

PPC 和 MIPS 指令集下二进制代码中函数参数个数的识别方法

尹小康, 刘臻, 刘龙, 刘胜利

(数学工程与先进计算国家重点实验室, 河南 郑州 450001)

摘 要: 函数参数个数的识别有助于函数原型的恢复, 是进行数据流分析以及其他安全分析的基础。为了提高对函数参数个数识别的准确率, 提出一种依据函数调用关系的投票机制来确定函数参数个数的算法——Findargs。Findargs 从 PPC 和 MIPS 指令集的函数调用特点出发, 利用函数调用关系和参数传递分析, 识别函数参数的个数, 为函数原型的恢复提供帮助。为了评估 Findargs 的识别效果, 选取大型的二进制文件进行了测试, 并与 radare2 进行了对比。实验结果表明, Findargs 具有更高的准确率。对于 PPC 指令集, 其准确率达到 90.3%; 对于 MIPS 指令集, 其准确率为 86%。

关键词: 静态分析; 函数调用分析; 参数个数识别; 投票机制

中图分类号: TP309

文献标识码: A

doi: 10.11959/j.issn.2096-109x.2020047

Function argument number identification in stripped binary under PPC and MIPS instruction set

YIN Xiaokang, LIU Liu, LIU Long, LIU Shengli

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

Abstract: The identification of the number of function argument contributes to the recovery of the function prototype and is the basis for data flow analysis and other security analysis. In order to improve the accuracy of the recognition of the number of function parameters, an algorithm (Findargs) which determines the number of parameters of the function according to the voting mechanism of the function call relationship was proposed. Findargs starts from the function call characteristics of PPC and MIPS instruction set, and uses function call relationship combined with argument pass analysis to identify the number of function arguments, which can help to recover function prototype. In order to evaluate the recognition effect of Findargs, a large binary file was selected and tested it with radare2. The experiments results show that Findargs has higher accuracy, and the accuracy rate for PPC instruction set reaches 90.3%. For MIPS instruction set, the accuracy rate is 86%.

Key words: static analysis, function call resolve, argument number identification, voting mechanism

收稿日期: 2019-07-29; 修回日期: 2019-10-13

通信作者: 刘胜利, mr.shengliliu@gmail.com

基金项目: 国家重点研发计划基金 (2016YFB0801505); 科技委基础加强项目 (2019-JCJQ-ZD-113)

Foundation Items: The National Key R&D Program of China (2016YFB0801505), Science & Technology Commission Foundation Strengthening Project (2019-JCJQ-ZD-113)

论文引用格式: 尹小康, 刘臻, 刘龙, 等. PPC 和 MIPS 指令集下二进制代码中函数参数个数的识别方法[J]. 网络与信息安全学报, 2020, 6(4): 95-103.

YIN X K, LIU L, LIU L, et al. Function argument number identification in stripped binary under PPC and MIPS instruction set[J]. Chinese Journal of Network and Information Security, 2020, 6(4): 95-103.

1 引言

二进制分析在安全研究中具有重大意义,在安全分析中的应用主要有:二进制代码审计^[1]、控制流完整性分析^[2]、污点分析^[3-4]、符号执行^[5-6]、漏洞修复^[7]、代码重用^[8]、漏洞挖掘^[9-11]等。在高级语言中,函数名、函数的参数个数、参数类型、函数的返回值等信息能够有效帮助理解函数的功能。源代码经过编译器编译丢失了如高级语言C/C++中的数据类型、数据结构、语义以及控制结构等信息,这给研究者分析二进制代码造成了诸多障碍。因此,如何从二进制代码中恢复出丢失的信息成为二进制分析的关键。对二进制程序进行逆向分析的第一步是对函数的识别,确定函数的起始位置、结束位置、函数的调用方式,以及函数的参数个数。二进制代码逆向分析的准确率影响后续安全分析的正确性。

函数参数个数的准确识别是进行函数原型识别的基础,是进行数据流分析以及其他安全分析的前提。由于精简过的二进制程序中的符号表等信息被移除,无法直接获得函数入口等相关信息^[12],增加了对二进制可执行文件进行分析的难度。Caballero等^[13]提出了X86指令集下基于动态分析的二进制函数接口识别方法。在无源代码和符号表的情况下,该方法对二进制代码进行动态分析,获取函数的原型,以达到重用二进制函数的目的。由于动态分析无法获取全面的信息,因此识别效果较差。radare2^[14]是最新的二进制分析工具,它拥有强大的分析功能,支持对不同架构语言的反汇编,在“aaa”命令中实现了对二进制函数分析的功能,在对PPC和MIPS指令集的二进制函数进行分析时,发现函数参数个数的识别效果比较差。

为提高函数参数个数的识别准确率,本文提出并实现了一种基于函数调用关系的函数参数个数识别算法——Findargs,用于识别PPC和MIPS指令集下二进制可执行代码中函数的参数个数。该方法结合PPC和MIPS指令集下函数参数的传递规则,通过分析函数调用时寄存器以及栈的使用情况,获取二进制函数的参数个数。为了提高识别的准确率,Findargs分析获得所有的函数调

用关系,然后对每个函数的调用者进行了分析,并通过投票机制,选取最有可能的参数个数。Findargs适用于对大型固件的分析,并且函数被调用的次数越多,准确率越高。本文的主要工作如下。

1) 分析了函数参数识别的问题,并提出了二进制可执行代码中二进制函数参数个数识别的概念。

2) 提出并实现了基于函数调用关系的函数参数个数识别算法——Findargs。Findargs使用静态分析的方法,无须动态执行二进制文件,适用性较好。

3) 选取大型的嵌入式系统固件,对该算法进行了评估,并与现有的二进制分析工具radare2进行了对比。实验结果表明,Findargs具有更高的准确率。

2 问题描述

2.1 问题定义

二进制函数是在二进制可执行文件中,通过函数识别技术识别出具有函数入口和出口的代码片段。与高级语言中的函数类似,二进制函数同样具有函数名、函数参数、函数返回值等。

参数传递指令序列是在函数调用指令前用于传递函数参数等信息的一段代码片段。

函数参数个数识别的目的是识别二进制函数中函数传递参数的个数,用于分析在二进制函数调用时参数的传递情况,进而分析程序的数据流的传递情况。

对二进制文件中的函数参数个数进行识别的基础是函数边界的识别。函数边界识别^[15]是给定包含 n 个函数的二进制文件输出完整的函数起始地址集合 $S(n) = \{s_1, s_2, s_3, \dots, s_n\}$ 。

函数参数个数的识别问题对于给定的二进制函数指令序列和函数的起始地址,输出函数的参数个数。对于精简二进制程序而言,输出所有二进制函数的参数个数对 $P(n) = \{(s_1, p_1), (s_2, p_2), (s_3, p_3), \dots, (s_n, p_n)\}$ 。

2.2 二进制函数调用特点

函数参数的传递一般有3种方式^[16]:栈传递、寄存器传递与全局变量进行隐式参数传递。通过栈传递参数,需要对参数的入栈顺序进行约定,

并对栈的平衡方式进行约定, 最终确定由调用者还是被调用者来恢复栈平衡。对于使用寄存器传递参数来说, 同样需要确定哪些寄存器用于传递参数, 以及参数传递的顺序, 并且需要确定当参数个数大于可用于传递参数的寄存器数时, 如何进行参数传递。对于全局变量进行隐式参数传递来说, 通过设置全局变量来共享变量, 在一个函数内对该变量的修改会影响其他函数中该变量的值。对于使用全局变量进行隐式参数传递, 需要进行数据流分析, 才能从二进制可执行代码中识别出全局变量。

函数参数调用中常用的调用约定共有 4 种:

`_cdecl` (C 规范)、`pascal`、`stdcall` 与 `fastcall`。表 1 详细展示了各个调用约定的参数传递方式和传递顺序以及平衡栈者。由表 1 可知, `_cdecl`、`pascal`、`stdcall` 均使用栈进行参数的传递, `fastcall` 利用处理器中寄存器的优势使用寄存器和栈来传递参数。

下面对常用的调用约定的参数传递方式进行详细介绍。

(1) 使用栈传递参数

栈作为一种先进后出的数据结构被广泛地应用。在调用函数时, 通过将参数压入栈中将参数传递给子程序。函数调用结束后, 对栈进行平衡操作, 将栈恢复到函数调用前的状态。当函数具有多个参数时, 需要对参数的传递顺序进行约定, 以及确定由函数调用者或者子程序来恢复栈的平衡。当函数具有多个参数时, `_cdecl` 和 `stdcall` 约定参数的传递顺序采用由右至左的顺序, `pascal` 约定参数传递顺序采用由左至右的顺序。对于栈的平衡方式, `_cdecl` 是调用者对栈进行平衡, 而 `pascal`、`stdcall` 和 `fastcall` 使用子程序对栈进行平衡。图 1 为利用栈进行函数参数的传递, 该参数传递方式是从右至左, 首先将第二个参数压入栈中, 然后将第一个参数压入栈中, 在使用参数时

通过 `ebp` 或者 `sp` 进行寻址。

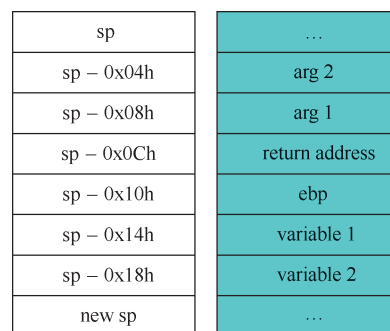


图 1 利用栈进行函数参数传递
Figure 1 Function arguments passing with stack

(2) 使用寄存器传递参数

使用寄存器进行参数传递具有快速高效的特点, 尤其当函数的参数个数较少时, 直接将参数放入寄存器中进行读取, 减少了一定的内存操作。处理器中拥有多个寄存器, 可以选择部分寄存器用于函数参数的传递。由表 1 可知, `fastcall` 采用寄存器和栈进行函数参数的传递。当函数参数较少时, 直接使用寄存器传递参数, 当函数参数超过可用参数寄存器时, 则将多余的参数通过栈来传递。使用栈传递参数时, 约定了参数的传递顺序和栈平衡操作者。对于 `fastcall`, 参数的传递顺序因编译器的不同而不同, 但都是通过子程序来平衡栈。

2.3 PPC 和 MIPS 指令集下的函数参数传递

表 2 列举了 PPC 和 MIPS 指令集中约定可用于传递参数的寄存器。在 PPC 指令集中, 寄存器 `r3` 作为函数的第一个参数, 剩余的参数分别依次放入 `r4 ~ r10`, 由此可知, PPC 指令集中可用寄存器最多传递 8 个参数。在 MIPS 指令集下, 分别使用编号为 4~7 的寄存器来传递参数, 名称分别为 `$a0 ~ $a3`, 因此, MIPS 指令集下可通过寄存器来传递参数的个数最大为 4, 多余的参数需要通

表 1 常见的函数调用约定
Table 1 Common conventions in function calling

约定类型	参数传递方式	参数传递顺序	平衡栈者
<code>_cdecl</code> (C 规范)	栈	从右到左	调用者
<code>pascal</code>	栈	从左到右	子程序
<code>stdcall</code>	栈	从右到左	子程序
<code>fastcall</code>	寄存器和栈	先寄存器后使用栈 (顺序由编译器确定)	子程序

过栈进行传递。

表2 PPC和MIPS指令集参数传递方式
Table 2 Function calling convention in PPC and MIPS ISA

指令集	传递方式	参数寄存器
MIPS	fastcall	\$a0 ~ \$a3
PPC	fastcall	r3 ~ r10

2.4 运行实例

下面以sub_801FDFA4和sub_600746B8函数参数的传递方式为例,具体分析PPC和MIPS指令集下函数参数的传递过程。在PPC指令下用于传递参数的寄存器为r3~r10,在调用函数sub_801FDFA4时使用了r3~r8寄存器,因此函数的参数个数为6。对于函数sub_600746B8,则为MIPS指令下的函数,用于传递参数的寄存器为\$a0~\$a3,在调用函数sub_600746B8时,在这段指令中使用了\$a1, \$a2, \$a3寄存器以及进行了两压栈“sw \$v0, 0x40+var_30(\$sp)”和“sw \$s2, 0x40+var_2C(\$sp)”。参数寄存器的使用具有以下事实:参数寄存器是按照顺序使用的,当后一个寄存器用于传递参数时,前一个寄存器必定用于传递参数。可知\$a0也用于传递参数,共有6个参数。PPC指令、MIPS指令下函数调用参数的传递分别如图2、图3所示。

```

mr    r3, r25
mr    r4, r28
mr    r5, r27
addi  r6, r1, 0x68+var_34
addi  r7, r1, 0x68+var_30
addi  r8, r1, 0x68+var_40
bl    sub_801FDFA4

```

图2 PPC指令下函数调用参数的传递
Figure 2 Function arguments passing in PPC instruction

```

loc_60107C3C:
li     $a1, 0x1F
li     $a2, 3
move   $a3, $zero
li     $v0, 2
sw     $v0, 0x40+var_30($sp)
jal    sub_600746B8
sw     $s2, 0x40+var_2C($sp)
bnez   $v0, loc_60107C84
lui    $a2, 0x615E

```

图3 MIPS指令下函数参数调用的传递
Figure 3 Function arguments passing in MIPS instruction

3 二进制函数参数个数识别

本节对二进制函数参数的识别算法进行详细介绍。二进制可执行文件的函数参数个数识别算法——Findargs的流程如图4所示。

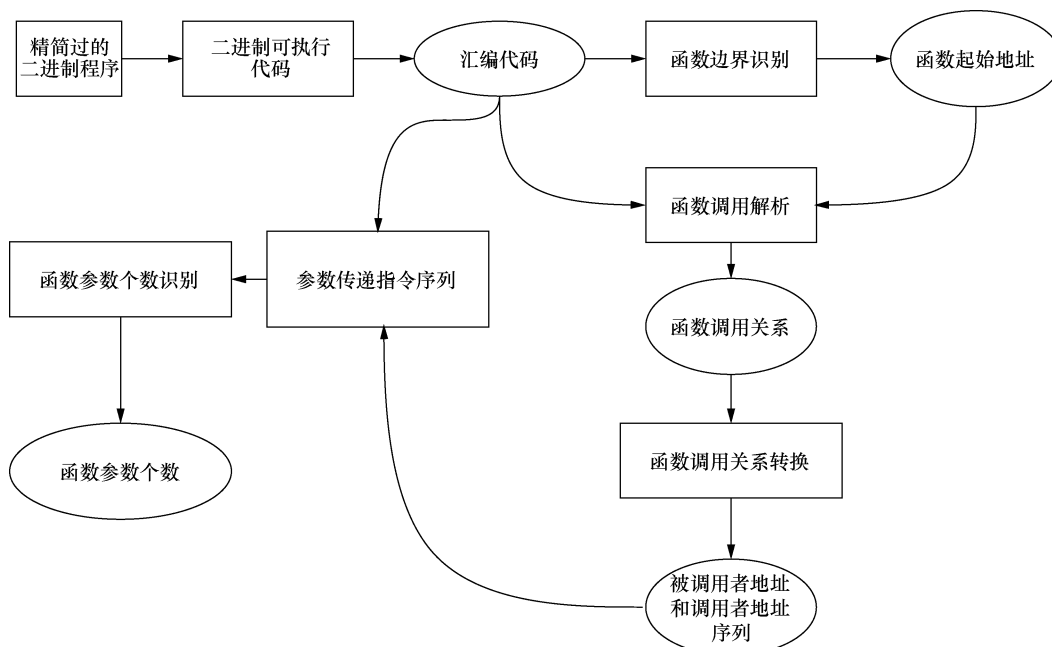


图4 函数参数识别算法——Findargs
Figure 4 Function argument identification algorithm—Findargs

3.1 函数边界的识别

对函数边界的识别是二进制分析的基础, 需采用的函数边界识别方法是 FRwithMBA^[17], 通过分析二进制可执行代码, 提取出函数的边界, 包括起始地址和结束地址, 用于函数调用关系的分析。

3.2 二进制函数调用关系的建立与转换

函数调用关系的分析在二进制逆向分析中具有重要作用, 如用于恶意代码分析^[18]与分类^[19]、计算代码相似性^[20-21]等, 本文将函数的调用关系用于函数参数个数的识别。函数调用关系的分析是函数控制流图生成以及研究数据流的基础, 函数调用关系图 (FCG, function call graph) 对理解大型程序有很大帮助, FCG 反映了函数的结构、可扩展性等信息。

按照分析目标的不同, 函数调用关系的分析可以分为源码层面和二进制层面; 按照分析方式的不同, 函数调用关系分析可以分为静态分析和动态分析。静态分析依赖于对目标语言的理解水平, 对目标语言中的函数调用指令分析的全面性, 能够遍历分析所有分支, 但是无法获取间接函数调用的关系。动态分析是借助函数运行时的信息进行分析, 在程序运行时, 能够获得相对于静态分析更多的信息, 但是对于程序未执行到的分支则无法进行分析。本文主要研究的是使用静态分析方式分析二进制可执行代码中的函数调用关系, 获取比较全面的函数直接调用关系, 用于后续的函数参数的识别。

与函数的边界识别一样, 对二进制代码中函数关系的分析可以采用线性扫描的方式或者递归下降回溯扫描的方式。对后续的参数识别只需要取函数调用的第一层关系即可。因此, 函数调用关系的分析采用静态分析的方式, 结合函数的边界, 采用线性扫描的技术对整个代码段进行分析。函数的边界为函数的起始和结束地址, 在每次解析函数调用关系时, 将整个函数的可执行代码取出进行解析, 通过解析生成格式为 {函数地址: 子程序 1, 子程序 2, 子程序 3, ..., 子程序 m } 的形式。在对所有的函数分析完成后, 将格式转换为 {子程序: 调用者 1, 调用者 2, 调用者 3, ..., 调用者 n } 的形式用于后续的函数参

数的识别。

3.3 基于调用关系的函数参数个数识别算法

函数调用前的一段指令是对参数进行传递的指令, 为了分析函数参数传递的情况, 只需对函数调用前的一段长度的指令进行分析, 这段代码为参数传递指令序列。函数调用前, 父函数会将调用子程序时需要的参数根据函数调用约定放入寄存器中或者压入栈中。对二进制函数指令序列进行扫描分析, 识别出其中的函数调用指令。选取函数调用前的一段指令序列作为函数参数传递指令序列, 对参数传递指令序列进行分析, 主要包括两个方面: 对参数寄存器的使用情况和栈操作的情况进行分析。对参数寄存器的使用情况分析主要是分析参数传递指令中使用了哪些参数寄存器, 对于 PPC 指令主要分析 r3 ~ r10 寄存器的使用情况, 对于 MIPS 指令则主要分析 \$a0 ~ \$a3 寄存器的使用情况。对于栈操作的指令分析, 主要分析在参数传递指令中是否进行了压栈操作来传递参数。

对于函数参数的传递, 有以下情况: 对于未使用栈进行参数传递的函数, 函数参数个数不会超过参数寄存器的个数, 因为当参数寄存器足够时, 必定会使用参数寄存器来传递参数; 并且当后一个参数寄存器用于传递参数时, 前一个参数寄存器必定也用来传递参数, 如在 PPC 指令下进行参数传递时, 如果 r5 寄存器被用于传递参数, 那么前面的 r3 和 r4 寄存器也一定被用于传递参数。

由于编译器的优化, 函数参数的传递位置可能会发生改变, 函数参数的传递会远离函数调用指令, 因此仅对单个的函数调用进行参数识别会出现识别错误的情况。为解决参数识别错误的问题, 本文提出采用多函数调用分析的方法, 即当该函数存在多个调用者时, 对每个调用者都进行函数参数传递分析, 获得该函数的参数个数, 然后使用基于投票的方法, 选择函数参数个数出现频率最多的为函数参数的个数。

通过对函数的调用关系的分析, 能够得到 {子程序: 调用者 1, 调用者 2, 调用者 3, ..., 调用者 n } 的函数调用关系。为了更准确地获得函数参数的个数, 本文首先分别计算函数的调用者中传

递的参数个数；然后基于投票的机制，对函数的参数个数进行投票，获得票数最多的为函数的参数个数；最后运用函数参数个数识别算法进行函数参数个数的识别。

函数参数个数识别算法——Findargs 如下。

输入 二进制代码序列 B 和函数调用关系集 $M = \{\{caller1: caller1, caller2, \dots, callerm\}, \dots\}$

输出 函数参数个数对 $P(f_i, n_j) = \{\{f_1: n_1\}, \{f_2: n_2\}, \dots, \{f_l: n_m\}\}$

1) 对参数变量进行初始化，设置函数参数个数数组 $S[]$ ，并对数组 $S[]$ 中元素初始化为 0；

2) 从函数调用关系集 M 中取出一个元素，进行函数参数个数的识别，当取出所有元素后，结束执行，进行步骤 5)；

3) 从函数关系中，依次取出函数调用者地址，并在二进制代码序列 B 中取出该函数的二进制指令 F_i ，若已经取出所有调用者，则转步骤 4)；

对函数指令序列 F_i 进行线性扫描解析，查找函数调用指令，取出函数调用指令前 m 条指令进行参数寄存器和栈使用情况的分析，计算出函数的参数个数 n ，则 $S[n] \leftarrow S[n] + 1$ ，继续步骤 3)；

4) 查找数组 $S[]$ 中最大数，则最大数的下标 j 即函数的参数个数，将 $\{calleri: j\}$ 添加到函数个数对集合 P 中，继续步骤 2)；

5) 输出集合 P 中的函数参数对。

3.4 运行实例

调用函数 Func_37326dc8 的函数关系如图 5

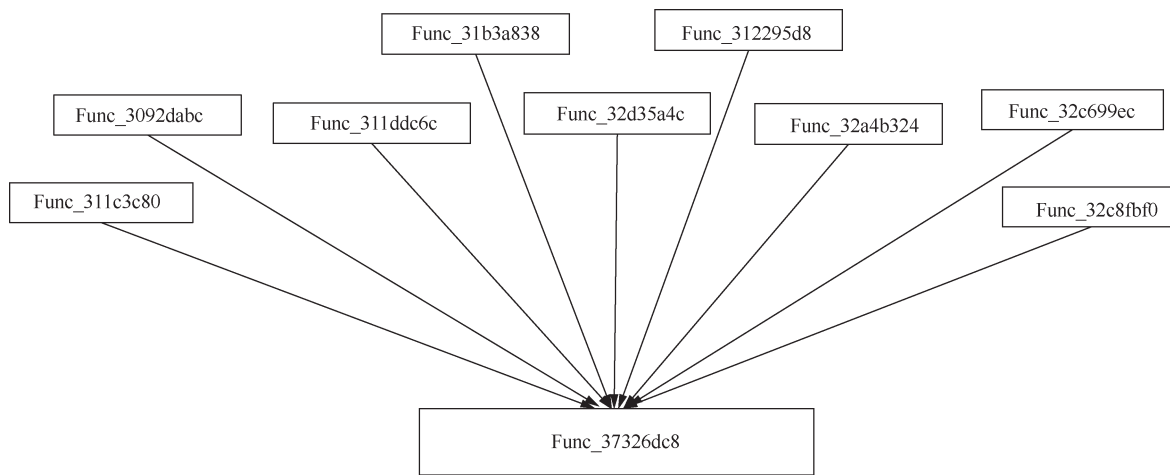


图5 调用函数 Func_37326dc8 的函数关系
Figure 5 The reference relation of function Func_37326dc8

所示。

由图 5 可知共有 9 个函数调用了该函数，为了准确识别函数 Func_37326dc8 的参数个数，对调用函数 Func_37326dc8 的这 9 个函数进行参数传递分析，最终选出现次数最多的数为函数的参数个数。其中，根据函数 Func_3092dabc 传递的参数分析函数 Func_37326dc8 的参数个数为 2，而对剩余的 8 个函数进行参数传递分析获得的函数 Func_37326dc8 的参数个数为 3，因此函数 Func_37326dc8 的参数个数为 3。经验证结果正确。

4 实验评估

4.1 实验设置

本节对 Findargs 的函数参数识别效果进行评估，并与现有的工具 radare2 进行对比。实验所用计算机的配置为 Intel R CoreTM 6-core 3.7 GHz i7-8700K CPU and 32 GB RAM。radare2 的版本为 radare2 2.7.0-git 18197 @ linux-x86-64 git.2.6.0-32-g3c8d7d53f。

4.2 实验结果

4.2.1 库函数的识别效果

为了验证函数参数个数的识别效果，需要选取拥有大量函数并且调用关系多样的二进制代码。本文选取了路由器固件 C2600-IPBA SEM-12.3(6f).BIN (PPC 指令集) 和 C2900-UNIVERSALK9 - M-15.7(3)M2.BIN (MIPS 指令集) 为测试样本，逆向分析得到二进制中的 C 语

言库函数,对函数参数个数的识别效果进行验证,C语言部分库函数的参数个数识别对比如表3所示。从表中可以看出该方法能够准确识别出函数的参数个数。

表3 C语言部分库函数参数个数识别效果对比
Table 3 Comparison of the identification result of function argument number in C library

函数名	指令集	函数地址	函数实际参数个数	识别出的参数个数
strcpy	MIPS	0x3732c5d0	2	2
	PPC	0x80ec0c70	2	2
strncpy	MIPS	0x3732d4d0	3	3
	PPC	0x80ec138c	3	3
strchr	MIPS	0x3732c534	2	2
	PPC	0x80ec0ba8	2	2
memcpy	MIPS	0x3732ab10	3	3
	PPC	0x80ec0630	3	3
memmove	MIPS	0x3732ad40	3	3
	PPC	0x80ec074c	3	3
bcopy	MIPS	0x37327590	3	3
	PPC	0x8047a460	3	3
strcat	MIPS	0x3732c4f0	2	2
	PPC	0x80ec0854	2	2
alps_copy	MIPS	0x3406ab04	3	3
	PPC	—	—	—
copy_mem	MIPS	0x30fc4bec	4	4
	PPC	0x80478994	4	4

“—”表示未在 PPC 指令集的二进制文件中发现此函数

由表3可知,Findargs对于常用库函数的识别的效果较好,原因是库函数调用的次数比较多,并且函数的参数个数较少,使用参数寄存器能够满足所有参数的传递。

4.2.2 未知函数的识别效果

为了能够更准确地评估Findargs的识别效果,本文从路由器固件中C2600-IPBASE-M-12.3(6f).BIN和C3725-I-M-12.3(1a).BIN逆向分析获得函数的参数个数,分别使用Findargs和radare2进行了参数个数的识别,识别结果如表4所示。

实验结果表明,与radare2相比,Findargs具有更高的准确率。对于PPC指令集,Findargs准确率达到90.3%;对于MIPS指令集,识别的准确率较低,达到86.0%。在实验中,Findargs的输出结果为函数的起始地址和参数个数对。而radare2首先通过命令“aaa”对二进制可执行文件进行分析,然后通过“afl”命令输出到文本中,并使用Python脚本对输出的结果进行提取以获得函数的参数个数。

4.2.3 函数调用次数对识别效果的影响

为了更好地展示Findargs的识别效果和函数调用次数对参数个数识别准确率的影响,本文依据函数的调用次数对这些函数进行了分类统计。对PPC和MIPS指令集分别统计了上述的300个函数,分类统计结果为:对于PPC指令集,函数调用次数大于或等于2次的共有139个,函数调用次数大于或等于3次的共有77个,函数调用次数大于或等于4次的共有54个;对于MIPS指令集,函数调用次数大于或等于2次的共有144个,函数调用次数大于或等于3次的共有105个,函数调用次数大于或等于4次的共有76个。函数调用次数对函数参数个数识别的影响如表5所示,其中,多调用分析是对所有的调用者进行分析识别准确的函数个数,单调调用分析是只分析其中的一个函数调用进行分析识别准确的函数个数。

表6展示了不同调用次数下的函数参数个数的识别准确率。由表中数据可知,函数调用次数越多,函数参数个数的识别准确率越高,并且多调用分析的效果好于单调调用分析。

4.2.4 函数参数个数对识别效果的影响

为了研究函数参数个数对参数识别准确率的影响,本文对不同参数个数的函数进行了统计(如表7所示),分别计算了识别的准确率(如表8所示)。由表可知,函数的参数个数一般小于6个。整体上来说,在此范围(参数小于6)内函数识别的准确率较高,由于PPC有8个参数寄存器,基本

表4 函数参数识别结果
Table 4 The identification result of function argument number

指令集	函数个数	radare2 准确识别个数	radare2 准确率	Findargs 准确识别个数	Findargs 准确率
PPC	300	86	28.7%	271	90.3%
MIPS	300	11	3.7%	258	86.0%

表 5 函数参数个数识别效果随调用次数的变化情况

Table 5 Changes in the function argument number identification result with different quantity of function calls

指令集	函数调用次数								
	≥ 2			≥ 3			≥ 4		
	总函数	多调用分析	单调用分析	总函数	多调用分析	单调用分析	总函数	多调用分析	单调用分析
PPC	139	124	124	77	72	72	54	52	51
MIPS	144	126	119	105	94	88	76	68	58

表 6 不同调用次数下函数参数的识别准确率

Table 6 Accuracy of function argument identification under different quantity of function calls

指令集	函数调用次数					
	≥ 2		≥ 3		≥ 4	
	单调用分析	多调用分析	单调用分析	多调用分析	单调用分析	多调用分析
PPC	89.2%	89.2%	93.5%	93.5%	94.4%	96.3%
MIPS	82.6%	87.5%	83.8%	89.5%	76.3%	89.5%

表 7 函数识别个数随函数参数个数的变化

Table 7 Changes in function identification with function argument number

指令集	函数参数个数																	
	0		1		2		3		4		5		6		7		≥ 8	
	函数个数	准确识别	函数个数	准确识别	函数个数	准确识别	函数个数	准确识别	函数个数	准确识别	函数个数	准确识别	函数个数	准确识别	函数个数	准确识别	函数个数	准确识别
PPC	35	30	94	83	71	65	45	40	34	34	13	13	3	3	1	0	4	3
MIPS	9	8	97	89	75	58	46	41	30	30	15	10	6	5	11	11	11	6

表 8 函数识别准确率随参数个数的变化

Table 8 Changes in function identification accuracy with function argument number

指令集	参数个数变化时的识别准确率								
	0	1	2	3	4	5	6	7	≥ 8
PPC	85.7%	88.3%	91.5%	88.9%	100.0%	100.0%	100.0%	0.0%	75.0%
MIPS	88.9%	91.8%	77.3%	89.1%	100.0%	66.7%	83.3%	100.0%	54.5%

能够满足函数进行参数的传递。对于 MIPS 指令而言,在参数个数大于 4 时,需要借助栈进行参数的传递,使用栈进行参数的传递比使用参数寄存器传递复杂,因此识别的准确率略低。后续将考虑结合数据流分析追踪参数的使用情况来提高 MIPS 指令集下函数参数个数识别的准确率。此外,经过分析函数调用的位置同样影响函数参数个数的识别准确率,位于函数头部的函数调用由于借用了上层函数的调用参数,没有明显的传参行为,因此对存在此类函数调用的函数的识别准确率较低。

5 结束语

函数的参数个数作为函数原型中的一个重要

特征,能够辅助后续的二进制程序分析。本文提出了二进制函数参数个数分析算法——Findargs,来识别 PPC 和 MIPS 指令集下二进制代码中函数的参数个数。实验结果表明,Findargs 具有较高的准确率,对于 PPC 指令集的二进制可执行文件,识别准确率达到 90.3%;对于 MIPS 指令集的二进制可执行文件,识别准确率稍低,为 86.0%。并且,本文与现有的工具 radare2 进行了对比,结果表明 Findargs 具有更高的准确率。

由于 Findargs 依赖于对参数寄存器的分析,对于_cdecl、pascal、stdcall 仅使用栈进行参数传递的方式适用性较差,但对多函数调用关系依据投票来选择最准确的参数个数的机制可以应用到其他 3 种

参数传递规范。这3种参数个数分析方法可以应用到对 fastcall 参数传递方式的分析。当使用 fastcall 进行参数传递时,函数调用传递的参数个数超过参数寄存器个数时,参数传递的方式与上述3种方式相同。后续将考虑通过数据流分析追踪参数的使用情况,来提高参数个数识别的准确率。

参考文献:

- [1] DURDEN T. Automated vulnerability auditing in machine code[J]. Phrack Magazine, 2007, 64.
- [2] ABADI M, BUDI M, ERLINGSSON Ú, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Transactions on Information and System Security (TISSEC), 2009, 13(1): 4.
- [3] CLAUDE J, LI W, ORSO A. Dytan: a generic dynamic taint analysis framework[C]//International Symposium on Software Testing and Analysis. 2007: 196-206.
- [4] NEWSOME J, SONG D X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[C]//NDSS. 2005: 3-4.
- [5] STAATS M, PASAREANU C. Parallel symbolic execution for structural test generation[C]//The 19th International Symposium on Software Testing and Analysis. 2010: 183-194.
- [6] LI H, KIM T, BAT-ERDENE M, et al. Software vulnerability detection using backward trace analysis and symbolic execution[C]//2013 International Conference on Availability, Reliability and Security. 2013: 446-454.
- [7] PERKINS J H, KIM S, LARSEN S, et al. Automatically patching errors in deployed software[C]//The ACM SIGOPS 22nd Symposium on Operating Systems Principles. 2009: 87-102.
- [8] SHACHAM H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)[C]//ACM Conference on Computer and Communications Security. 2007: 552-561.
- [9] SONG D, BRUMLEY D, YIN H, et al. BitBlaze: a new approach to computer security via binary analysis[C]//International Conference on Information Systems Security. 2008: 1-25.
- [10] SHOSHITAISHVILI Y, WANG R, SALLS C, et al. Sok:(state of) the art of war: offensive techniques in binary analysis[C]//2016 IEEE Symposium on Security and Privacy (SP). 2016: 138-157.
- [11] 李珍, 邹德清, 王泽丽, 等. 面向源代码的软件漏洞静态检测综述[J]. 网络与信息安全学报, 2019, 5(1): 1-14.
LI Z, ZOU D Q, WANG Z L, et al. Survey on static software vulnerability detection for source code[J]. Chinese Journal of Network and Information Security, 2019, 5(1): 1-14.
- [12] MENG X, MILLER B P. Binary code is not easy[C]//The 25th International Symposium on Software Testing and Analysis. 2016: 24-35.
- [13] CABALLERO J, JOHNSON N M, MCCAMANT S, et al. Binary code extraction and interface identification for security applications[R]. California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2009.
- [14] Radare. The official radare2 book (1st edition)[EB].
- [15] BAO T, BURKET J, WOO M, et al. BYTEWEIGHT: learning to recognize functions in binary code[C]//The 23rd USENIX Security Symposium. 2014: 845-860.
- [16] 段钢. 加密与解密 (第四版) [M]. 北京: 电子工业出版社, 2018: 104-105.
DUAN G. Encryption and decryption (4th edition) [M]. Beijing: Publishing House of Electronics Industry, 2018: 104-105.
- [17] YIN X, LIU S, LIU L, et al. Function recognition in stripped binary of embedded devices[J]. IEEE Access, 2018, 6: 75682-75694.
- [18] HU X, CHIUH T, SHIN K G. Large-scale malware indexing using function-call graphs[C]//The 16th ACM Conference on Computer and communications security. 2009: 611-620.
- [19] HASSEN M, CHAN P K. Scalable function call graph-based malware classification[C]//The Seventh ACM on Conference on Data and Application Security and Privacy. 2017: 239-248.
- [20] HUANG D X, TANG Y, WANG Y, et al. Toward efficient and accurate function - call graph matching of binary codes[J]. Concurrency and Computation: Practice and Experience, 2018, 31(21).
- [21] KONG D, YAN G. Discriminant malware distance learning on structural information for automated malware classification[C]//The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2013: 1357-1365.

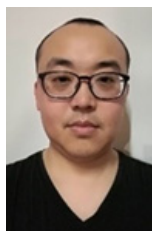
[作者简介]



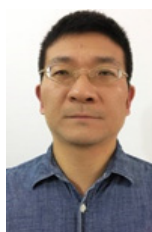
尹小康 (1994—), 男, 河南周口人, 数学工程与先进计算国家重点实验室博士生, 主要研究方向为网络空间安全和逆向工程。



刘鑫 (1998—), 女, 安徽合肥人, 主要研究方向为网络空间安全。



刘龙 (1983—), 男, 河南尉氏人, 数学工程与先进计算国家重点实验室副教授, 主要研究方向为网络空间安全和机器学习。



刘胜利 (1973—), 男, 河南周口人, 博士, 数学工程与先进计算国家重点实验室教授, 主要研究方向为网络空间安全。