

基于函数摘要的二进制程序污点分析优化方法

杨盼, 康绯, 舒辉, 黄宇垚, 吕小少

(数字工程与先进计算国家重点实验室, 河南 郑州 450001)

摘要: 污点分析是一种常用的软件分析方法, 在信息安全领域有较多的应用。现有的二进制程序动态污点分析大多采用指令级插桩的分析方法, 通常会产生巨大的性能开销, 使得程序执行效率大幅度降低, 难以在复杂恶意样本和商业软件分析环境中有效应用。为了提升污点分析效率, 降低指令级插桩分析带来的性能损耗, 使污点分析更加广泛地应用在软件分析中, 提出了基于函数摘要的二进制程序污点分析优化方法。所提方法使用函数污点传播规则代替指令污点传播规则, 以减少数据流传播分析次数, 有效提升污点分析效率。对于函数摘要, 提出了函数摘要的定义; 研究了不同函数结构的摘要生成算法。在函数内部, 针对非循环结构, 设计了路径敏感的分析方法; 针对循环结构, 设计了有限迭代的分析方法, 将这两种分析方法相结合, 解决混合结构函数的函数摘要生成。在函数摘要生成算法研究的基础上, 进一步设计实现了由函数摘要生成模块、数据流记录模块、污点分析模块3个部分构成的通用污点分析框架FSTaint。对FSTaint的分析效率进行了评估, 在分析真实APT恶意样本中, FSTaint的污点分析效率是libdft的7.75倍, 分析效率较高; 在准确性方面, FSTaint相对libdft在传播规则的准确性、完备性等方面也有所提高。

关键词: 函数摘要; 污点分析; 数据流分析; FSTaint

中图分类号: TP393

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2023026

Binary program taint analysis optimization method based on function summary

YANG Pan, KANG Fei, SHU Hui, HUANG Yuyao, LYU Xiaoshao

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

Abstract: Taint analysis is a popular software analysis method, which has been widely used in the field of information security. Most of the existing binary program dynamic taint analysis frameworks use instruction-level instrumentation analysis methods, which usually generate huge performance overhead and reduce the program execution efficiency by several times or even dozens of times. This limits taint analysis technology's wide usage in

收稿日期: 2021-08-30; 修回日期: 2022-06-22

通信作者: 康绯, mathswork@163.com

基金项目: 国家重点研发计划(2019QY1300)

Foundation Item: The Key R&D Program of China (2019QY1300)

引用格式: 杨盼, 康绯, 舒辉, 等. 基于函数摘要的二进制程序污点分析优化方法[J]. 网络与信息安全学报, 2023, 9(2): 115-131.

Citation Format: YANG P, KANG F, SHU H, et al. Binary program taint analysis optimization method based on function summary[J]. Chinese Journal of Network and Information Security, 2023, 9(2): 115-131.

complex malicious samples and commercial software analysis. An optimization method of taint analysis based on function summary was proposed, to improve the efficiency of taint analysis, reduce the performance loss caused by instruction-level instrumentation analysis, and make taint analysis to be more widely used in software analysis. The taint analysis method based on function summary used function taint propagation rules instead of instruction taint propagation rules to reduce the number of data stream propagation analysis and effectively improve the efficiency of taint analysis. For function summary, the definition of function summary was proposed. And the summary generation algorithms of different function structures were studied. Inside the function, a path-sensitive analysis method was designed for acyclic structures. For cyclic structures, a finite iteration method was designed. Moreover, the two analysis methods were combined to solve the function summary generation of mixed structure functions. Based on this research, a general taint analysis framework called FSTaint was designed and implemented, consisting of a function summary generation module, a data flow recording module, and a taint analysis module. The efficiency of FSTaint was evaluated in the analysis of real APT malicious samples, where the taint analysis efficiency of FSTaint was found to be 7.75 times that of libdft, and the analysis efficiency was higher. In terms of accuracy, FSTaint has more accurate and complete propagation rules than libdft.

Keywords: function summary, taint analysis, data flow analysis, FSTaint

0 引言

污点分析^[1]作为一种常用的软件分析方法,在信息安全领域有较多的研究和应用,典型的应用包括恶意代码分析^[2-3]、软件脆弱性分析^[4-7]和网络攻击行为检测^[8-9]等。

二进制程序动态污点分析技术虽然已有多年的研究,但是动态跟踪污点数据带来的程序执行效率严重下降,使得现有污点分析框架很难应用于商业软件或者大规模的恶意样本分析。在动态污点分析中,大多会使用到动态二进制插桩程序技术(如 Pin^[10]、Dynamorio^[11]等),对于任意一条指令,污点分析执行时间包括 3 个部分,即程序指令执行时间、插桩程序执行时间、污点传播分析时间,其中后两部分时间开销往往是第一部分的数倍。因此,指令级的污点分析会使程序执行效率大幅度降低。例如, Dytan^[12]的分析耗时是程序执行时长的 30 倍, libdft^[13]的分析耗时是程序执行时长的 3.72 倍。

目前的污点分析效率提升研究聚焦于生成快速分析路径^[14]、并行化处理^[15-16]等方面,这些优化方法大多在指令级分析的基础上进行优化,只能实现有限的效率提升。

在动态污点分析过程中,由于部分函数功能单一,污点传播关系明确,如加密算法函数、压缩函数、内存复制函数等,如果已知该函数的污

点传播摘要,则不需要对该函数进行指令级插桩,只需要依据函数污点传播关系摘要即可得出该函数内部的污点传播过程,有效提高污点分析的效率。文献[15,17]虽然使用函数摘要的方式优化污点分析,但只针对少量外部导入的 libc 库函数设置污点传播规则,没有规范的函数污点传播摘要定义,也未能较好地解决二进制代码的函数摘要问题。

针对以上问题,本文提出了基于函数摘要的二进制程序污点分析优化方法,通过实验对比发现,该方法可以有效提升二进制程序污点分析效率。本文的主要贡献如下。

1) 借鉴符号分析中的符号化概念,提出了污点符号的概念,基于污点符号提出了函数污点传播摘要的定义和描述方法。

2) 提出了基于路径敏感和有限迭代相结合的函数污点传播摘要生成方法,该方法很大程度上兼顾了函数污点传播摘要的准确率和摘要生成的效率。

3) 在函数摘要的基础上,设计实现了面向恶意代码分析的污点分析框架 FSTaint,实验表明, FSTaint 的污点分析效率是 libdft 的 7.75 倍。

1 二进制程序函数污点传播摘要的定义

在动态污点分析中,由于部分函数功能单一,污点传播关系明确,如果已知该函数的污点传播关系,则不需要对该函数进行指令级插桩和指令

数据流分析，只需要依据函数污点传播关系即可得出该函数的污点传播结果。以 AES 加密算法为例，通过实验统计发现，使用 AES 128 bit 加密算法加密 16 byte 数据过程中，执行的 CPU 指令数高达 18 000 余条，如果采用指令级分析，则需要进行 18 000 余次的污点传播分析，而实际产生的污点传播过程只是将密钥和明文受污染的数据传播到密文。如果已知该加密算法的函数污点传播关系，则通过少量的污点传播分析即可。基于此，本文提出了函数污点传播摘要（以下简称“函数摘要”）的优化思路，通过静态分析生成函数摘要，使用函数摘要优化动态污点分析过程，以提升污点分析效率。

在函数摘要生成过程中，由于还未引入污点数据，无法确定函数的输入数据是否是受污染的数据，因此函数的每一个输入数据都可能存在两种污点状态，即“没有被污染”和“被污染”。以图 1 的代码为例，函数 func 返回值的污点状态与参数 a、b 有关，此时输入参数有 4 种可能的污点状态，函数返回值有两种可能的污点状态（见表 1）。为了表示所有可能的污点状态，参考符号执行^[1]中，使用符号化的方法来描述所有可能的输入状态，本文提出了“污点符号”的概念。在符号分析中，“符号”用来表示被符号化对象的所有可能取值，称其为“值符号”。而“污点符号”是用来描述分析对象可能存在的污点状态，污点符号有两种取值 0 和 1，分别表示“没有被污染”和“被污染”两种状态。

```
int func(int a, int b)
{
    int t;
    t = a + b;
    t = t * t;
    return t;
}
```

图 1 函数示例
Figure 1 Function example

定义 1 （函数摘要）函数 func 的函数摘要如式(1)所示。

$$TS_{func} = \{(x_0, S_0), (x_1, S_1), \dots, (x_{n-1}, S_{n-1})\} \quad (1)$$

其中， $\{x_0, x_1, \dots, x_{n-1}\}$ 表示函数 func 的 n 个输出变量； (x, S) 为输出变量 x 的污点传播关系，其中 S 是一个集合，表示所有可能影响 x 污点状态的污点符号集合。 x 在函数入口处的状态为 $S_x = \{t_x\}$ ，即在入口处污点状态只与自身有关。对于污点传播关系 (x, S) ， $S = \{t_0, t_1, \dots, t_k\}$ ，计算 x 在函数 func 执行结束后的污点状态的方法如式(2)所示。程序语句与对应污点符号集合运算规则如表 2 所示。

$$t_x = t_0 | t_1 | \dots | t_k \quad (2)$$

表 1 函数 func 的输入输出污点关系

Table 1 func's func input and output taint relationship

参数 a 污点状态	参数 b 污点状态	返回值污点状态
否	否	否
是	否	是
否	是	是
是	是	是

表 2 程序语句与对应污点符号集合运算规则

Table 2 Procedure statement and corresponding taint symbol set operation rules

程序语句	污点符号集合	运算类型
$x = \text{imm}$	$S_x = \{\}$	赋值（常数）
$x = y$	$S_x = S_y$	赋值
$x = \bullet x$	$S_x = S_x$	$\bullet$ 为任意一元运算
$x = y \circ z$	$S_x = S_y \cup S_z$	$\circ$ 为任意二元运算

一般来说，函数摘要的结果关注函数输出变量与输入变量的关系，而局部临时变量只作为中间变量使用和中间分析过程使用，不作为函数污点传播摘要的输出结果。图 1 中函数 func 的污点传播摘要为 $TS = \{\text{ret}, \{t_a, t_b\}\}$ ，其中 ret 为函数 func 的返回值，而函数中的 t 只是临时变量，不作为函数摘要集合的一部分。

本文定义的函数摘要，直观地给出了函数输入变量与输出变量之间的数据流传播关系，没有多余的中间过程，相对于指令级跟踪，从函数汇编指令的角度来看，函数摘要有以下 3 方面的优化。

1) 在函数的汇编实现中，通常包括函数栈帧的保存与恢复，存在栈 Cookie 保护的函数，需要栈 Cookie 的保存与检查等，此类操作会产生数据流，但通常不是数据流分析所关心的内容，函数

摘要相对于指令数据流分析，避免了对此类数据流传播的跟踪。

2) 对于传播关系 (x, S) ， S 中的所有变量都会参与计算，最终得到变量 x ，在计算变量 x 的污点状态时， S 中的每个变量都只需要使用一次，而在实际代码中，为了得到 x ， S 中的变量可能需要被多次使用，进行指令级的分析时，则需要跟踪每一条计算指令，而通过传播关系 (x, S) 计算 x 的污点状态，不存在对 S 中的同一变量多次计算的情况。

3) 在汇编指令中进行数据计算和转移时，需要使用寄存器作为中间“桥梁”过渡使用，这会产生额外的数据传播，相对于指令级的分析，函数摘要避免了对数据传播中间过渡部分的分析。

2 函数污点传播摘要生成方法

常见的数据流分析中，一般存在两种分析方法，即路径敏感和路径不敏感的分析方法。通常，路径敏感的分析方法具有较高的准确度，但分析复杂度较高；而路径不敏感的分析方法则相反。为了保证一定的分析准确度和较小的分析复杂度，本文提出了基于路径敏感的非循环结构摘要生成方法和基于有限迭代的循环结构摘要生成方法。

2.1 基于路径敏感的非循环结构摘要生成方法

在函数内部，一般存在两种类型的控制流结构，即非循环结构和循环结构。对于非循环结构，函数内部执行路径数量有限，可以容易地找出函数的所有路径。而对于循环结构，则可能存在“路径爆炸”的情况。本节讨论基于路径敏感的非循环结构摘要生成方法。

不同的函数输入会导致不同的执行路径，在不同的执行路径下，可能会产生不同的污点传播结果。以图 2 中的代码为例，函数 `func` 内部存在两条执行路径①②③⑤和①②④⑤，如图 3 所示，分别对应不同的污点传播结果，如表 3 所示。

针对非循环结构，本文采用路径敏感的数据流分析方法。通过探索非循环结构函数内部所有可能的执行路径，分析不同路径下污点传播关系，

采用污点传播关系与执行路径相关联的污点摘要表示方法。在摘要生成过程中，使用符号执行的分析方法，将函数输入符号化表示，在符号执行中构造变量计算表达式的同时，构造污点传播关系 (x, S) 。与常见符号执行不同的是，此处的数据流分析需要遍历非循环结构中所有可能的执行路径，由于所有的执行路径都需要执行，遇到分支跳转时，不需要进行约束求解，分别执行分支跳转的每个目标地址即可，这在很大程度上提升了分析过程中符号执行部分的效率。

```
int func(int a, int b)
{
  ① int result = 0;
  ② if(a*a + b*b > 3*a*b)
  ③   result = a;
    else
  ④   result = b;
  ⑤ return result;
}
```

图 2 函数代码示例
Figure 2 Function code example

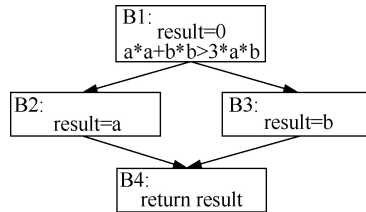


图 3 图 2 代码的 CFG 示意
Figure 3 CFG for code in figure 2

表 3 函数 `func` 的摘要
Table 3 The summary of `func`

执行路径	污点传播摘要
①②③⑤	$\{ (d, \{t_a, t_c\}) \}$
①②④⑤	$\{ (d, \{t_b, t_c\}) \}$

以图 2 中代码 2 为例，通常符号分析遇到②中的约束条件 $a*a+b*b>3*a*b$ 时需要进行约束求解，以求得分别满足接下来两种分支情况的解，再进行后续的分析。而在本文的路径敏感分析方法中，这个约束条件后面的两种分支都需要进行分析，因此遇到②的约束条件不需要进行约束求

解，分别让符号分析强制执行后续的③或者④即可。一般来说，约束求解是符号执行过程中时间复杂度很高的一部分，避免这一部分的时间开销，可以有效提升符号执行的效率。

通过上述方法生成的污点摘要在实际使用过程中，需要根据函数的实际执行路径，选择与执行路径相对应的摘要。确定函数实际执行路径，最直接的思路就是对函数内的所有基本块进行插桩，记录实际的执行路径。

通过分析可以看到，以图3中的流程为例，基本块 B1、B4 为两条执行路径共同需要经过的基本块，B2 和 B3 出现在不同的路径中。对基本块 B1 或 B4 进行插桩记录，对于确定函数具体执行哪一条路径没有实际价值。如果实际执行过程中确定经过了基本块 B2，则一定可以说明此时的程序执行路径为 B1、B2、B4；反之，则一定是另一条路径。因此，只需对基本块 B2 进行插桩记录，就可以知道该函数被调用的情况下，具体执行的是哪一条路径。为了确定该函数的具体执行路径，原本需要对4个基本块进行插桩记录，通过分析发现，只需要对1个基本块进行插桩即可，相对对所有基本块进行插桩，显然这种方式效率更高。

为了减少插桩次数，提升分析效率，需要找到可以反映函数内部执行路径的最小基本块集合，而这个最小基本块集合中的基本块对不同执行路径应该有较好的区分。为了描述基本块对不同路径的区分程度，本文引入基本块路径区分度的概念。

定义2（基本块对经过该路径区分度）表示在一个非循环结构的函数内，一个基本块对函数执行路径是否经过该基本块的区分程度，基本块 b 的路径区分度如式(3)所示。

$$Dp_b = \frac{1}{2} - \left| \frac{1}{2} - \frac{1}{n} \sum_{i=0}^{n-1} b_i \right| \quad (3)$$

其中， n 为函数内路径的数量， b_i 为基本块 b 是否经过第 i 条路径，经过为 1，不经过为 0。

$Dp \in \left[0, \frac{1}{2} \right]$ ， Dp 越大，则对应的基本块路径区分度越大，否则越小。

基本块路径区分度反映一个基本块对函数执

行路径的划分程度，如果函数内的所有路径都经过基本块 B ，则基本块 B 对该函数执行路径的区分度为 0，可以认为不具备区分度；如果函数内一半路径经过了基本块 B ，另一半没有，则该基本块的路径区分度为 1/2，具有最大区分度。在函数执行过程中，一个基本块只有这两种状态，被执行或者没有被执行，假设某个函数内有 8 条执行路径，那么最理想的情况就是使用 3 个不同基本块根据是否被执行来表示这 8 种状态，而这 3 个基本块都需要有最高的基本块路径区分度 1/2。

定义3（最小特征基本块集合）MinBBSet 为满足 $\forall p, p' \in \mathbf{MP}, p' : p \rightarrow S, p' \in \mathbf{MPS}$ 的最小集合。

其中， $\mathbf{MP}$ 为函数内的所有路径与函数内所有基本块 B 是否被执行的对应关系构成的矩阵，称为“路径基本块矩阵”，该矩阵为一个 01 矩阵， p 为 $\mathbf{MP}$ 中一条路径对应的向量； S 为 B 的一个子集， $\mathbf{MPS}$ 为 p 在基本块集合 S 上的“路径-基本块矩阵”； p' 为 p 在 S 上的向量， S 需满足 $\mathbf{MPS}$ 中不存在相同的 p' 指向不同的 p 。MinBBSet 为满足上述条件的最小 S 。

通过对 MinBBSet 内的基本块进行插桩记录，可反映出该函数的实际执行路径，MinBBSet 以外的其他基本块执行情况不需要插桩记录。

本文设计了 MinBBSet 生成算法，其基本思想为，通过贪心的思想，逐步找出 Dp 最大的基本块，构成 MinBBSet。具体过程如算法1所示。

算法1 MPS 生成算法

输入 $\mathbf{MP}, \mathbf{BBL_List}$ ，

输出 MinBBSet， $\mathbf{MPS}$

1) **global** MinBBSet = {}

2) **function** DIVIDE_MP(mp, bbl)

3) **for** $in\ mp$ **do**

4) **if** $mp[p][bbl] = 1$ **then**

5) $sub_mp0.add(mp[p])$

6) **else**

7) $sub_mp1.add(mp[p])$

8) **end if**

9) **end for**

10) **return** sub_mp0, sub_mp1

```

11) end function
12) function FIND_MINBB_IN_ONE_MP(mp)
13)   if len(mp) ≤ 1 then
14)     return
15)   end if
16)   B = MAX(calc_dp(mp))
17)   MinBBSet.add(B)
18)   submp0, submp1 = DIVIDE_MP(mp, B)
19)   FIND_MINBB_IN_TWO_MPS(submp1,
submp1)
20) end function
21) function FIND_MINBB_IN_TWO_MPS
(submp0, submp1)
22)   if len(submp0) ≤ 1 and len(submp1) ≤ 1
then
23)     FIND_MINBB_IN_ONE_MP(submp0)
24)     FIND_MINBB_IN_ONE_MP(submp1)
25)   return
26) end if
27)   B = MAX(calc_dp(submp0), calc_dp(submp1))
28)   MinBBSet.add(B)
29)   submp00, submp01 = DIVIDE_MP(submp0, B)
30)   submp10, submp11 = DIVIDE_MP(submp1, B)
31)   FIND_MINBB_IN_TWO_MPS(submp00,
submp01)
32)   FIND_MINBB_IN_TWO_MPS(submp10, submp11)
33) end function
34) FIND_MINBB_IN_ONE_MP(MP)
35) for BinMinBBSet do
36)   MPS.add(MP[B])
37) end for

```

在该算法中, 最坏情况下的 MinBBSet 包含了所有的基本块, 每找到一个特征基本块, 就需要计算一次所有基本块的 D_p 值, 则该算法在最差情况下的时间复杂度为 $O(n^2)$ 。

以图4的流程为例, 该函数共有4条路径, 其对应的MP如图5中第一个矩阵所示。首先, 计算每个基本块的 D_p 值, 可以发现基本块 B1 为

D_p 值最大的基本块之一, 选择基本块 B1 对 M 进行划分, 得到 M_1 、 M_2 。其次, 分别计算 M_1 、 M_2 中每个基本块的 D_p 值, 并将 M_1 与 M_2 中对应基本块的 D_p 值进行相乘, 得到相乘后 B4 的 D_p 值最大, 选择 B4 对 M_1 、 M_2 分别进行划分, 划分后的矩阵只剩一个路径, 划分结束。最后, 选择 B1、B4 组成最小特征基本块集合, 构成 MPS, 发现 B1、B4 这两个基本块的4种状态可以代表函数内部4条不同路径。

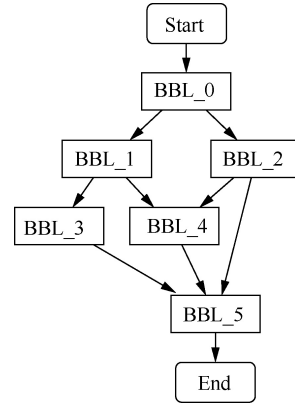


图4 非循环结构流程

Figure 4 Schematic diagram of acyclic structure flow chart

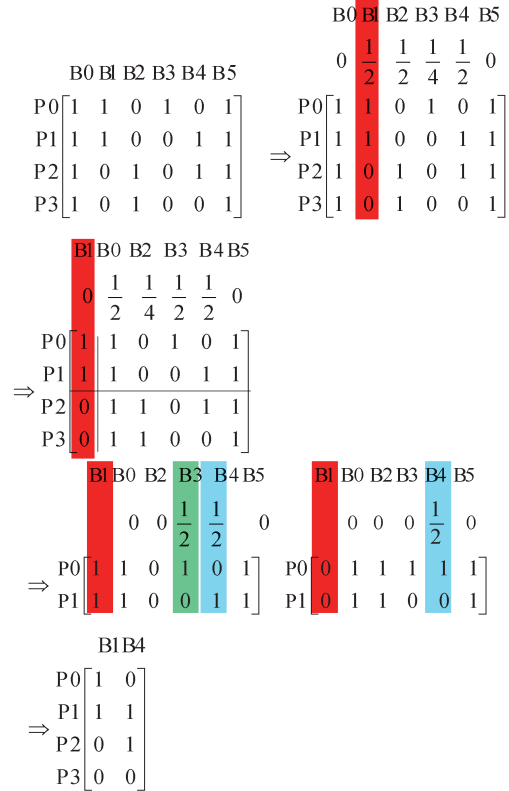


图5 MinBBSet 查找算法流程

Figure 5 MinBBSet search algorithm flow diagram

2.2 基于有限迭代的循环结构摘要生成算法

对于存在循环结构的代码块，直接分析所有执行路径可能会出现“路径爆炸”的情况。为了缓解“路径爆炸”导致分析无法进行的情况，本文提出在输入有穷的情况下，针对循环结构通过有限次迭代生成摘要的方法。

格 (Lattice) 是指在给定偏序关系 $(P, \leq)$ 上，满足 $\forall a, b \in P, a \vee b \in P, a \wedge b \in P$ ，称偏序关系 $(P, \leq)$ 为格。如果交或并关系只满足其中之一，则为半格 (Semilattice)。本文定义的函数摘要的数据流关系满足半格的约束条件。数据域值为 (x, S) ，偏序集 P 属于函数内所有参与数据传播的变量 $\{x_1, x_2, \dots, x_n\}$ 的幂集。

当两个基本块交汇到一个基本块时，对应的 $\vee$ 操作如式(4)所示。

$$IN[B_3] = OUT[B_1] \vee OUT[B_2] \quad (4)$$

其中， S 的 $\vee$ 关系满足集合的并关系，即

$$S_1 \vee S_2 = S_1 \cup S_2 \quad (5)$$

对于变量 x 来说，

$$(x, S_1) \vee (x, S_2) = (x, S_1 \cup S_2) \quad (6)$$

对于程序语句 $S: x = \dots$ ，

$$OUT[S] = \text{gen}_x \vee (IN[S] - \{(x, S_x)\}) \quad (7)$$

使用 $\text{val}(x)$ 取出污点传播关系中 x 指向的集合。

$$\text{gen}_x = \begin{cases} (x, \text{val}(x)), S: x = \circ x \\ (x, \text{val}(a) \cup \text{val}(b)), S: x = a \otimes b \\ (x, \text{val}(a)), S: x = a \end{cases} \quad (8)$$

其中， $\circ$ 表示一元计算， $\otimes$ 表示二元计算，当 a 为常量时， $\text{val}(a) = \{\}$ 。

首先，对其单调性进行分析：① 对于情况 1，没有改变 x 的值；② 对于情况 2、情况 3， a 或 b 增大时 $\text{val}(a) \cup \text{val}(b)$ 或 $\text{val}(a)$ 增大，因此， gen_x 单调递增。

其次，如果函数的输入是多个有限长度的缓冲区，满足有穷性，如果包含长度不确定的缓冲区，如 `strcpy` 函数，在没有上下文的情况下，该函数的输入长度任意，则输入为无穷。

对于单调且有穷的函数，其污点传播摘要生成算法满足格上的不动点原理，可以求出最大不

动点，即通过有限次迭代得到稳定的污点传播结果作为污点传播摘要。

基于有限迭代的函数摘要生成算法如算法 2 所示。

算法 2 基于有限迭代的函数污点摘要生成算法

输入 CFG, BBL_List

输出 OUT

1) $OUT[\text{entry}] = \{\}$

2) **for** bbl **in** BBL_List **do**

3) $OUT[\text{bbl}] = \{\}$

4) Add bbl to Worklist;

5) **end for**

6) **while** Worklist is not empty **do**

7) Get B from Worklist

8) $\text{old_out} = OUT[B]$

9) $IN[B] = \bigcup_{a \text{ predecessor of } B} OUT[a]$

10) $OUT[B] = \text{gen}_B \cup (IN[B] - \text{kill}_B)$

11) **if** $OUT[B] \neq \text{old_out}$ **then**

12) Add all successors of B to Worklist

13) **end if**

14) **end while**

其中， gen_B 为对于基本块 B 新生成的污点映射关系， kill_B 为需要删除的污点映射关系。由于数据流分析中，输入不变的情况下，输出也不变，则其后继基本块也不需要分析，使用 Worklist 保存输入产生变化的基本块，将其作为后续分析的对象，避免对输入无变化的基本块的再次分析，这样可以减少迭代分析的基本块数量，提升分析效率。由格上的不动点理论可知，在单调有穷的情况下，通过有限次迭代算法可以停止，最终得到最大上限。在该算法中，最差情况就是将所有的输入数据映射到输出数据且每次迭代只能产生一个数据的映射关系，则时间复杂度为 $O(nm)$ ，其中 n 为输入数据长度， m 为输出数据长度。

以图 6 的 TEA 加密算法为例 (算法流程如图 7 所示)，其加密部分需要进行 32 轮迭代，但是通过本文设计的摘要生成算法，只需 3 轮迭代即可得到 TEA 算法的函数摘要。TEA 算法摘要生成过程和最终的 TEA 算法函数摘要结果如图 8、图 9 所示。

该算法同样适用于汇编语言或中间语言（IR）。

```
void EncryptTEA(unsigned int *v0, unsigned int *v1, unsigned
int* key)
{
    unsigned int y = *v0;
    unsigned int z = *v1;
    unsigned int sum = 0;
    unsigned int delta = 0x9e3779b9;
    for (int i = 0; i < 32; i++)
    {
        sum += delta;
        y += ((z << 4) + key[0]) ^ (z + sum) ^ ((z >> 5) +
key[1]);
        z += ((y << 4) + key[2]) ^ (y + sum) ^ ((y >> 5) +
key[3]);
    }
    *v0 = y;
    *v1 = z;
}
```

图 6 TEA 算法源代码

Figure 6 The source code of TEA algorithm

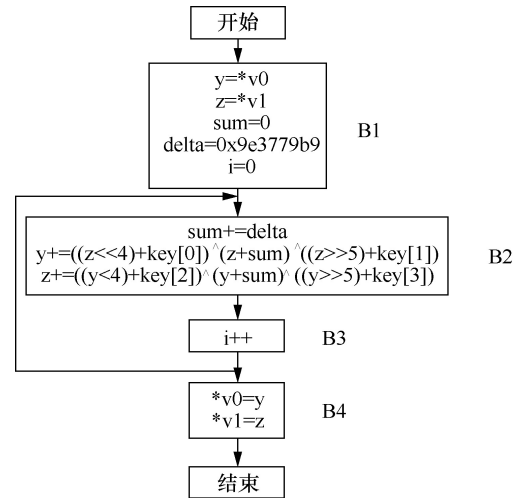


图 7 TEA 算法流程

Figure 7 The process of TEA algorithm

在迭代过程中，遇到多个基本块交汇到一个基本块时，算法采用“并”的运算规则，可能出现“过污染”的情况。“过”污染的流程示例如图 10 所示， x 的污点传播关系可能与真实情况不一致，算法生成结果为 $(x, \{k\})$ ，真实结果可能为 $(x, \{ \})$ 或 $(x, \{k\})$ 。

第一轮	输入	输出
B1	OUT(entry): $\{ \}$	$(y, \{ *v0 \}), (z, \{ *v1 \})$
B2	$\text{OUT}(B1) \cup \text{OUT}(B3): (y, \{ *v0 \}), (z, \{ *v1 \})$	$(y, \{ *v0, *v1, \text{key}[0], \text{key}[1] \}), (z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$
B3	$\text{OUT}(B2): (y, \{ *v0, *v1, \text{key}[0], \text{key}[1] \}),$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$	$(y, \{ *v0, *v1, \text{key}[0], \text{key}[1] \}), (z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$
B4	$\text{OUT}(B3): (y, \{ *v0, *v1, \text{key}[0], \text{key}[1] \}),$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$	$(y, \{ *v0, *v1, \text{key}[0], \text{key}[1] \}), (z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$ $(*v0, \{ *v0, *v1, \text{key}[0], \text{key}[1] \}), (*v1, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$
第二轮	输入	输出
B2	$\text{OUT}(B1) \cup \text{OUT}(B3): (y, \{ *v0, *v1, \text{key}[0], \text{key}[1] \}),$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$	$(y, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$
B3	$\text{OUT}(B2):$ $(y, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$	$(y, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$
B4	$\text{OUT}(B3):$ $(y, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$	$(y, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \}),$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$ $(*v0, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$ $(*v1, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$
第三轮	输入	输出
B2	$\text{OUT}(B1) \cup \text{OUT}(B3):$ $(y, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$	$(y, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$ $(z, \{ *v0, *v1, \text{key}[0], \text{key}[1], \text{key}[2], \text{key}[3] \})$

（注：红色字体表示本次迭代过程相对于上次产生了变化）

图 8 TEA 算法函数摘要生成过程

Figure 8 Function summary generation process of TEA algorithm

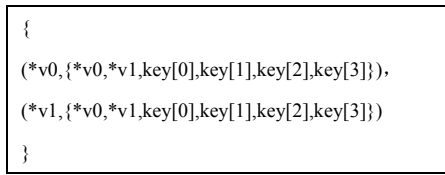


图 9 TEA 算法函数摘要结果

Figure 9 Result of the function summary of TEA algorithm

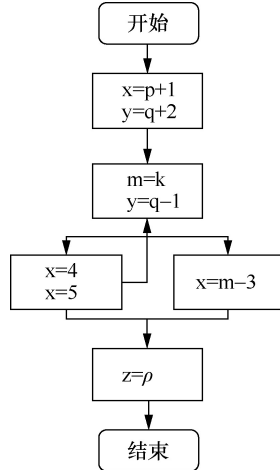


图 10 “过”污染的流程示例

Figure 10 Example of an over-taint flow chart

因此，有限迭代的摘要生成方法存在以下不足：① 路径不敏感，存在一定的过污染情况；② 只能处理数据域“有穷”的情况，对于 `strcpy` 这种输入变量长度可能“无穷”的情况，由于数据流方程无法收敛，最终无法终止迭代过程。

2.3 混合结构函数摘要生成方法

在污点分析过程中，为了在一定程度保证分析精度，并提升污点分析效率，本文将路径敏感分析与有限迭代相结合：对于非循环结构，采用路径敏感的分析方法；对于循环结构部分，采用迭代的分析方法。

在混合结构的函数摘要生成过程中，首先，在函数控制流图（CFG）的邻接矩阵上使用深度优先遍历查找循环结构；然后，将各个循环结构划分为独立的区域；对非循环结构分析每条路径上的污点传播过程，对循环结构使用算法 2 进行迭代分析；最后，输出不同执行路径下的函数摘要。

如图 11 所示，程序可分为两部分，第一部分为 {BBL_0, BBL_1, BBL_2} 组成的非循环结构，第二部分为 {BBL_3, BBL_4, BBL_5} 组成的循环结构。首先，分别计算 {BBL_0, BBL_1}, {BBL_0, BBL_2} 两条路径执行后的污点传播情况；然后，分别使

用两条路径执行后的状态，进入 {BBL_3, BBL_4, BBL_5} 的循环结构进行迭代，分别生成迭代后的污点传播情况作为最终的污点传播摘要。

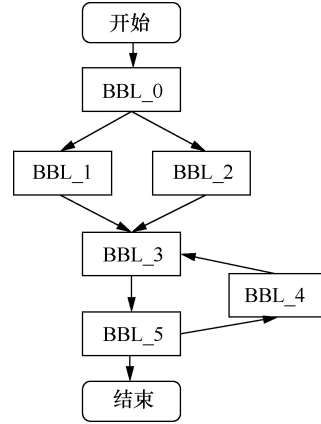


图 11 混合结构函数流程

Figure 11 Example of a hybrid structure flow chart

对于在函数内部同时存在多个循环结构的情况，通常来说，两个存在关联的循环结构有如图 12 中所示的 3 种结构关系，即内嵌、相交和相连。对于这 3 种循环结构，由于有限迭代算法的结果相当于一个基本块的输出，可以将这 3 种循环结构看作一个大的循环进行处理。在路径分析时，遇到这 3 种循环结构可以看作一个基本块进行分析。

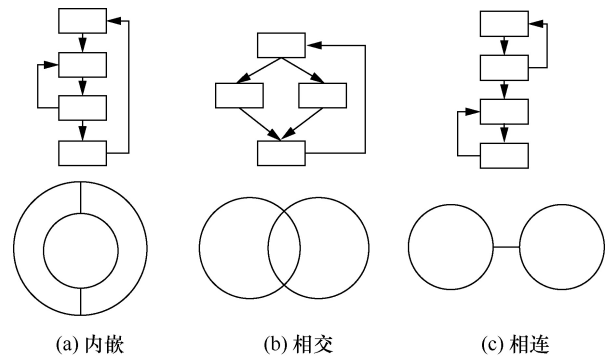


图 12 循环结构间关系

Figure 12 The relationship between circle structures

3 FSTaint 污点分析框架设计与实现

3.1 系统整体设计

本文提出的函数摘要，其性能优势体现在对指令插桩和分析次数的减少上，用函数污点传播关系代替部分指令污点传播关系，但是函数摘要

生成过程需要进行反汇编、函数结构分析和函数摘要生成 3 个主要步骤, 尤其函数摘要生成需要使用符号执行相关技术, 需要较多的时间开销。

函数摘要的效率优势体现在一次摘要分析, 多次摘要使用上。函数摘要的多次使用体现在 3 个方面: 首先, 程序在一次执行过程中, 如果对该函数有多次调用, 则函数摘要产生多次优化; 其次, 逆向分析一个程序时, 需要对其进行反复多次的数据流分析, 这样函数摘要会被多次使用; 最后, 对于有些函数, 可能会在其他程序中使用, 这样可以对摘要多次使用。

为了充分发挥函数摘要的效率优势, 并将污点分析在恶意代码行为提取中进行应用, 本文设计并实现了一款基于函数摘要的面向恶意代码行为分析的污点分析框架 FSTaint。在 FSTaint 中, 函数摘要分析模块和数据流记录模块为污点分析模块提供了数据流和高效传播规则的支撑; 对完整程序数据流的记录, 使分析框架可以在恶意代码一次执行的基础上, 进行任意次“回放”分析。这满足了恶意代码分析中, 需要分析大量 API 调用之间数据流关系的需求, 从而实现基于数据流的 API 关联, 实现对恶意代码的行为提取。FSTaint 整体框架如图 13 所示。

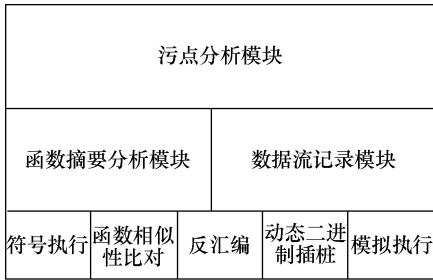


图 13 FSTaint 整体框架
Figure 13 FSTaint overall framework

3.2 函数摘要生成模块

在函数摘要生成前, 需要考虑选择合适的函数作为摘要生成的对象, 在常见的二进制程序中, 函数数目很多, 对全部函数进行污点摘要分析, 存在 3 方面的问题: ① 函数数量太多, 污点摘要生成效率低; ② 部分函数调用关系复杂, 导致污点传播关系很复杂, 难以处理; ③ 部分函数存在 I/O 交互的过程, 摘要分析无法确定输入输出情况。为了选择合适的函数作为摘要生成的对象,

FSTaint 从 3 个方面衡量函数生成污点摘要的优先级: ① 函数是否存在系统调用、I/O 操作、外部动态链接库调用等, 这些情况会导致函数执行过程不确定, 且通常会作为污点分析的 Source 点或 Sink 点, 不宜作为摘要生成对象; ② 函数被静态调用次数以及函数是否处于循环结构内, 如果一个函数被静态调用次数越多, 或者函数处于循环结构内, 那么这个函数在实际执行过程中被调用的次数可能会越多, 使用函数摘要的方式进行污点分析, 会带来更好的优化效果; ③ 函数结构复杂程度, 如果函数内部存在越多子函数调用以及控制流转移, 那么生成的污点摘要越复杂, 生成污点摘要的效率也会越低。综合考虑这 3 个方面的因素, FSTaint 选择优先级较高的函数作为摘要生成对象。

函数摘要生成过程中, FSTaint 使用符号分析框架 *angr*^[18] 将函数参数符号化表示, 按照路径敏感和有限迭代的算法, 逐个基本块进行符号执行分析, 最终在函数结束位置, 对比函数所有参数和返回值与函数入口时的符号状态, 收集数据映射关系, 作为函数污点传播摘要。

本文设计了函数摘要数据库, 用来存储已经分析过的函数摘要, 遇到新的需要生成摘要的函数时, 先进行二进制对比, 如果在摘要库中找到相同的函数, 则直接使用已有的函数摘要, 函数摘要库可以在一定程度缩短摘要分析时间。

在实验中发现, 当使用 *angr* 分析较为复杂的加密算法函数时, 输出数据的 AST 抽象语法树会比较复杂, 导致遍历 AST 收集数据映射关系时效率较低。为了解决这一问题, FSTaint 在 *angr* 的符号表达式构造过程中加入了描述污点映射关系的集合 *taint*, 构造新的 AST 节点时, 将子节点的 *taint* 集合合并得到新节点的 *taint* 集合。最后只需要取根节点的 *taint* 集合即可得到污点映射关系, 减少递归遍历 AST 产生的时间开销。

FSTaint 生成函数摘要的过程如图 14 所示。FSTaint 生成的函数摘要是一种通用的函数污点传递关系, 可以比较容易地移植到其他污点分析系统。

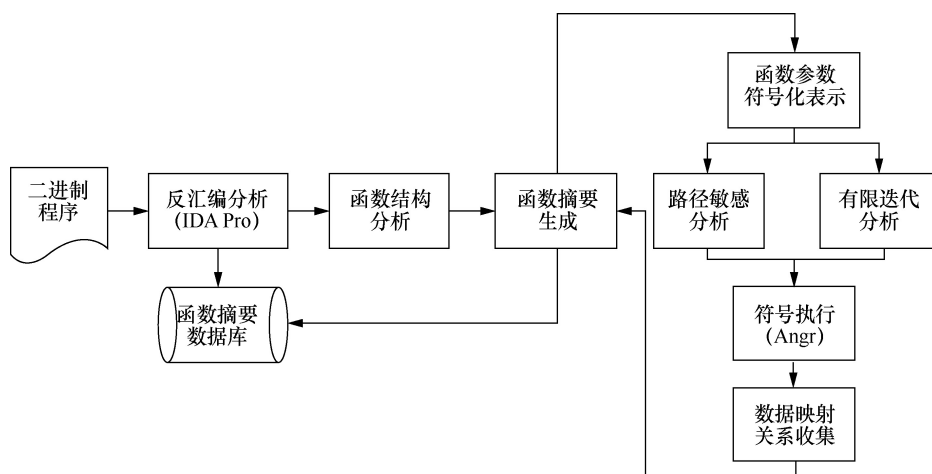


图 14 FSTaint 生成函数摘要的过程
Figure 14 Function summary generating module

在函数摘要生成模块具体实现过程中，函数结构识别、函数参数信息识别使用到了反汇编工具 IDA Pro；函数摘要生成过程中，使用了符号分析框架 angr，使用 Python 调用 angr 提供的符号分析接口，并对 angr 进行部分修改。

3.3 数据流记录模块

在恶意代码分析中，经常需要在二进制程序一次执行的基础上，分析多组 API 之间的关联以提取恶意代码行为，而现有的污点分析框架大多不支持同时标记多个污点标签。FSTaint 通过先记录程序执行轨迹，通过模拟执行的方法还原程序数据流，然后使用记录的数据流进行污点分析。这样的结构设计，使得程序可以在一次执行的基础上进行多次重复的污点分析，从而实现对不同类型的污点数据的跟踪。FSTaint 通过 Pin 插件记录程序执行轨迹，使用反汇编引擎 Capstone^[19]作为反汇编引擎分析指令数据流传递关系，使用 CPU 指令模拟器 Unicorn^[20]进行指令模拟执行和数据流还原。

如果直接记录所有数据流，则存在一个明显

问题，即在进行指令级数据流记录时，程序执行过程中由于执行的指令数量庞大，数据记录需要大量的时间和空间的开销。本文设计的函数污点传播摘要，可以优化数据流的存储，使用函数摘要可以减少数据指令级数据流的记录，用函数摘要生成的数据流传递关系代替。数据流记录模块结构如图 15 所示。

在该模块的实现过程中，在程序执行轨迹记录部分，使用了动态二进制插桩平台，并使用 C++ 语言开发了相应的插桩插件；数据流记录模块，使用了 C++ 语言，其中用到了反汇编引擎 Capstone 和 CPU 指令模拟执行引擎 Unicorn。

3.4 污点分析模块与恶意代码行为提取

数据流记录模块记录了程序执行过程中的完整数据流，后续的污点分析工作只需要在记录的数据流上进行分析，不需要考虑具体的指令污点传播规则的细节。在数据流记录的基础上，通用污点分析模块提供了污点分析必要的 API 接口，FSTaint 提供的通用污点分析 API 接口如表 4 所示。

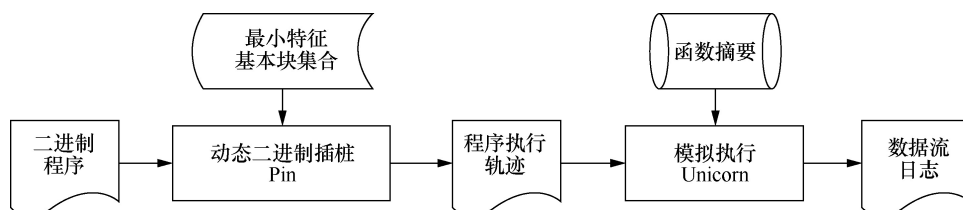


图 15 数据流记录模块结构
Figure 15 Data flow recording module

表 4 FSTaint 提供的通用污点分析 API 接口
Table 4 General taint analysis API interface provided by FSTaint

API 接口	说明
SetSourceIndex	通过 Index 设置 Source 点 (Index 为基本块执行过程中的序号)
SetSinkIndex	通过 Index 设置 Sink 点
SetTaintMemory	设置污点内存
SetSourceByBBLAddress	通过基本块地址设置 Source 点
SetSinkByBBLAddress	通过基本块地址设置 Sink 点
SetSourceIndexByAPIID	通过 API 调用序号设置 Source 点
SetSinkIndexByAPIID	通过 API 调用序号设置 Sink 点
DoAnalysis	开始污点分析
ShowTaintMemory	获取分析结束后的污点内存

该模块的实现, 主要涉及对函数摘要的使用和数据流日志的解析, 使用 Python 语言实现, 并提供了通用的污点分析接口, 接口可使用 Python 语言调用。

FSTaint 内置了任意 Windows API 调用数据关联分析的功能, 通过分别将每个 API 作为污点源, 将其输出数据作为污点数据, 跟踪污点数据是否作为输入参数进入其他 API 函数, 进行 API 间关联。常见的沙箱分析, 大多是对一系列 API 调用的罗列, 而通过污点分析将不同 API 的输入输出数据进行关联, 可以得到更为清晰的恶意代码行为描述。

在恶意代码行为分析中, 以图 16 的行为为例, 将 CreateFileA 函数生成的句柄作为污点数据, 可以关联出写文件和关闭句柄的操作; 将文件名作为污点数据, 关联出写文件后, 存在创建进程的操作。综合以上分析, 可以猜测该恶意样本可能存在释放文件并执行的恶意行为。

4 实验评估

污点分析的效率问题一直是污点分析技术的研究重点, 本文的函数摘要和 FSTaint 聚焦于提升污点分析, 实验部分主要是对 FSTaint 的分析效率进行测试。此外, 在实验分析部分, 对 FSTaint 的准确性进行了简要说明。

4.1 FSTaint 污点分析框架测试

为了对 FSTaint 的分析能力有比较深入的了解, 并发现 FSTaint 在真实污点分析中的优势和

不足, 本节针对 FSTaint 的两个核心问题, 即函数摘要生成与使用情况和 FSTaint 对污点分析的优化效率, 进行实验测试。在摘要生成与使用情况测试中, 主要测试函数摘要生成情况、使用函数摘要后指令级污点分析减少情况。之后, 本文进一步测试了使用函数摘要产生的污点分析优化效果, 并与经典的污点分析框架 libdft 进行了对比。

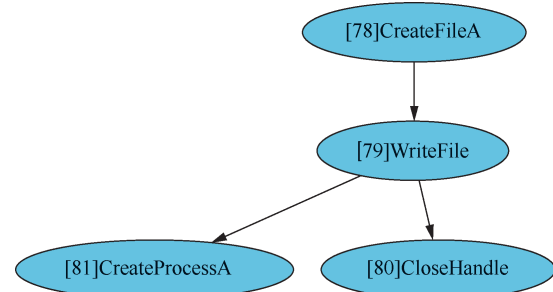


图 16 恶意代码行为分析示例
Figure 16 Example of Malicious code behavior analysis

实验用到的样本来自恶意样本手机网站, 从中选择 APT28、APT41、APTC01 样本集合以及其余 APT 组织 2020 年和 2021 年的样本, 从这些样本中筛选满足以下两个条件的样本: ① Windows 系统下的 32 位 exe 可执行程序; ② 样本可以在指定时间 (实验时使用 180 s) 内执行结束。首先, 当前的 FSTaint 分析只支持 Windows 系统下的 32 位可执行程序; 其次, 为了方便对比污点分析系统产生的性能损耗, 样本需要在指定时间内自行执行结束。实验分析样本如表 5 所示, 实验环境如图 17 所示。

本文选择 libdft 作为对比分析, 由于 libdft 论文提供的开源分析框架只支持 Linux 系统, 本文选用 Github 上针对 Windows 系统的开源项目 dftwin^[21], 其核心部分与原始 libdft 完全一致。

(1) 函数摘要生成情况

依据本文提出的函数摘要生成方法, 对 100 个实验样本进行摘要生成分析。分析过程中存在两种特殊情况: 一是函数内存在通过函数指针进行调用, 但指针内容未知的情况, 此类情况很难在摘要生成前通过反汇编分析得到, 因此只能尝试进行摘要生成, 在 angr 提示目标地址未知的时候, 停止对该函数的分析; 二是循环结构是否可以通过迭代而收敛的情况, 在分析前是未知的, 因此, 只能先尝试进行摘要生成

表 5 实验分析样本
Table 5 Samples used in experiment

序号	MD5	序号	MD5	序号	MD5
1	03bd34a9ba4890f37ac8fed78feac199	35	64ff0a8730472e36e62ce29a20f61529	69	ae004a5d4f1829594d830956c55d6ae4
2	048271f7f2f8d900485dd020cdea2dd9	36	6590830061f85c0acc5259013555d079	70	ae84767abec5a769b0c7cb5a356d74c5
3	081e2ce7e2a603a78cc6c20a05b08ca8	37	6983f7001de10f4d19fc2d794c3eb534	71	b2b51a85bdad70ff19534cd013c07f24
4	0b2733ffa73ecfc16224ce409da93b70	38	6fbccda7837bccd2ed9b3f69872d6ece	72	b80aa583591eaf758fd95ab4ea7afe39
5	0ba6bb2ad05d86207b5303657e3f6874	39	7525bc47e2828464ce07fa8a0db6844f	73	c1ab32afb0e2d7b7b1cad3fb831e9373
6	133a159e86ff48c59e79e67a3b740c1e	40	76f099ad94acf28d3eb1341804855824	74	c3dd32551450e15e7e81bb523dba7124
7	171ffa1fb15a298bcca8d8108fe913a9	41	773adf87ee49f9bf32851d33662dea79	75	c43edb579e43aaebf0c0703f84e43f7
8	1e217668d89b480ad42e230e8c2c4d97	42	78450806e56b1f224d00455efcd04ce3	76	ce26e91fc13ccb1be4b6bf6f55165410
9	22dfdcddd4f4da04b9ef7d10b27d84bc	43	7affcfb9857cc14dcc07fb8d226f03e0	77	ce5a7612892f27299362ae0569507e04
10	2366918da9a484735ec3a9808296aab8	44	7b6ca860c3e6bdc75b0be26db70a603a	78	cfb08ee3399604d37470797d49c01f72
11	23ae20329174d44ebc8dbfa9891c6260	45	7b973145f7e1b59330ca4dd1f86b3d55	79	d31e57fcb728a4f36e21764b164a9e57
12	23e27e5482e3f55bf828dab885569033	46	8674e3c77e8051cfd1e4d321a7188bf	80	d3e2c0943952285ce359fcbf28472491
13	2798b66475cf0794e9b868d656defca7	47	89081f2e14e9266de8c042629b764926	81	d4c93b85ffe88ddd552860b148831026
14	28c6f235946fd694d2634c7a2f24c1ba	48	8b6d824619e993f74973eedfaf18be78	82	d5e45a9db7f739979105e000d042f1fe
15	31c81459c10d3f001d2cce830239c16	49	8e3e4b006af3c1835ef3b7b4dcd3f1de	83	d61f883a59c3e0729a3ecf78b70044cf
16	36524c90ca1fac2102e7653dfadb31b2	50	8e744f7b07484afcf87c454c6292e944	84	d661dc2ad44bd056f7ca292727007693
17	36d99c1b915544c1bc9110b22e884acb	51	8f54cf08ee45a8d5eb31d05dbab4b561	85	dc5695024cd23c90883db145cb236490
18	39c361abb74f9a338ea42a083e6c7df8	52	90508ff4d2fc7b9c68636c716d84e6b4	86	de82407423aad8009e378e406515e92
19	3a4e1f3f7e1baab8b02f3a8ee20f98c9	53	9070f7100fa2f41c2c0757b34e0a401c	87	e598898f04596623be5bf9d168611d4
20	3a8d74bc002e1acae49a64f3f77aac0b	54	914151fa49be06ab570bb0db77ce6960	88	e5db592704f30d42537b1257e79ff223
21	40528e368d323db0ac5c3f5e1efe4889	55	92274d90c221b0aad382f816026a4781	89	e6348ee5beb9c581eeef4e076c5d631
22	40e698f961eb796728a57ddf81f52b9a	56	9227678b90869c5a67a05defcaf21dfb	90	ea726d3e8f6516807366584f3c5b5e2a
23	4489d5077c5d2396e3a94d652adae1ca	57	947892152b8419a2dfe498be5063c1da	91	eac0e57a22936d4c777aa121f799fee6
24	4585b2f7e7909bcb12528a0b383133d2	58	9617f3948b1886ebc95689c02d2cf264	92	eb0fd733a3f3ec67cce7f09f1c5ff6428
25	460cebaa45f98070a3b5abf7f578645a	59	961cef5861317dd9966dc3f6eb5387d8	93	ef1b7fd90b274d872ee15a3f2ca35193
26	48971e0e71300c99bb585d328b08bc88	60	97aaf130fca251e5207ea74b2558293d	94	ff01a9a2d1e31332ed36c1a4d2839f412
27	49b1ca0752d166c2cc5e04cbab8b71ee	61	97e49353a25c3f1d81a6139735697940	95	f172ad4e906d97ed8f071896fc6789dc
28	56450799fe4e44d7c5aff84d173760e8	62	98a073e1e545075aa0030995cc07745a	96	f267e9b788eb1ffa509d0d88e0c0cecc
29	595e43cdf0edcaa31525d7aad87b7be4	63	991ffdbf860756a4589164de26dd7ccf	97	f2b63405dba8c2c1867234f1172bb34d
30	5a4d42e7cf6b6f188821a854f1829df3	64	9a8f39ebcc580aa56d6ddaf5804eae61	98	faea496c34cdc5a401badcdebbd4a87e
31	603935efa89d93ea39b4b4d4a52ec529	65	9d1a09bb98bf1ee31f390b60b0cf724d	99	fc9e7dc13ce7edc590ef7dfce12fe017
32	60ddb540da1aeffe1e14f12578eafda8	66	9d3c177cbd83d31e61fb12f6a40c295f	100	fed912fd7ed80e2cdf905873c6ced4ad
33	61e3571b8d9b2e9ccfadc3dde10fb6e1	67	9e05a9c264c8a908a8e79450fcbff047		
34	626270d5bf16eb2c4dda2d9f6e0c4ef9	68	9f022df0bcfded9377baf4da1fbc7b8c		

分析，设置迭代次数上限（实验中采用 100 次迭代），当到达迭代次数上限依然无法生成摘要时，则生成摘要分析结束。

实验测试的 100 个样本的函数摘要生成情况

如图 18 所示。统计发现，对 21.4% 的函数尝试进行污点摘要生成（含函数指针未知和迭代无法收敛的情况），在这些函数中，有 45.5% 的函数可以生成摘要，占总函数数量的 9.72%。

	FSTaint	Libdft
操作系统环境	Windows 7 64 位虚拟机	
虚拟机系统	VMWare Workstation 16.1.2	
虚拟机硬件配置	2 个双核心处理器, 8 GB 内存, 60 GB 硬盘	
插桩程序	Pin2.12	Pin2.14
其他	Unicorn1.0.1 Capstone4.0.2 Angr9.0.5171 Python3.8	

图 17 实验环境

Figure 17 Environment of experiment

(2) FSTaint 对指令数据流分析的减少情况

基于函数摘要的污点分析优化方法的核心目的是使用函数污点传播关系代替函数内密集的指令级污点传播关系, 从而提升污点分析效率。为了对使用函数摘要后所需指令污点传播分析数量的减少情况进行测试, 本文设计了如下的对比实验。

- ① 使用 Pin 对实验样本进行基本块级插桩, 记录程序执行的指令总数。
- ② 使用 FSTaint 对程序进行污点分析, 统计记录的数据流日志中指令数据流传播关系的数量。
- ③ 对比使用函数摘要前后, 指令数据流传播数量的减少情况。

通过对 100 个样本进行测试, 统计发现, 使用函数摘要 (含用户函数摘要和 API 函数摘要) 优化后, 指令数据流分析相对减少 99% 以上的样本占 33%, 减少的指令数据流分析均超过 85%。总体而言, 函数摘要对指令级数据流分析平均减少 97.23%, 效果十分显著, 如图 19 所示。

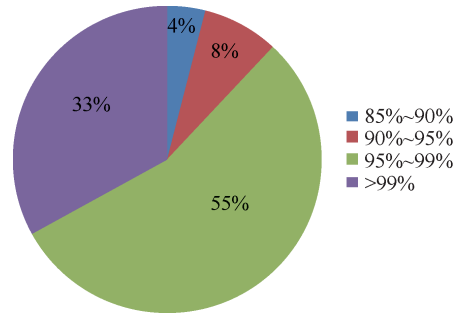


图 19 指令级污点分析减少情况

Figure 19 Instruction-level taint analysis reduction

(3) FSTaint 分析效率测试与对比

为了对 FSTaint 的效率进行分析评估, 本文选择真实 APT 样本作为分析对象, 并将 FSTaint 的污点分析综合效率与 libdft 进行了对比。

FSTaint 污点分析过程包括 4 个部分, 分别为函数摘要生成、程序执行轨迹记录、数据流还原和污点分析, 其中摘要生成过程可以与程序执行轨迹记录过程并行进行。在 FSTaint 中, 进行一次污点分析, 有 3 个执行步骤: 摘要生成与程序执行轨迹记录 (T1)、数据流还原与记录 (T2) 和污点分析 (T3), 各模块耗时占比如图 20 所示。从第二次污点分析开始, 则每次只需在内存解析好的数据基础上进行分析, 耗时很少, FSTaint 分析次数与平均耗时的关系如图 21 所示。

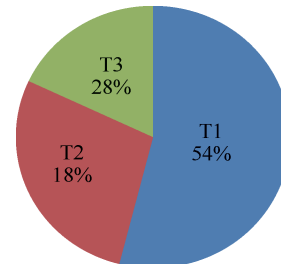


图 20 FSTaint 各模块耗时占比

Figure 20 Time-consuming ratio of each module

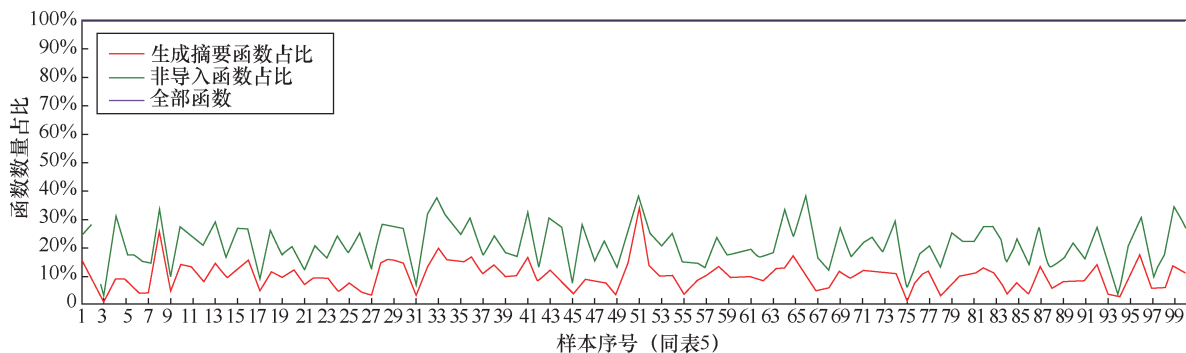


图 18 函数摘要生成情况

Figure 18 Function summary generation

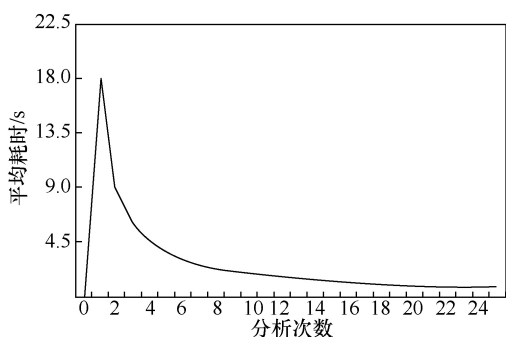


图 21 FSTaint 分析次数与平均耗时的关系

Figure 21 The relationship between the number of FSTaint analyses and the average time spent

图 22 为 FSTaint 和 libdft 平均单次污点分析耗时与程序自身执行时长的对比, 其中 FSTaint 针对每个样本的污点分析次数为样本执行过程中调用的存在输出的 API 数量。在分析的 100 个样本中, FSTaint

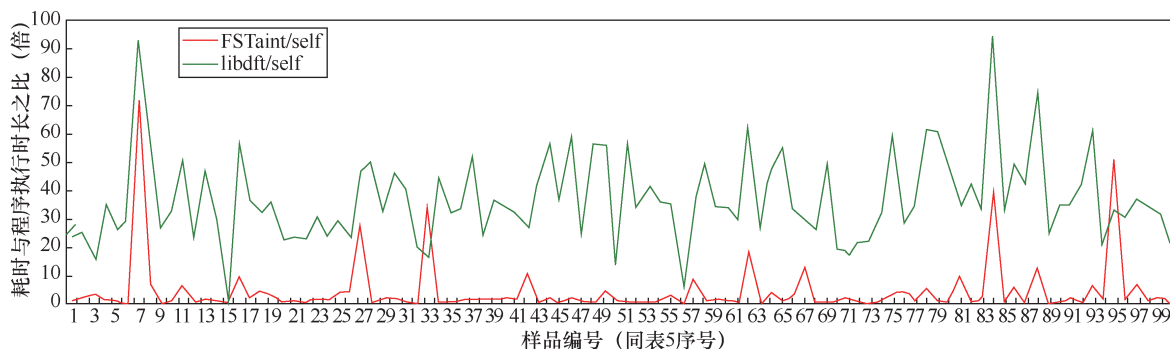


图 22 分析耗时与程序执行时长的对比

Figure 22 Comparison of the ratio of analysis time to program execution time

虽然函数摘要在一定程度上缓解了数据流日志太大, 难以记录和分析的问题, 但是执行过程很长或者计算密集型的程序这个问题依然存在, 并且对污点分析效率有比较严重的影响。对恶意代码来说, 其核心功能大多是通过系统 API 实现的, FSTaint 不需要跟踪进入系统 API 内部, 只需要记录其输入输出参数即可, 需要记录的数据流相对较少; 对于计算密集型的程序, 需要记录的指令数据流太多, 数据流日志会过大。当数据流日志过大时, 读入和解析会耗时较多, 且不利于检索, 这是 FSTaint 后续需要重点优化提升的部分。

相对于 libdft, FSTaint 的另一个优势是使用了 API 函数调用信息, 从而避免了进入 API 函数内部, 而 libdft 会一直进入 API 内部进行分析, 直到遇到系统调用。恶意代码通常有比较密集的

平均单次分析耗时为程序执行时长的 4.94 倍, 而 libdft 则为 37.53, FSTaint 仅为 libdft 分析时长的 12.9%, 即 FSTaint 的分析效率为 libdft 的 7.75 倍。

4.2 FSTaint 的综合分析

通过框架结构设计与实验分析可以发现, FSTaint 存在以下几个方面的优势: ① 可回放的污点分析框架设计可以在一定程度上提升污点分析的便捷性和实用性, 针对恶意样本这样需要反复多次进行数据流跟踪的分析情况十分便利; ② FSTaint 在对同样本进行多次污点分析时, 相对于 libdft 等常见的污点分析框架具有较好的性能优势, 可以很大程度上提高分析效率; ③ 函数摘要在一定程度上提升了分析效率, 减小了数据流日志的大小, 提升了进行程序执行数据流记录的可行性。

API 调用, FSTaint 的优势则更加明显。

关于 FSTaint 的准确性, 在指令数据流传播处理方面, 相较于 libdft, FSTaint 增加了对 SSE 扩展指令、浮点指令的处理; 在系统 API 处理方面, libdft 需要自定义系统调用的数据传播方式, FSTaint 目前支持对 12 000 余个 Windows 系统 API 的调用情况进行记录。此外, FSTaint 在采用路径不敏感的分析方法处理循环结构时可能会出现过污染的情况, 影响污点分析的准确性。

此外, 在实际污点分析中, 本文发现了 libdft 在常见 X86 指令污点传播规则方面存在的一些不足, 以下面一条指令为例。

```
mov al, ds:byte_418AA0[eax]
```

libdft 中的污点传播规则是将 byte_418AA0[eax]这个位置的污点状态复制到寄存器

al 中。而实际中, 如果这个地址存储的内容不是污点数据, 而偏移量 `eax` 为污点, 此时污点传播就会出错, 而这种污点传播现象在 AES 等加密算法的混淆过程中十分常见, 不恰当的处理会在一定程度上降低分析的准确性。对于此类指令污点传播规则, 相较于 `libdft`, `FSTaint` 更加完善, 从而提高污点分析的准确性。

5 结束语

本文针对污点分析中指令级插桩和数据流传播分析带来的效率较低的问题, 提出了基于函数摘要的优化方法。本文给出了函数污点传播摘要定义, 并针对非循环结构路径提出了路径敏感的摘要生成方法, 针对循环结构基于格上的不动点原理, 提出了有限迭代的函数摘要生成方法。在函数摘要研究的基础上, 本文设计实现了面向恶意代码分析的污点分析框架 `FSTaint`, 并使用真实 APT 攻击样本进行了测试评估。通过实验对比, 在恶意代码分析过程中, `FSTaint` 的污点分析效率是 `libdft` 的 7.75 倍。

`FSTaint` 存在两个方面的不足, 一方面, 在有限迭代的摘要生成过程中, 采用路径不敏感的分析方法, 使得分析结果可能存在一定的“过污染”情况, 同时有限迭代的生成方法不支持输入数据“无穷”的情况; 另一方面, 对于执行过程较长的程序或者计算密集型程序, `FSTaint` 可能产生较多数据流日志, 使得日志读取和分析过程时间较长, 影响分析效率。

参考文献:

- [1] SCHWARTZ E J, AVGERINOS T, BRUMLEY D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)[C]//Proceedings of 31st IEEE Symposium on Security and Privacy. 2010: 16-19.
- [2] 郭方方, 王欣悦, 王慧强, 等. 基于最大频繁子图挖掘的动态污点分析方法[J]. 计算机研究与发展, 2020, 57(3): 631-638.
- [3] GUO F F, WANG X Y, WANG H Q, et al. A dynamic stain analysis method on maximal frequent sub graph mining[J]. Journal of Computer Research and Development, 2020, 57(3): 631-638.
- [4] KORCZYNSKI D, YIN H. Capturing malware propagations with code injections and code-reuse attacks[C]//Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17). 2017: 691-1708.
- [5] CABALLERO J, GRIECO G, MARRON M, et al. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities[C]//Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012. 2012: 133-143.
- [6] CHEN P, CHEN H. Angora: efficient fuzzing by principled search[C]//Proceedings of 2018 IEEE Symposium on Security and Privacy. 2018: 711-725.
- [7] DUMITRU C. Detecting software vulnerabilities static taint analysis[D]. Bucharest: University Politehnica of Bucharest, 2009.
- [8] 高凤娟, 王豫, 陈天骄, 等. 基于污点分析的数组越界缺陷的静态检测方法[J]. 软件学报, 2020, 31(10): 2983-3003.
- [9] GAO F J, WANG Y, CHEN T J, et al. Static checking of array index out of bounds defects in C programs based on taint analysis[J]. Journal of Software, 2020, 31(10): 2983-3003.
- [10] JANG D, JHALA R, LERNER S, et al. An empirical study of privacy-violating information flows in JavaScript web applications[C]//Proceedings of the 17th ACM Conference on Computer and Communications Security - CCS '10. 2010: 270-283.
- [11] WANG X J, MA R, DOU B W, et al. OFFDTAN: a new approach of offline dynamic taint analysis for binaries[J]. Security and Communication Networks, 2018, 2018: 7693861.
- [12] LUK C K, COHN R, MUTH R, et al. Pin: building customized program analysis tools with dynamic instrumentation[C]//Proceedings of 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05). 2005: 190-200.
- [13] BRUENING D, ZHAO Q, AMARASINGHE S. Transparent dynamic instrumentation[C]//Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments - VEE '12. New York: ACM Press, 2012: 133-144.
- [14] CLAUSE J, LI W C, ORSO A. Dytan: a generic dynamic taint analysis framework[C]//Proceedings of the 2007 International Symposium on Software Testing and Analysis - ISSTA '07. 2007: 196-206.
- [15] KEMERLIS V P, PORTOKALIDIS G, JEE K, et al. Libdft[J]. ACM SIGPLAN Notices, 2012, 47(7): 121-132.
- [16] GALEA J, KROENING D. The taint rabbit: optimizing generic taint analysis with dynamic fast path generation[EB].
- [17] JIANG M, WU D H, XIAO G Y, et al. TaintPipe: pipelined symbolic taint analysis[C]//Proceedings of 24th USENIX Conference on Security Symposium (SEC'15). 2015: 65-80.
- [18] CHABBI M. Efficient dynamic taint analysis using multicore machines[D]. Arizona: The University of Arizona, 2015.
- [19] 代伟, 刘智, 刘益和. 基于二进制代码的动态污点分析[J]. 计算机应用研究, 2014, 31(8): 2497-2501, 2505.
- [20] DAI W, LIU Z, LIU Y H. Binary code-based dynamic taint analysis[J]. Application Research of Computers, 2014, 31(8): 2497-2501, 2505.
- [21] YAN S, WANG R, SALLS C, et al. SOK: (State of) The Art of War:

Offensive Techniques in Binary Analysis[C]//Proceedings of IEEE Security & Privacy. 2016:138-157.

[19] NGUYEN A Q, DANG H V. Unicorn: next generation CPU emulator framework[C]//Proceedings of BlackHat USA. 2015.

[20] Capstone[EB].

[21] dftwin[EB].

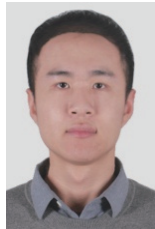


舒辉（1974—），男，江苏盐城人，数字工程与先进计算国家重点实验室教授、博士生导师，主要研究方向为网络安全。

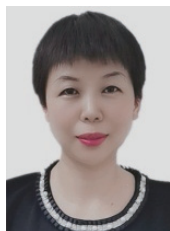
[作者简介]



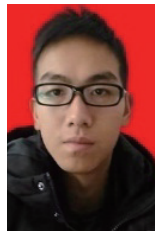
杨盼（1993—），男，陕西宝鸡人，数字工程与先进计算国家重点实验室硕士生，主要研究方向为软件逆向分析。



黄宇垚（1997—），男，福建福州人，数字工程与先进计算国家重点实验室硕士生，主要研究方向为网络协议安全。



康绯（1972—），女，河南周口人，数字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信息安全。



吕小少（1989—），男，湖北荆门人，数字工程与先进计算国家重点实验室硕士生，主要研究方向为软件逆向分析。