

基于代码碎片化的软件保护技术

郭京城, 舒辉, 熊小兵, 康绯
(信息工程大学, 河南 郑州 450001)

摘要: 针对当前软件保护技术存在的不足, 提出一种代码碎片化技术, 该技术是一种以函数为单元, 对函数进行代码 shell 化、内存布局随机化、执行动态链接化的新型软件保护技术, 代码 shell 化实现代码碎片的位置无关变形, 内存布局随机化实现代码碎片的随机内存加载, 动态链接化实现对代码碎片的动态执行, 通过上述 3 个环节实现对程序的碎片化处理。实验表明, 代码碎片化技术不仅能实现程序执行过程中函数碎片内存位置的随机化, 还能实现函数碎片的动态链接执行, 增加程序静态逆向分析和动态逆向调试的难度, 提高程序的抗逆向分析能力。

关键词: 代码碎片化; 软件保护; 分离; 动态链接

中图分类号: TP309.5

文献标识码: A

doi: 10.11959/j.issn.2096-109x.2020063

Software protection technology based on code fragmentation

GUO Jingcheng, SHU Hui, XIONG Xiaobing, KANG Fei
Information Engineering University, Zhengzhou 450001, China

Abstract: Aiming at the shortcomings of the current software protection technology, a code fragmentation technology was proposed. This technology is a new software protection technology that takes functions as units, shells functions, randomizes memory layout, and performs dynamic linking. The code shellization realizes the position-independent morphing of code fragments, the memory layout randomizes the random memory loading of the code fragments, the dynamic linking realizes the dynamic execution of the code fragments, and the program fragmentation processing is achieved through the above three links. The experiments show that the code fragmentation technology can not only realize the randomization of the memory location of function fragments during program execution, but also the dynamic link execution of function fragments, increasing the difficulty of static reverse analysis and dynamic reverse debugging of the program, and improving the anti-reverse analysis ability of the program.

Key words: code fragmentation, software protection, separation, dynamic linking

收稿日期: 2019-12-05; 修回日期: 2020-02-21

通信作者: 舒辉, shuhui@126.com

基金项目: 国家重点研发计划 (2016YFB08011601)

Foundation Item: The National Key R&D Program of China (2016YFB08011601)

论文引用格式: 郭京城, 舒辉, 熊小兵, 等. 基于代码碎片化的软件保护技术[J]. 网络与信息安全学报, 2020, 6(6): 57-68.
GUO J C, SHU H, XIONG X B, et al. Software protection technology based on code fragmentation[J]. Chinese Journal of Network and Information Security, 2020, 6(6): 57-68.

1 引言

软件保护问题一直是计算机安全领域的重要研究课题,虽然当前的软件保护手段能够在一定程度上增加软件被恶意修改和逆向分析的难度,但攻击方可以实现对程序的结构特征进行静态分析^[1-2],也可以反复地对代码进行逐步运行、调试、反汇编、反编译,能够获取软件在内存中的数据、执行期间的中间值,通过连续获取系统的内存镜像,进行前后镜像之间的对比分析,可以获取程序的具体行为^[3]的动态分析,因此,如何有效地实现对应用程序的防破解、防盗版是一个亟待解决的问题。当前有很多软件保护技术,代码加壳^[4]是通过对原程序的压缩加密实现对可执行文件的保护手段,代码虚拟化^[5]是将本地指令转换成自定义的虚拟指令,并通过虚拟解释器解释执行虚拟指令,实现原始程序的功能,代码混淆^[6-9]分为外形混淆、控制混淆、数据混淆和预防混淆^[10-12]。在源代码基础上实现若干种控制混淆转换与数据混淆转换,通过适当插入垃圾代码和不透明谓词实现的改进型控制流混淆算法^[13],通过增加冗余基本块的数量和随机下标生成实现改进型的二进制程序控制流扁平化^[14],是常见的代码混淆手段。内存布局随机化通过不断改变进程内存布局,实现进程攻击面的随机性变换,从而消除了攻击者的不对称优势,增加了攻击者的攻击难度^[15-16]。这些软件保护技术能够实现对代码的保护,很大程度上增加逆向攻击者对程序静态分析的难度,但不能实现抗动态调试,当逆向攻击者通过动态调试工具对保护后的代码进行动态调试时,代码的执行逻辑依然会被暴露,进而可以通过内存 dump 方法获取完整的程序代码。针对当前软件保护技术在动态分析中存在的不足,提出一种基于代码碎片化的新型软件保护技术,该技术通过在中间语言(中间语言是一种采用静态单赋值形式的中间表达)层面对功能函数代码碎片进行分离,实现对代码碎片的动态链接执行。该技术将功能函数的直接调用转换成间接调用,能够避免被完整地控制流分析,进而增加静态分析的难度;各个函数模块能够加载到内存中的任意位置,有效防止程序被 dump 分析,增加程序动态分析的难度。

2 基本思路

代码碎片化技术基于 LLVM 编译平台,LLVM 是一系列分模块、可重用的编译工具链,它提供了一种代码编写良好的中间表示(IR),可以作为多种语言的后端,还可以提供与编程语言无关的优化和针对多种 CPU 的代码生成功能。LLVM 中间表示是一种采用静态单赋值形式(SSA, static single assignment)的中间表达(intermediate representation),包含一套汇编语言类似的指令集和一个类型系统,该指令集是类似于 RISC(reduced instruction set computer)的三地址指令集。以可移植的执行体(PE, portable executable)程序碎片化处理过程为例,说明代码碎片化技术的原理,如图 1 所示。首先,对源代码进行 shell 化变形,在中间语言层面对源代码进行相应的修改,将代码中的所有全局变量和应用程序编程接口(API, application programming interface)函数等全局数据提取出来保存在全局数据区。然后,在中间语言层面将所有功能函数以数组的形式保存起来,并将功能函数的调用由直接调用改为间接调用的形式,添加 Loader 函数,负责对功能函数进行调用,同时构建功能函数的动态地址表,依据构建的函数数组将所有函数代码碎片分离到独立的中间语言文件中。最后,将主函数、Loader 函数、全局数据区数据和分离的功能函数碎片作为一个整体进行编译,生成类 PE 文件。

3 碎片化技术模型建立

代码碎片化技术有 3 个阶段:函数代码 shell 化、函数内存布局随机化、函数执行动态链接化。本文通过符号化形式建立碎片化的技术模型。

定义 1 函数 $f_i = \{G_{di}, LC_{di}, T_i, I_i, Q_{I_i \rightarrow G_{di}}, Q_{I_i \rightarrow L_{di}}, Q_{I_i \rightarrow T_i}, P_i, L_{P_i \rightarrow f_i}, C_i, L_{f_i \rightarrow C_i}, M_{f_i}\}$, $0 < i \leq n$, 其中, n 是程序中函数的个数。

定义 2 f_i 中的元素定义如表 1 所示。

定义 3 函数代码 shell 化 $S = \{S_1, S_2, S_3, S_4, S_5\}$, 其中, S_1 表示全局变量数据聚合, S_2 表示导入函数信息聚合, S_3 表示实现全局变量和 API 函数调用指令的修改, S_4 表示全局变量调用方式的修

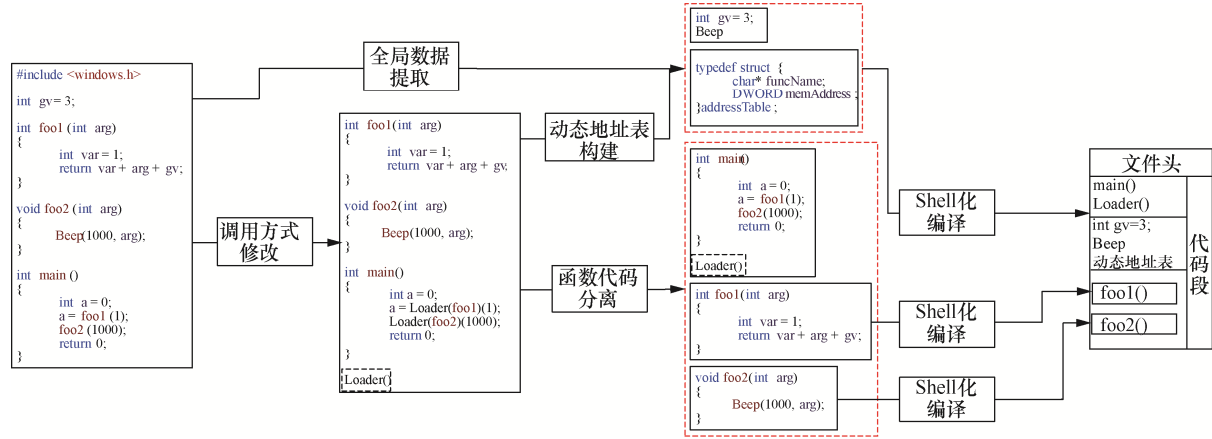


图 1 碎片化处理过程

Figure 1 The process of fragmentation

改, S_5 表示 API 函数调用方式的修改。

表 1 函数 f_i 中各元素定义
Table 1 Definition of element in f_i

标识符	定义
G_{di}	f_i 中所使用的全局变量的集合
LC_{di}	f_i 中所使用的局部变量的集合
T_i	f_i 的导入表
I_i	f_i 中所有指令的集合
$Q_{I_i \rightarrow G_{di}}$	指令与全局变量的关系
$Q_{I_i \rightarrow LC_{di}}$	指令与局部变量的关系
$Q_{I_i \rightarrow T_i}$	指令与导入函数的关系
P_i	调用 f_i 的所有函数的集合
$L_{P_i \rightarrow f_i}$	调用 f_i 的函数与 f_i 的调用关系
C_i	f_i 调用的所有函数的集合
$L_{f_i \rightarrow C_i}$	f_i 与 f_i 调用的函数的调用关系
M_{f_i}	f_i 在 PE 文件中的内存布局

定义 4 函数代码 shell 化各映射关系定义如表 2 所示。

表 2 代码 shell 化各映射关系定义
Table 2 The definition of mapping relationship in code shellization

标识符	定义
S_1	$G_{di} \rightarrow G_{newdi}$
S_2	$T_i \rightarrow T_{newi}$
S_3	$I_i \rightarrow I_{newi}$
S_4	$Q_{I_i \rightarrow G_{di}} \rightarrow Q_{I_{newi} \rightarrow G_{newdi}}$
S_5	$Q_{I_i \rightarrow T_i} \rightarrow Q_{I_{newi} \rightarrow T_{newi}}$

其中, G_{newdi} 是提取 f_i 中所使用的所有全局变

量形成的集合, T_{newi} 是提取 f_i 中所使用的所有 API 函数形成的集合, I_{newi} 是修改后所有指令的集合, $Q_{I_{newi} \rightarrow G_{newdi}}$ 是修改后的指令对提取出的全局变量的调用关系, $Q_{I_{newi} \rightarrow T_{newi}}$ 是修改后的指令对提取出的 API 函数的调用关系。

定义 5 函数内存布局随机化 $R: M_{f_i} \rightarrow M_{newf_i}$, 其中, M_{f_i} 是 f_i 在 PE 文件中的内存布局, M_{newf_i} 是 f_i 函数分离后的内存布局。

定义 6 函数执行动态链接化 $D = \{D_1, D_2\}$, 其中, D_1 是函数分离后调用 f_i 的函数对 f_i 调用关系的修改, D_2 是函数分离后 f_i 与 f_i 调用的函数的调用关系的修改。

定义 7 函数执行动态链接化各映射关系定义如表 3 所示。

表 3 动态链接化各映射关系定义
Table 3 The definition of mapping relationship in dynamic linking

标识符	定义
D_1	$L_{P_i \rightarrow f_i} \rightarrow L_{newP_i \rightarrow f_i}$
D_2	$L_{f_i \rightarrow C_i} \rightarrow L_{newf_i \rightarrow C_i}$

其中, $L_{newP_i \rightarrow f_i}$ 是调用 f_i 的函数与 f_i 函数分离后的调用关系, $L_{newf_i \rightarrow C_i}$ 是 f_i 函数分离后与 f_i 调用的函数的调用关系。

定义 8 程序 $P = \{f_1, \dots, f_i, \dots, f_n, G, T\}$, 其中, $G = \{G_i\}$, $T = \{T_i\}$, $0 < i \leq n$ 。

定义 9 代码碎片化 $F = \{S, R, D\}$, 其中, S 表示函数代码 shell 化, R 表示函数内存布局随机化, D 表示函数执行动态链接化。

代码碎片化模型可以形式化地表示为

$$P_{\text{fragment}} = F(P_{\text{original}})$$

其中, P_{original} 为原始程序, P_{fragment} 为代码碎片化之后的程序。

代码碎片化 F 由 3 个阶段组成: 函数代码 shell 化 S , 函数内存布局随机化 R 以及函数执行动态链接化 D 。每个阶段都对程序进行相应的修改, 具体算法如算法 1 所示。首先, 对中间语言 IR_{original} 提取 API 函数和全局变量; 然后, 对 API 函数调用方式和全局变量访问方式进行修改, 接着, 修改函数调用方式, 并将函数碎片进行分离, 同时生成动态地址表; 最后, 通过 Loader 函数实现对分离的函数碎片进行调用。

算法 1 代码碎片化算法

输入 原始中间语言文件 IR_{original}

输出 碎片化处理后的中间语言文件 IR_{fragment}

begin

1) **Array**<API_INFO*> T , **Array**<GV_INFO*> G ;

2) **Array**<Func_INFO*>FuncInfoList, **Array**<Table_INFO*>TableList;

3) **procedure** Shell(module)

4) GetGVData(module)/*全局数据聚合*/

5) RetriveGVDataCall()/*修改全局数据调用方式*/

6) **end procedure**

7) **procedure** GetGVData(module)

8) **foreach** GV &&ImportFn **in** module /
*构建全局变量列表 G 和 API 列表 T */

9) **GV_INFO***oGV = **new**
GV_INFO(GV);

10) Add(G , oGV);

11) **API_INFO***oImportFn = **new**
API_INFO(ImportFn);

12) Add(T , oImportFn);

13) **end foreach**

14) getAPIAddr(T)/*获取 API 函数地址*/

15) getGVData(G)/*获取全局变量内存数据*/

16) **end procedure**

17) **procedure** RetriveGVDataCall()

18) **foreach** gv_info **in** G

19) **foreach** GV_User **in** gv_info's
Users

20) /*得到该变量在全局数据
区中的位置*/;

21) **var** GV_Addr = Create-
GetElementPtr(GlobalStruct, index)

22) /*替换原全局变量的使用 */

23)

GV_User.ReplaceUseWith(GV, GV_Addr);

24) **end foreach**

25) **end foreach**

26) **foreach** api_info **in** T

27) **foreach** API_User **in** api_info's

Users

28) /*得到该变量在全局数据
区中的位置*/;

29) **var** API_Addr = Create-
GetElementPtr(GlobalStruct, index);

30) /*读取该地址存储的数据*/

31) **var** API_Addr_Value =
CreateLoadMemory(API_Addr);

32) /*替换原 API 的使用 */

33) API_User.ReplaceUseWith
(API, API_Addr_Value);

34) **end foreach**

35) **end foreach**

36) **end procedure**

37) **procedure** Seperate(module)

38) ModifyCallFunc(module)/*修改函数
 f_i 调用方式*/

39) GenerateDynAddrTable()/*初步生成
动态地址表*/

40) **foreach** CustomerFn **in** FuncInfoList

41) SeperateFrag-
ment(CustomerFn)/*分离功能函数碎片*/

42) **end foreach**

43) **end procedure**

44) **procedure** ModifyCallFunc(module)

45) CollectFunc(module, FuncInfoList)

/*收集每个功能函数保存在结构体中*/

```

46)      foreach CustomerFn in FuncInfoList
47)          foreach UserInst in CustomerFn's Users
48)              CreateCallOrInvoke(NewUserInst, Params)/*创建新的指令*/
49)              UserInst.RemovefromFunction()/*删除原指令*/
50)          end foreach
51)      end foreach
52) end procedure
53) procedure GenerateDynAddrTable()
54)      foreach CustomerFn in FuncInfoList
55)          GenerateTable(TableList) /*初步生成动态地址表*/
56)      end foreach
57) end procedure
58) procedure DynamicLink()
59)      addLoader()/*添加 Loader 函数*/
60)      foreach CustomerFn in FuncInfoList
61)          Loader(CustomerFn) /*动态调用功能函数*/
62)      end foreach
63) end procedure
end

```

4 碎片化技术流程设计

碎片化技术分为 3 个阶段, 如图 2 所示, 分别是: 代码 shell 化变形、函数碎片分离、碎片代码执行时动态链接。代码 shell 化变形能够实现函数代码 shell 化, 函数碎片分离能够实现函数内存布局随机化, 碎片代码执行时动态链接能够实现函数执行动态链接化。代码 shell 化变形将函数 f_i 中的全局变量 G_{di} 和导入表 T_i 中的 API 函数等全局数据进行提取并对其调用方式 $Q_{I_i \rightarrow G_{di}}$ 和 $Q_{I_i \rightarrow T_i}$ 进行修改。函数碎片分离将函数 f_i 进行分离, 修改其调用方式 $L_{P_i \rightarrow f_i}$ 和 $L_{f_i \rightarrow C_i}$, 添加 Loader 函数, 构建动态地址表。碎片代码执行时动态链接实现对分离后的函数碎片的动态调用, 通过使用添加的 Loader 函数和动态地址表实现对离散的函数碎片的调用。

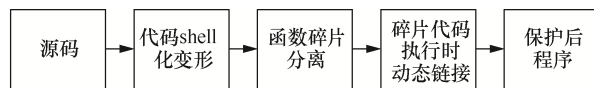


图 2 碎片化技术基本流程

Figure 2 The basic process of fragmentation technology

4.1 代码 shell 化变形

代码 shell 化变形需要解决两个关键问题。

1) 生成的碎片代码必须具有位置无关性, 即碎片代码可以在任意内存空间中运行, 碎片代码对内存地址的引用及跳转操作只依赖碎片代码的基址。

2) 碎片代码需要自行解决对外部函数的引用, 即碎片代码需要自行索引系统函数的地址。

代码 shell 化变形由全局数据提取和全局数据调用方式修改两个过程组成, 如图 3 所示。函数 f_i 中的全局数据由全局变量 G_{di} 和导入表 T_i 中的 API 函数组成。正常编译的 PE 文件中的全局数据以绝对地址的方式进行引用, API 函数的地址通过导入表进行获取, 函数 f_i 进行分离后对全局数据 G 使用原始的调用方式行不通。因此, 实现代码 shell 化变形需要从以下两个方面着手: 全局数据提取和调用方式修改。全局数据提取是将全局变量内存数据和 API 函数调用地址以内联宏代码的形式保存在构建的全局数据区结构体中。调用方式修改是将全局变量 G 和 API 函数集合 T 的调用方式修改为“基地址+索引”的形式。函数调用的全局变量分为两种: 一种是在程序中显式定义的全局变量, 如“int gv = 3”, 另一种是在程序中未显式定义的全局变量, 如“test”, 显式定义的全局变量和非显式定义的全局变量, 在中间语言层面都作为全局变量进行处理, 在对程序进行碎片化处理时, “test”与“int gv = 3”的调用方式一致, 都是通过“基址+索引”的方式进行调用。

全局数据的聚合包括两部分, 全局变量数据聚合 S_1 和导入函数信息聚合 S_2 。对于全局变量数据聚合 S_1 , 需要对中间语言文件内全局变量的类型和初始化值进行分析, 获取其对应的十六进制内存表示数据。对于导入函数信息聚合 S_2 , 需要从中间语言文件中提取导入函数名称, 生成相应的导入函数宏代码。全局数据区的构建分为两个部分, 分别是全局变量数据区构建和私有导入表

数据区构建。构建全局数据区, 首先对全局变量进行预处理, 消除复杂全局变量的定义形式; 然后依次遍历模块内的全局变量和导入函数, 将其分别存储在各自的数据结构链表中, 并转换成对应的宏代码, 构成全局数据区; 最后根据全局变量和导入函数在数据区中的排列位置, 构建相应的数据结构体。

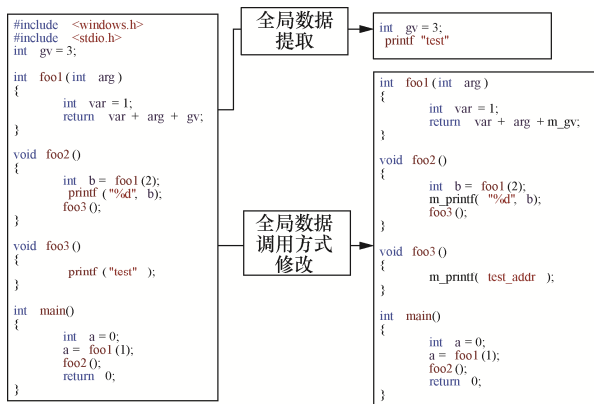


图 3 shell 化变形过程

Figure 3 The process of Shell deformation

在完成全局数据区聚合和相应结构体的构造后, 数据区存放着程序所需要使用的全局变量和导入函数地址, 结构体内存放着数据区中对应的类型信息。全局变量调用方式的修改过程 S_4 实现过程如下: 修改函数签名, 增加全局数据区指针参数, 通过“地址+索引”的方式访问指定的全局数据, 添加新的入口函数, 实现程序的正常编译。函数签名指函数的声明信息, 包括参数、返回值、调用约定等。此处所做的签名修改主要是对函数参数结构的修改, 为每个函数增加一个全局数据区指针参数, 在实际处理中, 对于线程函数、回调函数、变参函数等不能修改签名的函数, 在函数的入口指令前插入一条引导函数 `GetGlobalDataPtr` 的调用指令, 并使用该函数返回值作为全局数据区的指针值, 实现全局数据区地址的获取。修改签名后的函数在代码上与原函数完全一致, 因此在其内部的函数调用上, 是对原函数的调用而非新建签名函数, 所以还需对模块内整个函数调用关系进行重建, 建立新函数之间的调用关系。至此, 每个函数都拥有一个指向全局数据区的指针, 可以通过“地址+索引”的方式访问指定的数据。

API 函数的调用方式一般分为以下两种: 以被调用者的身份(Callee)引用和以参数方式引用。因此, API 函数调用方式的修改 S_5 分成两部分实现: 以被调用者的身份引用的函数, 在源码中表现为对一个函数的直接调用, 在进行函数指针的重定位时, 只需要在函数调用指令前增加一条指令, 获取函数在全局结构体中保存的地址, 并使用该地址对原指令中的函数名进行替换; 以参数方式引用的函数会产生重定位项, 对于需要进行重定位的函数, 为保证执行过程中动态获取函数的实际加载地址, 需要加入地址重定位函数, 以获取函数执行时的真实内存地址。

通过对 IR 文件的分析和变形, 分别完成了全局数据区和结构体的构建、函数重定位的处理和位置无关变形。在这个过程中, 生成的全局数据以内联汇编宏代码的方式保存在生成的头文件中, 同时引导文件内包括负责全局数据初始化和函数地址重定位的功能函数。因此, 头文件和引导文件需要同变形后的 IR 文件合并编译链接。经过函数签名修改后的 IR 文件, 默认的入口函数符号已经发生变化, 不再被链接器识别。因此, 在合并编译之前, 需要新建 IR 文件的入口函数, 该函数负责调用全局数据初始化函数和原入口函数。

4.2 函数碎片分离

通过分析中间语言, 遍历模块中的所有函数, 排除声明函数以及未被调用到的函数, 将所有函数 f_i 的函数名、函数类型信息保存到函数结构体数组中, 每一个数组项对应一个函数 f_i , 根据函数结构体数组, 建立动态地址表, 为分离后的函数 f_i 调用做好准备。遍历函数结构体数组, 依次将所有函数 f_i 的代码分别写入单独的中间语言文件中。

代码碎片分离由 3 个步骤完成: 函数调用方式修改、动态地址表初步构建和碎片代码分离。如图 4 所示, 首先, 在进行函数 f_i 调用方式修改之前, 需要收集所有函数, 组成函数结构体数组, 结构体中记录函数名、函数类型、函数返回值类型、函数调用者信息; 然后, 修改每个函数 f_i 的调用方式, 由直接调用改为通过使用添加的 Loader 函数获取函数 f_i 的地址的方式进行调用,

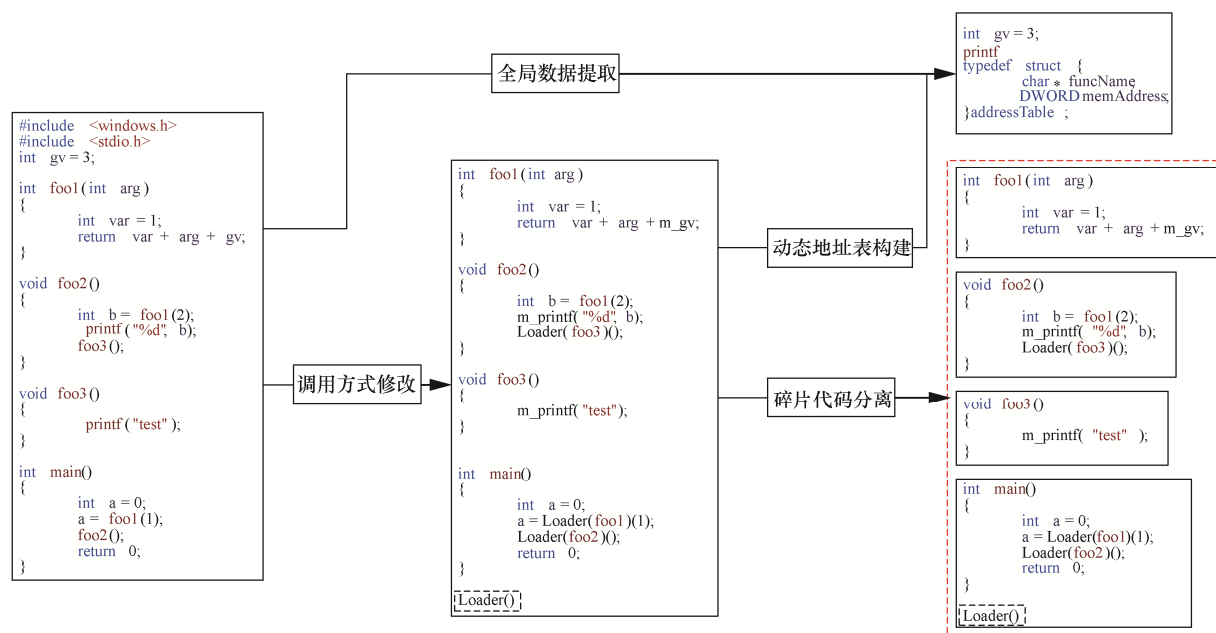


图4 代码碎片分离过程
Figure 4 The process of code fragmentation

此处的间接调用形式是通过 `Loader` 函数获取函数地址实现对函数碎片的调用。一方面, 通过 `Loader` 函数对函数碎片进行调用时, 可以对调用函数的参数进行检查, 避免缓冲区溢出等类型的攻击。另一方面, 虽然程序中多次调用 `Loader` 函数, 但 `Loader` 函数只有查表和加载这两个动作, 通过 `Loader` 函数实现间接调用可以实现函数与调用者之间的紧耦合关系的降耦, 对于 `Loader` 函数需要查询的动态地址表采取压缩加密进行保护, 降低动态地址表被攻击的可能性。根据函数结构体数组, 初步建立动态地址表, 每个表项由函数名和函数内存地址两项组成, 函数名根据函数结构体数组直接进行填充, 函数内存地址初始化为全零。完成函数 f_i 调用方式修改和动态地址表的初步构建, 最后将函数 f_i 分别保存在单独的中间语言文件中。

函数 f_i 分离需要考虑函数分离后代码碎片的数量和代码碎片的粒度, 分离的依据是调用函数 f_i 的函数数量, 如算法 2 所示。根据函数结构体数组可以获得函数 f_i 的调用函数的个数, 根据调用函数的个数将函数 f_i 分成调用函数数量大于 1 和调用函数数量等于 1 两类。对于第一类, 直接对该函数进行分离, 因为该函数被多个函数调用, 无法与任何一个调用函数进行合并处理, 因此作

为一个独立的碎片进行分离。对于第二类, 需要回溯到父节点, 对父节点的调用函数个数进行判断, 如果父节点的调用函数个数大于 1, 父节点无法与其父节点合并处理, 将该父节点与子节点作为碎片进行分离, 如果父节点的调用函数个数等于 1, 继续回溯其父节点, 直到回溯到 `main` 函数, 将该分支上的所有节点作为碎片进行分离。

算法 2 功能函数分离算法

输入 shell 化变形后 IR_{shell} 文件, 函数 f_i 结构体数组 `FuncList`
输出 `main` 函数 IR_{main} 文件, 函数 f_i 的独立 IR_i 文件

begin

1) **foreach** `Fn` in `FuncList`

2) `SeparaeProcess(Fn)`

3) **end foreach**

4) **procedure** `SeparateProcess(Func)`

5) **if** `FuncCallNum` > 1 /*处理调用函数数量大于 1 的情况*/

6) `separate(Func)` /*分离函数*/

7) **end if**

8) **if** `FuncCallNum` == 1 /*处理调用函数数量等于 1 的情况*/

9) `ParentNode = search(Func)` /*回

溯父节点*/

```

10)      if ParentNode is main
11)          separate(Func)
12)      end if
13)      if ParentNodeCallNum> 1/*处
理父节点调用函数数量大于 1 的情况*/
14)          separate(ParentNode&
&Func)/*分离父节点与其子节点*/
15)      end if
16)      if ParentNodeCallNum == 1
17)          SeparatePro-
cess(ParentNode) /*递归处理*/
18)      end if
19)  end if
20) end procedure
end

```

4.3 碎片代码执行时动态链接

函数 f_i 以代码碎片的形式保存, 主函数调用函数 f_i 时, 无法确定被调用的函数 f_i 是否存在内存中, 并且无法确定该函数在内存中的地址是否已知。如果函数 f_i 代码碎片不存在内存中, 需要对函数 f_i 代码碎片进行动态加载, 将函数 f_i 代码碎片加载到内存中, 并返回该代码碎片在内存中的地址, 以跳转执行的形式进行调用。

在完成动态地址表的初步构建之后, 函数代码碎片不是在加载的时候装入内存, 而是在调用的时候通过动态加载的方式装入内存, 并将内存地址保存在动态地址表中, 对于函数的引用方式 f_i 也需要进行相应的修改。正常函数的引用方式是通过 `call` 指令, 直接跳转到函数地址处进行执行, 经过分离之后的函数在主程序加载之后没有一起加载到内存之中, 需要调用独立的加载程序, 如图 5 所示。 f_i 函数分离后调用 f_i 的函数对 f_i 调用关系的修改 D_1 的实现如下: 当调用函数 f_i 时, 首先查找动态地址表中是否存在该函数 f_i 的内存地址, 如果存在, 直接跳转到内存地址处执行, 如果不存在, 调用加载函数, 将函数 f_i 加载到内存中, 同时将内存地址填入动态地址表, 跳转到内存地址处执行。函数 f_i 的调用方式由直接调用方式改为调用 `Loader` 函数加载函数 f_i , 并填写动态地址表、然后执行的间接调用方式。

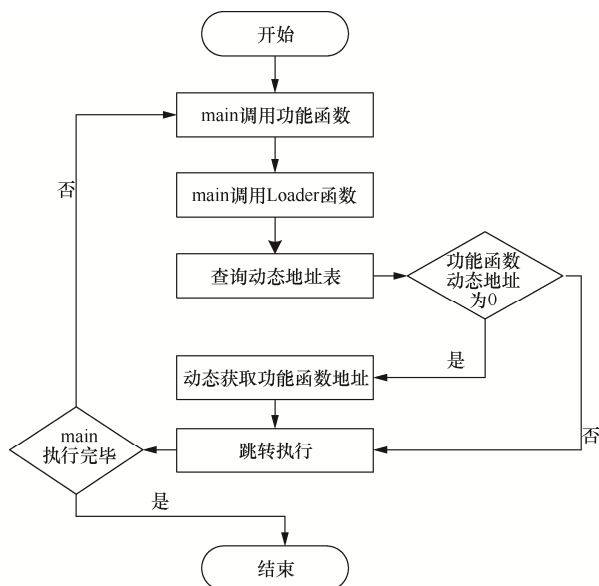


图 5 功能函数调用流程
Figure 5 The process of function call

动态地址表是一个结构体数组, 每一个结构体由函数名和函数内存地址两项组成, 函数名是记录函数 f_i 在代码中的名称, 函数内存地址是记录函数 f_i 代码碎片加载到内存中的地址, 可以直接跳转执行。函数 f_i 代码碎片索引号是记录代码碎片在代码段全局数据区中的位置, 这是 `Loader` 函数加载函数 f_i 的关键。

构建动态地址表分为两个步骤: 函数名写入过程和函数内存地址的动态生成过程。函数名写入过程: 将中间语言代码加载为模块, 构建动态地址表结构体数组, 通过中间语言获取所有函数 f_i 的函数名, 并对每一个函数生成一个索引, 将函数名直接填入构建的动态地址表结构体中。函数内存地址的动态生成过程: 函数内存地址在函数 f_i 代码碎片通过 `Loader` 函数加载到内存中执行时动态添加。

5 原型系统

根据代码碎片化技术设计流程, 设计实现了代码碎片化原型系统, 如图 6 所示, 该系统主要由 3 个模块组成: 代码 `shell` 化变形模块、代码碎片分离模块和代码碎片动态链接模块。

代码 `shell` 化模块实现对程序源码中的全局数据进行提取, 保存到统一的全局数据区中, 并实现对全局数据引用方式修改, 在全局数据区中

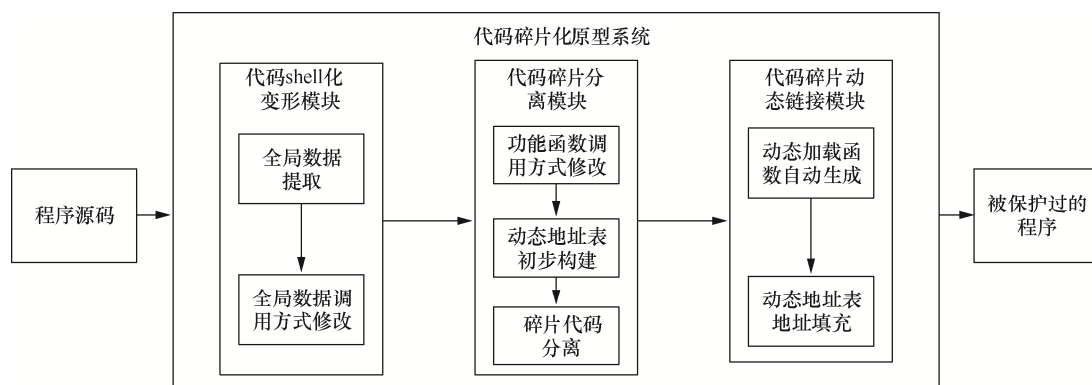


图6 代码碎片化原型系统

Figure 6 Code fragmentation prototype system

通过“地址+索引”的方式实现对全局数据的引用。

代码碎片分离模块实现对功能函数代码碎片的分离，是原型系统的关键。首先，在中间语言层面通过 LLVM 接口获取所有功能函数，并修改对功能函数的调用方式，由直接调用修改为间接调用；然后，根据功能函数列表构建动态地址表；最后，通过 LLVM 编译平台将功能函数实体分离为代码碎片，写入独立的中间语言文件中。

代码碎片动态链接模块完成代码碎片的动态链接，这需要在主函数之外自动添加一个 Loader 加载函数，负责对分离后的代码碎片进行动态调用，同时实现对动态地址表的动态填充。

6 实验

代码碎片化技术实现了对 PE 文件的结构布局的改变，通过该技术产生的类 PE 文件在代码段中存在主函数和加载函数两部分，其他功能函数以碎片的形式保存，通过间接调用的形式调用。因此，代码碎片化技术能够有效增加逆向攻击者逆向分析的难度。目前学术界和工业界尚未提出一套合理有效的方法来评测代码碎片化技术的强度，因此这里从程序执行的角度来说明该技术对软件的保护效果。

本文选取实验环境为 Windows7 操作系统，Intel I7 2.6 GHz CPU，4 GB 内存；Windows8 操作系统，Intel I7 2.6 GHz CPU，4 GB 内存；Windows10 操作系统，Intel I7 2.6 GHz CPU，4 GB 内存。选用的测试集由 rsa、rc4、MD5、aes、des、sm4、sha256、gzip、huffman、Huffcomprs 这 10

个程序组成，测试集的基本信息如表 4 所示，分别记录代码行数、函数数量和程序功能。

表4 测试集基本信息
Table 4 Basic information of the test set

测试源码程序	代码行数	函数数量	功能描述
rsa	678	52	非对称加密算法
rc4	184	13	流加密算法，密钥长度可变，对称加密算法
MD5	115	10	密码哈希函数，产生一个 128 bit (16 Byte) 的哈希值
aes	279	25	区块长度固定为 128 bit，密钥长度是 128 矩阵加密算法
des	203	5	密钥加密的块算法，区块长度 64 bit，密钥长度 56 bit
sm4	259	20	分组加密算法，分组长度与密钥长度均为 128 bit
sha256	125	18	密码哈希函数，产生一个 256 bit 的哈希值
gzip	4 514	144	文件压缩算法
huffman	225	21	可变字长编码方式
Huffcomprs	630	27	哈夫曼压缩算法

对程序进行碎片化保护后需要对每个程序的可用性进行测试，分别在 Windows7、Windows8、Windows10 操作系统下进行测试，代码碎片化保护之后所有程序功能均正常。

针对原型系统，采用 3 个指标对原型系统进行衡量，分别是导入函数变化、函数在内存中的布局与间接调用在程序中函数调用的比例。

函数导入表是记录程序所用到的其他模块的导出符号的地址、名称和序号。在程序执行时导

入函数是与位置有关的, 代码碎片化技术需要实现程序的位置无关化, 消除导入函数的位置有关性。通过比较碎片化前后导入函数数量的变化能够体现代码碎片化的位置无关化, 如表 5 所示。

表 5 导入函数数量变化
Table 5 Changes in the number of imported functions

测试源代码程序	碎片化前	碎片化后
rsa	52	0
rc4	45	0
MD5	46	0
aes	43	0
des	42	0
sm4	43	0
sha256	45	0
gzip	80	0
huffman	48	0
Huffcomprs	51	0

对于函数在内存中的布局, 以 gzip 程序为例, 记录程序执行 30 次 zip 函数在内存中的加载位置, 如图 7 所示, zip 函数在程序每次执行时在内存中的加载位置都不同, 体现函数内存布局的随机化。

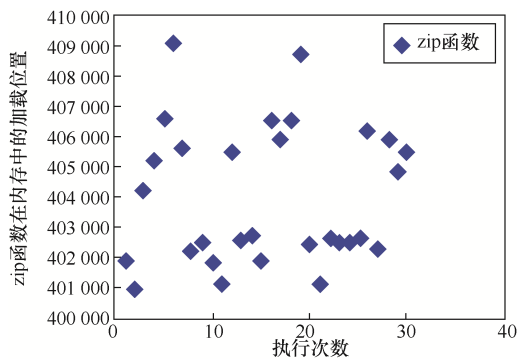


图 7 gzip 程序中 zip 函数的内存布局
Figure 7 The memory layout of the zip function in the gzip program

如表 6 所示, 碎片块数即程序中函数 f_i 的数量, 决定程序碎片化的离散程度, 碎片块数越多, 程序的离散程度越大, 程序碎片化越彻底。函数调用表项数与碎片块数保持一致, 函数调用表项对应碎片化之后的函数碎片, 实现对函数碎片的

调用。间接调用比例是间接调用数量占整个程序函数调用数量的比例。

表 6 程序中间接调用比例
Table 6 Proportion of indirect calls in the programs

测试源代码程序	碎片块数	函数调用表项数	间接调用比例 (碎片化前)	间接调用比例 (碎片化后)
rsa	37	37	10%	82%
rc4	4	4	40%	80%
MD5	3	3	33%	67%
aes	17	17	26%	76%
des	1	1	50%	67%
sm4	12	12	17%	69%
sha256	11	11	28%	49%
gzip	99	99	12%	78%
huffman	10	10	24%	54%
Huffcomprs	9	9	22%	56%

函数碎片是在执行时通过动态链接的形式进行调用的, 调用方式由直接调用修改为间接调用, 因此, 对程序进行代码碎片化后, 程序中间接调用的比例会有所增加, 如图 8 所示。

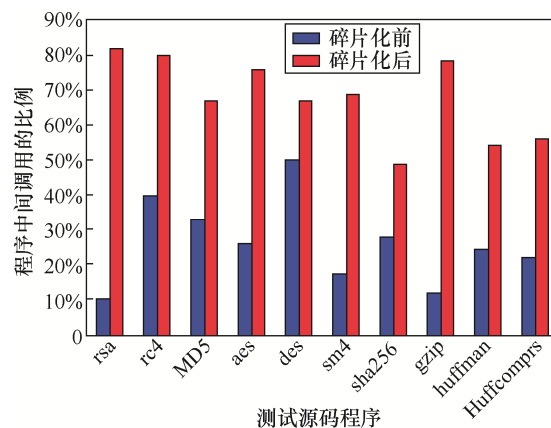


图 8 程序中间接调用比例变化
Figure 8 Changes in the proportion of indirect calls

函数间接调用能够有效增加程序逆向分析的难度, 但同时伴随着程序执行时间开销增加的问题, 如表 7 所示, 碎片化之后, 执行时间虽然有所增加, 但在可接受范围之内, 不会给用户体验带来较大不适, 其中, 时间开销增加比例 = $\frac{\text{碎片化后时间开销} - \text{碎片化前时间开销}}{\text{碎片化前时间开销}}$ 。

碎片化前时间开销

表7 碎片化前后时间开销变化
Table 7 Time cost changes before and after fragmentation

测试源码程序	碎片化前/ms	碎片化后/ms	时间开销增加比例
rsa	3 515	3 813	8.48%
rc4	15	16	6.67%
MD5	15	16	6.67%
aes	15	16	6.67%
des	31	32	3.23%
sm4	16	31	93.75%
sha256	15	16	6.67%
gzip	437	734	67.96%
huffman	31	47	51.62%
Huffcomprs	94	141	50.00%

7 结束语

代码碎片化技术对代码的保护是通过将功能函数分离并独立编译成能够在内存中任意位置执行的碎片,通过动态加载函数碎片实现对功能函数的动态执行。攻击者通过静态分析碎片化保护后的程序时,只能实现对主函数和加载函数的分析,无法分析以碎片存在的功能函数。当攻击者通过动态调试工具对程序进行动态调试时,只能跟踪调试主函数和加载函数的控制流,无法对碎片中的控制流进行跟踪,大大增加了程序逆向分析的难度。通过实验分析,碎片化技术能够实现函数碎片的内存布局随机化,碎片化保护后的程序间接调用比例有所增加,通过将函数的直接调用修改为间接调用,能够实现对函数碎片执行的动态链接。同时,代码碎片化技术有很多不完善的地方。控制流完整性分析在程序逆向分析中具有重要作用,未来将针对模糊函数与函数之间的控制流,增加程序的控制流完整性分析的难度作进一步研究。

函数单个调用链的函数组采取以非内联的形式进行处理,将每个函数单独碎片化,将函数组中的函数内联化为一个函数是减少程序时间开销的有效手段,是将这些函数单独处理的进阶措施。程序因间接调用的增加而造成的攻击面增加问题,将从两个方面进行研究:一方面,采取随机转换策略,随机选取部分函数进行碎片化,避免间接调用增加过多;另一方面,引入垃圾函数调

用,平衡因碎片化而造成的间接调用比例失衡。软件升级也是需要考虑的问题,对于以下载二进制补丁和汇编补丁的方式进行更新的软件,需要作进一步研究。

参考文献:

- [1] 许团, 屈蕾蕾, 石文昌. 基于结构特征的二进制代码安全缺陷分析模型[J]. 网络与信息安全学报, 2017, 3(9): 31-39.
XU T, QU L L, SHI W C. Analysis model of binary code security flaws based on structure characteristics[J]. Chinese Journal of Network and Information Security, 2017, 3(9): 31-39.
- [2] 孙博文, 黄炎裔, 温俏琨, 等. 基于静态多特征融合的恶意软件分类方法[J]. 网络与信息安全学报, 2017, 3(11): 68-76.
SUN B W, HUANG Y Y, WEN Q K, et al. Malware classification method based on static multiple-feature fusion[J]. Chinese Journal of Network and Information Security, 2017, 3(11): 68-76.
- [3] 李伟明, 邹德清, 孙国忠. 针对恶意代码的连续内存镜像分析方法[J]. 网络与信息安全学报, 2017, 3(2): 20-30.
LI W M, ZOU D Q, SUN G Z. Successive memory image analysis method for malicious codes[J]. Chinese Journal of Network and Information Security, 2017, 3(2): 20-30.
- [4] 王健. 基于完整性验证和壳的软件保护技术研究[D]. 太原: 中北大学, 2018.
WANG J. Research on software protection technology based on integrity verification and shell[D]. Taiyuan: North China University, 2018.
- [5] 杜春来, 孔丹丹, 王景中, 等. 一种基于指令虚拟化的代码保护模型[J]. 信息安全, 2017(2): 22-28.
DU C L, KONG D D, WANG J Z, et al. A code protection model based on instruction virtualization[J]. Netinfo Security, 2017(2): 22-28.
- [6] BALACHANDRAN V, EMMANUEL S, KEONG N. Obfuscation by code fragmentation to evade reverse engineering[C]//2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC). 2014.
- [7] BALACHANDRAN V, KEONG N W, EMMANUEL S. Function level control flow obfuscation for software security[C]//Eighth International Conference on Complex, Intelligence and Software Intensive Systems. 2014: 133-140.
- [8] LASZLO T, KISS A. Obfuscating C++ programs via control flow flattening[R]. 2009.
- [9] JUNOD P, RINALDINI J, WEHRLI J, et al. Obfusca-tor-LLVM—software protection for the masses[C]//2015 IEEE/ACM 1st International Workshop on Software Protection. 2015: 3-9.
- [10] DAVIDSON J, HILL J, KNIGHT J. Protection of software-based survivability mechanisms[C]//2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2001: 193-193.
- [11] WANG C, HILL J, KNIGHT J. Software tamper resistance: obstructing static analysis of programs[J]. 2000.
- [12] WANG C. A security architecture for survivability mechanisms[D]. Virginia: University of Virginia, School of Engineering and Applied Science, 2000.
- [13] 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.
JIANG H, LIU Y, WANG X. Code confusion technology research based on control flow[J]. Application Research of Computers, 2013,

30(3): 897-899.

- [14] 王旭. 基于目标代码的控制流混淆技术研究[D]. 北京: 北京邮电大学, 2013.

WANG X. Research on control flow obfuscation technology based on object code[D]. Beijing: Beijing University of Posts and Telecommunications, 2013.

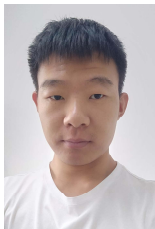
- [15] 王丰峰, 张涛, 徐伟光, 等. 进程控制流劫持攻击与防御技术综述[J]. 网络与信息安全学报, 2019, 5(6): 10-20.

WANG F F, ZHANG T, XU W G, et al. Overview of control-flow hijacking attack and defense techniques for process[J]. Chinese Journal of network and information security, 2019, 5(6): 10-20.

- [16] 乔向东, 郭戎潇, 赵勇. 代码复用对抗技术研究进展[J]. 网络与信息安全学报, 2018, 4(3): 1-12.

QIAO X D, GUO R X, ZHAO Y. Research progress in code reuse attacking and defending[J]. Chinese Journal of Network and Information Security, 2018, 4(3): 1-12.

[作者简介]



郭京城 (1994-), 男, 山东济南人, 信息工程大学硕士生, 主要研究方向为网络安全与软件保护。



舒辉 (1974-), 男, 江苏盐城人, 博士, 信息工程大学教授、博士生导师, 主要研究方向为网络安全、嵌入式系统分析与信息安全。



熊小兵 (1985-), 男, 江西丰城人, 博士, 信息工程大学副教授, 主要研究方向为网络安全。



康绯 (1972-), 女, 河南周口人, 硕士, 信息工程大学教授, 主要研究方向为网络信息安全。