的虚拟化过程与“jmp”指令是相似，只有一些很小的差别。条件跳转指令（jcc）只支持直接跳转，并且在处理程序中需要几条额外的指令来判断跳转条件是否成立，进而决定是否进行跳转。“call”指令可以看作是一个“push”指令和一个“jmp”指令组合实现，“push”指令将返回地址压入栈中，“jmp”指令跳向函数的起始地址。

“retn”指令是间接跳转指令，目的地址通过栈来获取。

[0088] 通过上述对虚拟指令以及对应的本地指令类型的描述，从而确定了虚拟指令与本地指令的映射关系，便于后续步骤中对本地指令向虚拟指令的转换。

[0089] 可选的，所述在所述待保护程序中定位关键代码段，即步骤 03 包括：

[0090] 在所述待保护程序中的关键代码段的段首添加首标记；

[0091] 在所述待保护程序中的关键代码段的段尾添加尾标记。

[0092] 在实施中，关键代码段是指目标文件中需要被保护的代码，通常是待保护文件中核心算法的实现代码或对重要数据进行操作的代码。确定关键代码段后需要在关键代码段的段首和段尾分别添加首标记和尾标记；

[0093] 具体的，采用的首尾标记如下：

[0094] #define NISL\_START\_emit\_(0xEB, 0x0C, 0x4E, 0x49, 0x53, 0x4C, 0x56, 0x4D, 0x53, 0x54, 0x41, 0x52, 0x54, 0x00)

[0095] #define NISL\_END\_emit\_(0xEB, 0x0C, 0x4E, 0x49, 0x53, 0x4C, 0x56, 0x4D, 0x45, 0x4E, 0x44, 0x00, 0x00, 0x00)

[0096] 实际操作时，将 NISL\_START 和 NISL\_END 两个宏定义复制到待保护文件的源文件中，并将两个宏添加到关键代码段的首尾处，源文件编译后即可实现首尾标记的嵌入。

[0097] 这里通过在关键代码段的首、尾分别添加标记的方法，使得在后续步骤中能够方便地确定该关键代码段所在的位置。

[0098] 可选的，所述对所述关键代码段进行反汇编，得到本地指令序列，即步骤 04 包括：

[0099] 获取所述关键代码段的起始地址和结束地址；

[0100] 使用反汇编工具将所述起始地址与所述结束地址中的二进制代码进行反汇编，得到本地指令序列；

[0101] 其中，所述本地指令序列中的指令按照地址顺序进行排列。

[0102] 在实施中，在待保护程序中获取关键代码段的起始地址和结束地址，利用反汇编工具（如 xde 等）将得到的关键代码段的二进制代码反汇编得到汇编指令，按照汇编指令按照地址顺序组成本地指令序列。

[0103] 可选的，所述根据设计好的虚拟指令集将所述本地指令序列转换为第一虚拟指令，即步骤 05 包括：

[0104] 将所述本地指令序列的操作数压入栈中；

[0105] 执行所述本地指令序列的目标操作，获取执行结果，并存放于所述栈中；

[0106] 将所述执行结果存入虚拟寄存器环境或内存中。

[0107] 在实施中，将本地指令序列转换为虚拟指令，主要分为如下步骤：

[0108] (1) 通过“load”虚拟指令将本地指令的操作数压入栈中。

[0109] (2) 执行目标操作的指令。执行本地指令的目标操作，该虚拟指令不需要考虑操作数的类型，直接从栈顶获取相关操作数，但需要考虑操作数的大小。

[0110] (3) 通过“store”虚拟指令将操作执行的结果存入虚拟寄存器环境或内存中。

[0111] 数据传输指令虚拟化过程主要使用“load”、“store”指令，如“mov”、“push”和“pop”指令；算数和逻辑运算指令的虚拟化过程严格按照上面的三步操作来实现；控制跳转指令虚拟化过程通过“load”指令和“jmp”指令组合实现。表 5 给出了一些本地指令虚

拟化的示例。

[0112] 有些本地指令有复杂的寻址方式,在虚拟化的过程中会反复用到上述虚拟指令,例如表 5 中的“move eax, dword[esi+4]”指令;其中,表 5 中的“42a583h”是在地址“4020a8h”中存储的本地指令相对应的字节码指令的地址。

[0113]

本地指令	虚拟指令
mov eax, ebx	load_r 4 store_r 0
Mov eax, dword[esi+4]	load_r 4 load_i 4 add32 load_ms store_r 0
add eax, edx	load_r 0 load_r 3 add32 store_r 0
Jmp 4020a8h (直接内部跳转)	load_i 42a583h jmp_di

[0114] 表 5 :本地指令和对应虚拟指令的示例

[0115] 通过将本地指令按上述步骤转换为虚拟指令,从而在一定程度上提高了对待保护程序的保护效果。

[0116] 可选的,所述将所述字节码指令和所述附属虚拟机组件组合为附加文件,将所述附加文件与所述待保护程序构成第一文件,在所述第一文件中进行垃圾数据填充,获得保护后的程序,包括:

[0117] 将所述字节码指令和所述附属虚拟机组件组合为附加文件,将所述附加文件附在所述待保护程序后,构成第一文件;

[0118] 在所述第一文件中的所述关键代码段的起始位置处添加指向所述附加文件中虚拟机初始化入口起始地址的跳转指令,在所述第一文件中的所述关键代码段的剩余内容中随机填充垃圾数据,填充完毕后,获得保护后的程序。

[0119] 在实施中,步骤 06 中提出对第一虚拟指令按预设的编码规则进行编码后,得到对应的字节码指令。

[0120] 关于字节码指令,具体为:虚拟指令和字节码指令是一种简单的对应关系,本系统采取一种简单的编码规则,即将虚拟指令中的操作码和操作数分别进行编码。在实现中,给每一个虚拟指令指定不同的 ID,这些 ID 取值范围是 0 ~ 255,用以表明虚拟指令的操作码,

使用一个字节即可编码所有的操作码,。由于操作数的种类和大小不一样,所以需要使用一个或多个字节进行编码:虚拟寄存器的索引是 8 位,用一个字节编码;立即数的值可以是 8/16/32 位,分别用一个、两个、四个字节进行编码;内存地址是 32 位,使用四个字节进行编码。图 3 中给出了一些虚拟指令和它们对应的字节码的示例。

[0121] 对于每一个保护实例,本发明的具有指令集随机化的代码虚拟化保护系统都会生成一个独特的编码规则对虚拟指令进行编码。同一个字节码指令在不同的保护实例中很可能是不同的,用一个字节来定义 ID,所以两个字节码指令完全相同的概率是:

$$[0122] p = \frac{1}{256},$$

[0123] 假设在保护系统中有 N 条虚拟指令,生成的不同的编码规则的总数是:

$$[0124] p(256, N) = \frac{256!}{(256 - N)!},$$

[0125] 这是一个较大的数字。根据这样的编码规则可以实现虚拟指令随机化,编码时先随机化改变操作码和虚拟指令之间的对应关系,利用随机化后的对应关系来编码字节码指令。图 3 给出了三种虚拟指令的两种不同的编码结果。软件具有多样性,是阻止大规模开发和破解的有效策略,本发明的指令集随机化方法通过产生这种多样性的本质,进一步的增强了攻击的难度并且阻碍了软件攻击方法的自动化,能够有效的达到保护软件的目的。

[0126] 步骤 07 中提出生成保护系统的附属虚拟机组件。关于附属虚拟机组件,具体为:包括:VMcontext、VMinit、VMloop、Handlers、VMexit:

[0127] 生成的虚拟机各个组件功能说明如下:

[0128] (1) VMcontext 是代码虚拟化保护系统的虚拟寄存器环境,对应了 7 个真实寄存器(即除了 ESP 之外的 7 个通用寄存器)一个标志寄存器。

[0129] (2) VMinit 是虚拟机的入口,当保护后的程序运行到关键代码段时就会有个无条件跳转操作,跳向虚拟机的入口,开始虚拟寄存器环境的初始化,是虚拟机第一个运行的组件。VMinit 开始工作后会将本地环境中通用寄存器的内容全部存入虚拟寄存器环境中对应的虚拟寄存器位置,然后转到 VMloop 继续执行。

[0130] (3) VMloop 是虚拟机运行的核心部分,程序运行时,VMloop 会逐条读取字节码指令,并根据操作码找到对应的 handler 去执行以解释字节码指令,直到解释执行完所有的字节码指令,最后将转到 VMexit 继续执行。

[0131] (4) Handlers 是所有虚拟指令的解释程序的集合,集合中某一 handler 的执行完成对应虚拟指令所定义的操作。通过各个 handler 的先后执行,最终能够实现和原始关键代码段相同的功能。

[0132] (5) VMexit 是虚拟机的出口,当 VMloop 解释执行完所有的字节码指令后,VMexit 开始工作,它的功能是将虚拟寄存器环境恢复到本地环境,即将所有虚拟寄存器的内容恢复到对应的本地环境寄存器中。然后跳向受保护的关键代码段的下一条指令,继续执行关键代码段的后续指令。

[0133] 当根据上述步骤获取字节码指令和附属虚拟机组件后,将二者组合为附加文件,

并且将附加文件挂靠在待保护程序后方与待保护程序构成第一文件。这里的第一文件的结构如图 4 所示。

[0134] 值得注意的是,此时的第一文件相对于待保护程序因为添加了附加文件导致原有的属性发生变化,因此此时需要在与待保护程序对应的属性中对文件大小以及包括的区段数目进行数据更新。

[0135] 进一步的,需要将位于第一文件中的关键代码段的起始位置处添加一条无条件跳转语句,该跳转语句用于指向附加文件中所包含的虚拟机初始化入口即 VMinit 处代码段的起始地址处。除此之外,在关键代码的其余位置均使用垃圾数据进行随机填充。在具体执行时,这里填充的垃圾数据对应的代码不会被执行,不会对程序的功能进行影响,还可以对分析者起到迷惑作用,进一步提高了最终生成的保护后程序的安全性。

[0136] 本申请提供一种具有指令集随机化的代码虚拟化软件保护系统,在该系统中构建虚拟指令集,将待保护程序中的关键代码进行反汇编得到本地指令序列,接着根据已构建的虚拟指令集将本地指令序列转换为第一虚拟指令,并将第一虚拟指令进行编码得到字节码指令,将字节码指令与生成的附属虚拟机组件组合为附加文件,将附加文件与待保护程序组合为第一文件,在对第一文件进行垃圾数据填充后得到最终的保护后的程序。相对于现有技术,能够提高待保护程序的复杂性,延长了分析者逆向分析程序的时间,从而增强了待保护程序的保护能力。

[0137] 实验部分 :

[0138] 为了检验系统的性能和时空开销,进行了如下实验 :

[0139] 实验环境为 Win 7 操作系统,3.0GHz 处理器,4GB 内存。选用的测试程序有四个,分别是 md5. exe (md5 消息摘要计算), gzip. exe (gzip 压缩), bcrypt. exe (blowfish 加密) 和 mat \_mul. exe (矩阵乘法)。其中,前三个测试程序用于处理一个 5KB 大小的文本文件 (test. txt), mat \_mul. exe 用于计算两个 5 阶矩阵的乘积。表 6 给出了四个测试程序的基本信息,对于每一个测试程序,都选取了一段关键的代码进行保护,表 6 中第二列给出了这些关键代码,第三列 ( $I_p$ ) 是这些关键代码对应的 x86 指令的条数,最后一列 ( $I_e$ ) 是这些程序在处理输入数据时,位于关键代码中的指令执行的条数,该数据是通过 Pin 动态跟踪得到的。

[0140]

测试程序	保护的代码	$I_p$	$I_e$
md5. exe	Transform()	563	41662
gzip. exe	deflate()	153	267771
bcrypt. exe	Blowfish_Encrypt()	54	1570756
mat _mul. exe	ijkalgorithm()	60	84325

[0141] 表 6 测试程序的基本信息

[0142] 利用所述的代码虚拟化保护系统对测试程序进行保护,得到各程序保护后的版本。分别记录原始程序和保护后程序的文件大小,并记录下原始程序和保护后程序运行的平均时间(前三个程序处理 5KB 的文本文件, mat \_mul. exe 计算两个 5 阶矩阵的乘积),结

果显示在表 7 中。

[0143] 所述保护系统对文件大小的影响体现在添加的虚拟机新节。在虚拟机新节中，除了字节码程序的大小是不固定的外，其余部分的大小都是固定的，和测试程序无关。由于 Windows 中 PE 文件中各个节都是按照一定的对齐值 (0.5KB 或 4KB) 进行对齐的，所以后面三个程序文件大小的增幅都为 8KB，而 md5.exe 中由于需要保护的指令条数比较多，生成的字节码程序也比较大，所以文件大小的增幅最大。

[0144] 从表 7 中可以看出，所述保护系统对测试程序的性能影响较小。其中所述保护系统对 bcrypt.exe 的性能影响最大，这是因为 bcrypt.exe 中被保护的指令执行的条数最多（表 7 最后一列），因此，保护后生成的字节码程序的执行次数也最多，时间消耗也最大。

[0145]

	原始程序		保护后程序	
	size/KB	time/ms	size/KB	time/ms
md5.exe	11	1.858	24	4.869
gzip.exe	56	4.518	64	13.305
bcrypt.exe	64	3.605	72	39.754
mat_mul.exe	184	12.964	192	13.736

[0146] 表 7 所述代码虚拟化保护系统对文件大小和程序性能的影响

[0147] 更进一步地，我们计算了所述保护系统对每条 x86 指令的平均性能消耗，计算方法为：

[0148]

$$\frac{\text{保护后程序的执行时间} - \text{原始程序的执行时间}}{\text{受保护的指令执行的条数}}$$

[0149] 表 8 给出了所述保护系统对各测试程序每条 x86 指令的平均性能消耗。从表 8 中可以看出，md5.exe 中每条 x86 指令的平均性能消耗最大，这是因为 md5.exe 程序中受保护的指令绝大部分都是算术和逻辑运算指令，而其他程序中则主要以数据传送指令为主。算术和逻辑运算指令比数据传送指令需要更多的 handler 来解释，因此性能消耗更大。

[0150]

md5.exe	gzip.exe	bcrypt.exe	mat_mul.exe

[0151]

7.23	3.28	2.30	0.92

[0152] 表 8 每条 x86 指令的平均性能消耗 ( $10^5$ ms/ 条)

[0153] 需要说明的是：上述实施例提供的一种具有指令集随机化的代码虚拟化软件保护系统进行代码保护的实施例，仅作为该代码虚拟化软件保护系统在实际应用中的说明，还可以根据实际需要而将上述代码虚拟化软件保护系统在其他应用场景中使用，其具体实现过程类似于上述实施例，这里不再赘述。

[0154] 上述实施例中的各个序号仅仅为了描述,不代表各部件的组装或使用过程中的先后顺序。

[0155] 以上所述仅为本发明的实施例,并不用以限制本发明,凡在本发明的精神和原则之内,所作的任何修改、等同替换、改进等,均应包含在本发明的保护范围之内。

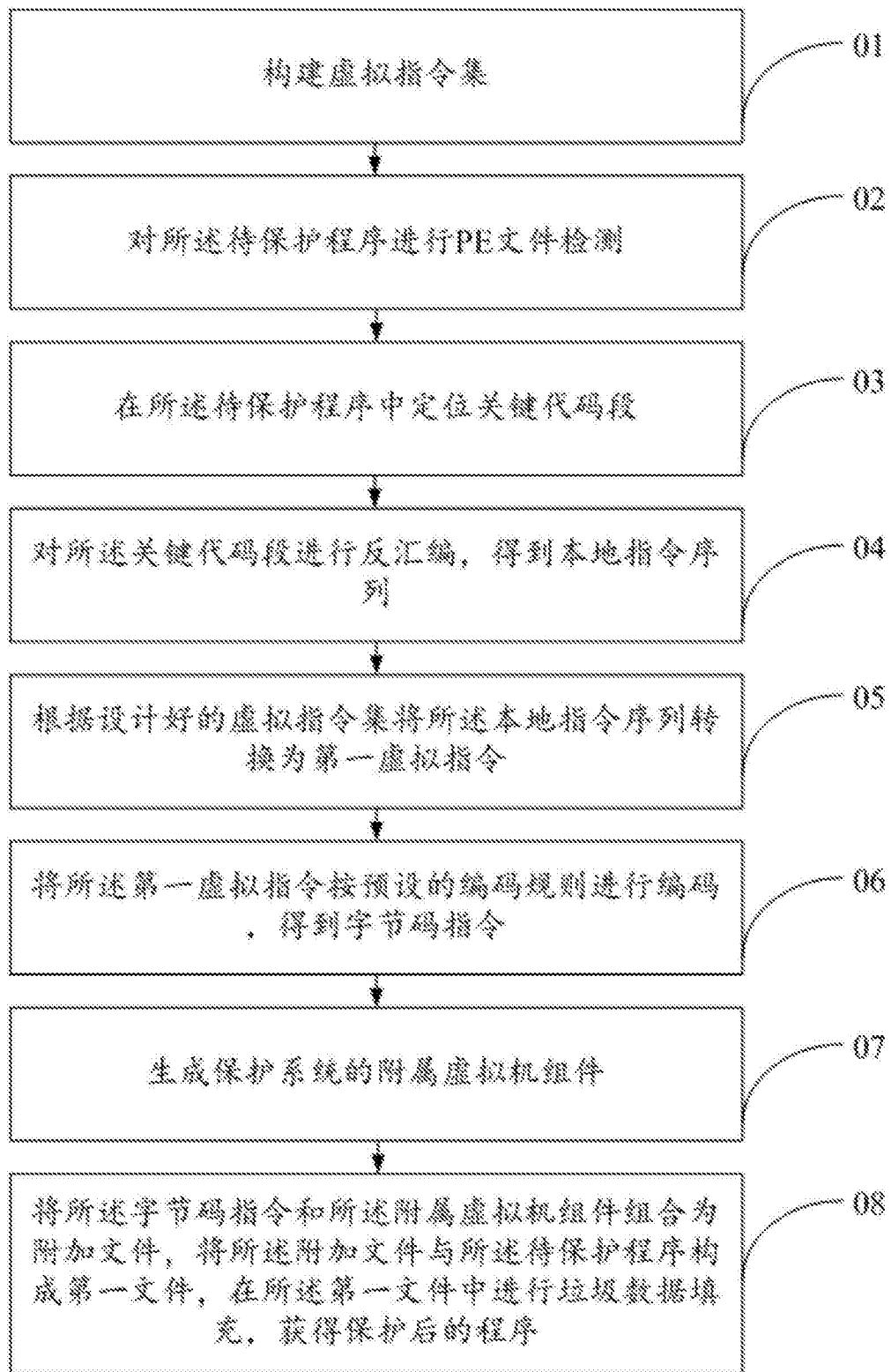


图 1

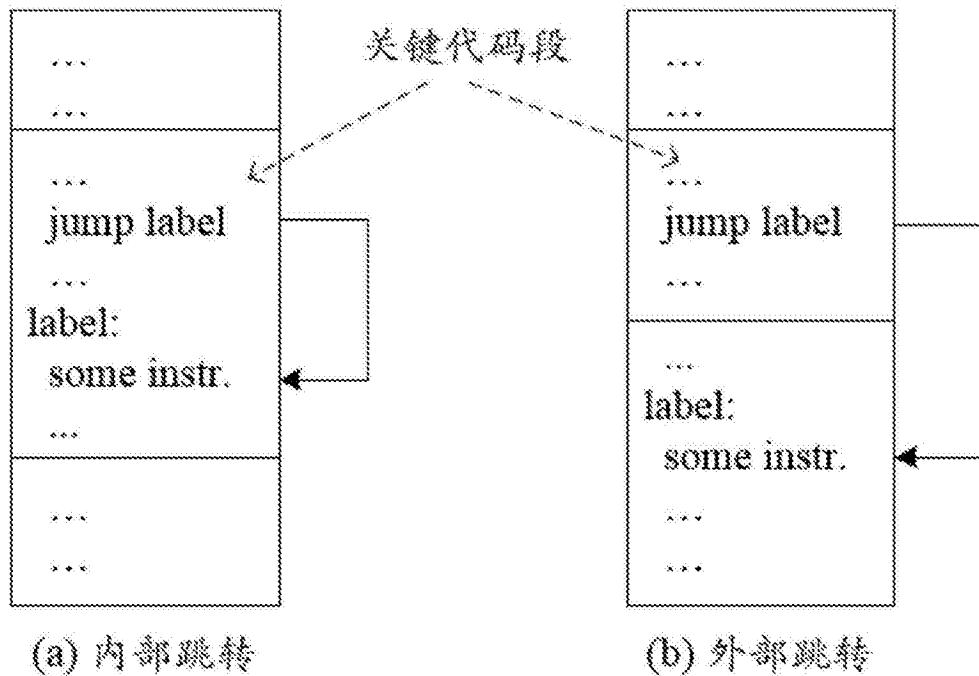


图 2

虚拟指令1: load_r 4	
字节码编码1:	00 04
字节码编码2:	c6 04
	↓ ↗
虚拟指令2: load_i 4020a8h	
字节码编码1:	04 a8 20 40 00 (小端存储)
字节码编码2:	42 a8 20 40 00
	↓ ↗
虚拟指令3: add32	
字节码编码1:	04
字节码编码2:	00

图 3

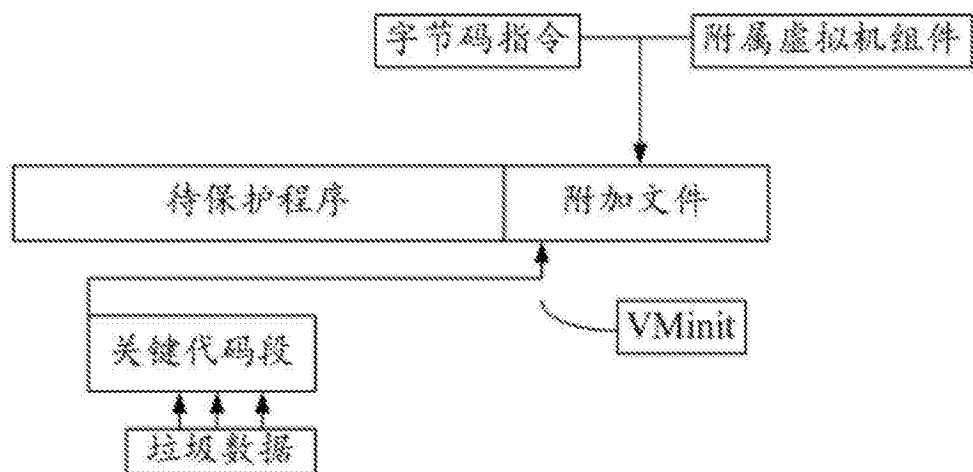


图 4