# 二进制代码切片技术在恶意代码检测中的应用研究

梅 瑞[1,2], 严寒冰[3*], 沈 元[4], 韩志辉[3]

[1]中国科学院信息工程研究所 北京 中国 100093
[2]中国科学院大学网络空间安全学院 北京 中国 100049
[3]国家计算机网络应急技术处理协调中心 北京 中国 100029
[4]北京航空航天大学计算机学院 北京 中国 100191

**摘要** 恶意代码检测技术作为网络空间安全的重要研究问题之一, 无论是传统的基于规则的恶意代码检测方法, 还是基于机器学习的启发式恶意代码检测方法, 首先都需要自动化或人工方式提取恶意代码的结构、功能和行为特征。随着网络攻防的博弈, 恶意代码呈现出隐形化、多态化、多歧化特点, 如何正确而有效的理解恶意代码并提取其中的关键恶意特征是恶意代码检测技术的主要目标。程序切片作为一种重要的程序理解方法, 通过运用“分解”的思想对程序代码进行分析, 进而提取分析人员感兴趣的代码片段。由于经典程序切片技术主要面向高级语言, 而恶意代码通常不提供源代码, 仅能够获取反汇编后的二进制代码, 因此二进制代码切片技术在恶意代码检测技术中的应用面临如下挑战: (1)传统的面向高级语言的程序切片算法如何准确而有效的应用到二进制代码切片中; (2)针对恶意代码如何尽可能完整的提取能够表征关键恶意特征的程序切片。本文通过对经典程序切片算法的改进, 有效改善了二进制代码过程间切片和切片粒度问题, 并通过人工分析典型恶意代码, 提取了 42 条有效表征恶意代码关键恶意特征的切片准则。实验表明, 本文提出的方法可以提升恶意代码同源性检测的精度和效率。

**关键词** 程序切片; 二进制分析; 恶意代码检测
**中图法分类号** TP309.5 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2021.05.08

# Application Research of Slicing Technology of Binary Executables in Malware Detection

MEI Rui[1,2], YAN Han-Bing[3*], SHEN Yuan[4], HAN Zhi-Hui[3]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China
[3]National Computer Network Emergency Response Technical Team/Coordination Center of China (CNCERT/CC), Beijing 100029, China
[4]School of Computer Science and Engineering, Beihang University, Beijing 100191, China

**Abstract** Malware detection technology has been an important research topics of cybersecurity. Both traditional rule-based malware detection methods or the heuristic malware detection methods based on machine learning are all need to extract structural, functional, and behavioral characteristics of malware automatically or manually. With the game of cyber attack and defense, malware presents the characteristics of stealthy, polymorphic and multipartite. How to understand the malware accurately and effectively and extract the key malicious features is the main goal of malware detection technology. As a kind of important program understanding method, program slicing analyzes the program code by using the idea of “decomposition”, and then extracts the code snippets that the analyst is interested in. Because the classic program slicing technology is mainly for high-level program languages, and malware usually does not provide source code, but only the binary code can be obtained. Therefore, the application of binary code slicing technology in malware detection technology faces the following challenges: (1) how the classical high-level language-oriented program slicing algorithm can be applied to binary code slices accurately and effectively; (2) how to extract the program slices that can represent the key malicious features as completely as possible for malware. Through the improvement of the classical program slicing algorithm, this paper effectively improves interprocedural slicing and slicing granularity issues. By analyzing the typical malware manually, we extract 42 slicing criterions that effectively characterize the malicious features of malware. Experiments show that the proposed method can improve the accuracy and efficiency of malware homology detection.

**Key words** program slicing; binary analysis; malware detection

**通讯作者:** 严寒冰, 博士, 教授级高级工程师, 博士生导师, Email: yhb@cert.org.cn。
本课题得到国家自然科学基金重点项目(No. U1736218)和科技部重大专项(No. 2018YFB0804704)资助。
收稿日期: 2019-07-12; 修改日期: 2019-08-23; 定稿日期: 2021-03-05

## 1 引言

随着计算机软件的规模越来越大、复杂度越来越高, 软件内部实体数量和交互次数也呈指数级增长, 这种软件复杂性(Software Complexity)一方面增加了无意中干扰交互的风险, 从而增加了在代码变更时引入缺陷的可能性, 在极端的情况下甚至使修改软件变得几乎不可能。另一方面, 这意味着软件更加难以理解, 使得对于软件代码的结构、功能、行为和逻辑设计的认识更加困难[1]。程序切片(Program Slicing)作为一种重要的程序理解(Program Understanding)技术, 它的概念由 M. Weiser 于 1979 年首次提出, 并在其后来的论文中对程序切片理论进行了形式化的论述。M. Weiser 提出了一种基于控制流图(Control Flow Graph, CFG) 结合数据流方程的计算程序切片的算法, 以及基于程序切片的软件度量(Software Measurement) 的基本框架, 为程序切片技术的发展和应用提供了基本的理论支持[2-3]。随着的一些研究扩展了 M. Weiser 的程序切片理论和方法, 提出了静态切片和动态切片、前向切片和后向切片、削片、砍片等不同的方法, 应用于不同的与领域中[4]。

当前网络空间安全的主要威胁之一是恶意代码的传播, Malwarebytes 发布的最新恶意代码现状报告中显示[5], 2018 年检测到的后门、间谍软件、木马等类型的恶意代码同比增加 20%以上, 恶意代码通过系统漏洞或垃圾邮件等大规模传播, 这对于恶意代码检测提出了挑战。检测和分析恶意代码的首要工作是提取恶意代码中的恶意特征, 而程序切片正是一种通过对程序进行分解并提取特定代码片段(Code Snippet)的重要方法。然而, 由于恶意代码通常以二进制代码形式传播, 因此许多针对恶意代码检测技术的研究都是基于二进制代码开展的, 尽管有一些针对二进制代码切片的研究, 但是很少有将二进制代码切片应用于恶意代码检测的方法。

本文提出的二进制代码切片在恶意代码检测中的应用方法有 2 点贡献:

1)针对经典程序切片算法对二进制代码执行程序切片的问题, 我们提出了改进的面向二进制代码的过程间切片方法, 通过删除二进制代码过程调用时参数传递依赖关系的冗余节点, 降低了经典程序切片算法对二进制代码生成系统依赖图(System Dependence Graph, SDG)的复杂度。此外, 我们针对二进制汇编代码的多赋值(Multi-assignments)属性, 分析了二进制代码切片的粒度问题, 通过将二进制代码转换为中间表示(Intermediate Representation, IR), 并提出一种带有中间表示的系统依赖图, 有效改善二进制代码切片的粒度问题。

2)针对二进制代码切片如何有效提取恶意代码关键恶意特征的问题, 我们通过人工分析典型恶意代码, 根据恶意代码的基础行为和反分析行为, 提出了 10 类共 42 条有效表征恶意特征的切片准则(Slicing Criterion), 并在实验中得到了验证。

本文余下部分按如下内容组织: 第 2 章作为研究背景介绍了程序切片的主要概念和关键算法; 第 3 章针对经典程序切片应用于二进制代码切片引发的问题, 提出了二进制代码切片的改进方法; 第 4 章基于恶意代码检测技术, 提出基于二进制代码切片的恶意代码同源性检测模型, 并阐述了模型中所选取的切片准则; 第 5 章对本文提出方法进行实验验证和评估; 第 6 章是本文的总结和进一步研究展望。

## 2 程序切片研究背景

程序切片是针对程序代码(包括高级语言源代码或者编译后的二进制代码)进行分析, 并依据分析人员的特定需求提取部分语句(指令)序列的过程[6-7]。程序切片基于指令的控制流和数据流, 分析指令的执行序列和数据的产生、复制、传递和消失的过程, 最终生成所需的代码片段[8]。

### 2.1 控制依赖分析

计算机程序的执行包括 2 类控制结果, 即顺序结构和非顺序结构, 非顺序结构通过条件控制语句或挑战指令将代码分为若干区域, 称为基本块(Basic Block)[9-10], 一个基本块是满足下列条件的一组连续指令代码:

1)**原子性**: 基本块是程序控制流的最小执行单元, 基本块中的指令仅存在全部执行和全部不执行两种状态。

2)**完整性**: 程序执行时只能从该基本块的第一条指令进人该基本块, 并且只能从该基本块的最后一条指令结束。

3)**序列性**: 基本块内的指令执行顺序仅遵循指令序列在存储器中的排列顺序。

算法 1 描述了代码基本块识别的基本方法[11], 其中, 输入参数 *I* 为待识别代码基本块的指令序列; *LeaderSet* 是基本块首条指令的集合, 初始为空集; *EntryInsType* 是基本块入口指令类型枚举, 包括程序入口指令、函数入口指令、跳转指令的目的地址指令。算法的输出结果 *BlockSet* 是以代码块入口指令为索引的代码块指令序列集合。

**算法 1.** 代码基本块识别算法

**输入**: $I=\{ins_i, i\in\{1, 2, 3, \cdots, n\}\}$
$LeaderSet\rightarrow \phi$
$EntryInsType$
**输出**: $BlockSet\rightarrow \phi$
**过程**:
1: FOR $ins_i$ IN $I$
2: IF $ins_i$ IS $EntryInsType$
3: $LeaderSet=LeaderSet\cup\{ins_i\}$
4: FOR $x$ IN $LeaderSet$
5: $BlockSet[x]=x$
6: $i=x+1$
7: WHILE $i\leqslant n$ AND (NOT $i\in LeaderSet$)
8: $BlockSet[x]=BlockSet[x]\cup\{i\}$
9: $i=i+1$

控制流图: 将基本块视为一个基本单元节点, 基本块之间在程序执行流程上互为前趋和后继关系视为两个基本块之间存在一条边, 则整个程序能够转换为一个有向图[12]。算法 2 描述了控制流图的构建算法。

**算法 2.** 控制流图构建算法

**输入**: $BlockSet$
$BranchInsSet$
$BranchMap$
**输出**: $CFG$
**过程**:
1: FOR $b$ IN $BlockSet$
2: $x=b[\text{len}(b)-1]$
3: IF $x\in BranchInsSet$
4: FOR $b\_target$ IN $BranchMap[x]$
5: $CreateEdge(CFG,b,b\_target)$
5: ELSE
6: $CreateEdge(CFG,b,\text{Next}(b))$

控制流图是一个有向图, 可以表示为一个四元组, $G=\{V, E, Entry, Exit\}$, 基于图论的观点, 对控制流图的关键属性做如下定义:

**定义 1. 可执行路径**: 在控制流图中, 从任意节点开始进行节点遍历会形成一条路径, 路径上的基本块串联后可以形成程序的一条执行路径, 该路径称作控制流图上的一个可执行路径。

**定义 2. 前必经节点**: 对于控制流图中的两个节点 $a$、$b$, 如果从开始节点 $Entry$ 到节点 $b$ 的所有路径都经过节点 $a$, 则称节点 $a$ 支配节点 $b$, 并称节点 $a$ 是节点 $b$ 的前必经节点, 记为 $a\rightarrow b$。

**定义 3. 严格前必经节点**: 如果节点 $a$ 是节点 $b$ 的前必经节点, 并且 $a\neq b$, 则称 $a$ 是 $b$ 的严格前必经节点, 记为 $a\rightarrow_p b$。

后必经节点和严格后必经节点也有类似的定义, 上述定义中的节点均为代码基本块。

**控制依赖分析**以控制流图为分析对象, 对基本块之前的控制依赖关系进行分析, 上文所述的前必经和后必经关系并不等价于控制依赖关系, 还需要依赖节点能够决定被依赖节点是否执行。

**定义4. 控制依赖关系:** 令 $G$ 为程序 $P$ 的控制流图, 其中 $a$ 和 $b$ 是 $G$ 中的两个节点, 当 $a$ 和 $b$ 满足下列两个条件时, 则称 $b$ 控制依赖于 $a$, 记作 $b\rightarrow_{cd} a$: (1) 从 $a$ 到 $b$ 有一条可执行路径, 并且对于该路径上除 $a$, $b$ 外的任意节点 $n$, 节点 $b$ 都是其后必经节点; (2) 节点 $b$ 不是 $a$ 的后必经节点。

由控制依赖关系可以定义控制依赖图 $G=(V, C)$, 其中 $V$ 是程序中所有语句(或基本块)对应节点的集合, $C$ 是控制依赖图的边集合。如果有节点 $u$ 直接控制依赖于 $v$, 即 $u\rightarrow_{cd} v$, 则将 $u$ 到 $v$ 的边添加到 $C$ 中。于是利用控制依赖图就可以完整的描述程序中每个基本块的控制依赖关系。如果将每条语句作为图的一个节点, 也可以得到类似的控制依赖图。

## 2.2 数据依赖分析

数据流分析关注的是变量(在汇编语言中还包括内存地址和寄存器的值)在程序执行过程中的变化, 即跨越多条语句的同一变量定义、赋值和运算操作[13]。数据依赖分析包括可到达定义分析(Reaching Definition Analysis)和活性分析(Liveness Analysis)。

**可到达定义分析**的目标在于获取操作指令或语句所引用变量的来源, 即引用的变量如何产生、复制和传递。为了准确地刻画程序状态, 我们将程序执行路径表示为程序中连续的语句执行前后的状态序列, 若程序包含 $n$ 条语句, 则存在 $n+1$ 个执行状态, 记为 $P_0, P_1, P_2, P_3, \cdots, P_n$, 其中对于任意语句 $s\in\{0, 1, 2, \cdots, n\}$, 则 $s$ 语句执行前的状态为 $P_{s-1}$, 执行后的状态为 $P_s$。

**定义5. 可到达定义**: 若语句 $s$ 定义了变量 $x$, 则该语句的定义到达状态位置 $P$, 当且仅当在程序控制流图中存在从语句 $s$ 对应的状态位置 $P_s$ 到 $P$ 的一条路径, 并且该路径上没有变量 $x$ 的其他定义, 同时称语句 $s$ 是代码位置 $P$ 的一个可到达定义。

可到达定义相关概念的非形式化描述如下:

**1)定义集 *Def(x)***: 定义变量 $x$ 的所有语句的集合。

**2)引用集 *Use*(*x*)**: 任何使用变量 *x* 的语句的集合。

**3)产生集 *Gen*(*s*)**: 所有由语句 *s* 给出的变量定义所在的语句构成的集合。

**4)消灭集 *Kill*(*s*)**: 若语句 *s* 重新定义变量 *x*, 而 *x* 此前由语句 *s*′定义, 则称 *s* 消灭定义 *s*′。所有由 *s* 消灭的定义的集合称为 *s* 的消灭集。

**5)入集 *In*(*s*)**: 所有在语句 *s* 之前仍然有效(没有被消灭)的定义语句的集合。

**6)出集 *Out*(*s*)**: 所有离开语句 *s* 时的定义语句的集合, 添加 *s* 产生(Gen)的语句, 同时去掉语句 *s* 所消灭(Kill)的定义语句。

上述可到达定义的相关概念, 若将其中的分析对象由语句变换为代码基本块, 也存在相似的定义。

**算法 3. 可到达定义算法**

**输入**: *CFG*={*V,E,Entry,Exit*}
**输出**: *Out*
**过程**:

```
1: FOR b IN V
2:     In(b)=ϕ
3:     Out(b)=Gen(b)
4: IsConvergence=false
5: WHILE IsConvergence==false
5:     FOR b IN V
6:         In(b)=∪{Out(p) | p∈Pred(b)}
7:         OldOut=Out(b)
8:         OldIn=In(b)
9:         Out(b)=Gen(b)∪(In(b)-Kill(b))
10:                IF Out(b)==OldOut AND In(b)==OldIn
11:            IsConvergence=true
12:        ELSE
13:            IsConvergence=false
```

算法 3 描述了可到达定义的计算方法, 其中输入为程序的控制流图, 输出为指定语句或基本块的出集 *Out*(*s*)。在算法中, 使用变量 *IsConvergence* 作为是否继续迭代的开关, 即经过有限轮迭代后出集 *Out* 最终达到收敛。在最坏的情况下, 出集 *Out* 可能包括所有的语句或基本块。

另一方面, **活性分析**是对语句中定义的变量是否在后续语句中被引用以及被哪些语句引用的分析。对于任意语句 *s* 和 *s* 中的一个变量 *x*, 如果 *x* 在 *s* 上的值在控制流图中从 *s* 出发的某条路径上使用, 就称 *x* 在 *s* 上活跃, 否则称 *x* 在 *s* 上消亡。此外, 活性分析还关注变量保持活跃的范围称为活性范围, 即分析任意语句 *s*, 变量 *x* 在后续语句中继续保持活跃的语句范围。活性分析常用于编译器的代码生成阶段对基本块分配寄存器, 即依据变量的活性进行寄存器分配的优化。

数据依赖分析描述引用变量的语句或基本块对定义该变量的语句或基本块的依赖, 是一种“定义—引用”的依赖关系。

**定义 6. 数据依赖关系**: 设 *a* 和 *b* 分别为程序 *P* 的控制流图 *G* 中的两个节点, *x* 为 *P* 中的一个变量, 若 *a* 和 *b* 满足下列条件, 则称 *b* 关于变量 *x* 数据依赖于 *a*, 记为 $b\rightarrow_{dd} a$:

1)*a* 对变量 *x* 进行定义, 即 $a\in Def(x)$;

2)*b* 中引用了变量 *x*, 即 $b\in Use(x)$;

3)*a* 到 *b* 有一条可执行路径, 且在此路径上不存在语句对 *x* 进行定义。

显然, 如果 $b\rightarrow_{dd} a$, 则节点 *b* 是节点 *a* 的一个可到达定义。

由数据依赖关系可以定义数据依赖图 *G*=(*V*, *D*), 其中 *V* 是程序中所有基本块对应节点的集合, *D* 是数据依赖图的边集合。如果有节点 *b* 数据依赖于 *a*, 即 $b\rightarrow_{dd} a$, 则将 *b* 到 *a* 的边添加到 *D* 中。于是利用数据依赖图就可以完整的描述程序中每个基本块的数据依赖关系。如果将每条语句作为图的一个节点, 也可以得到类似的控制依赖图[6-7]。

## 2.3 程序依赖图

控制依赖关系描述程序控制流的序列关系, 数据依赖关系描述程序中变量的影响和被影响的关系, 刻画控制依赖关系和数据依赖关系的工具分别是控制依赖图和数据依赖图。为了从整体上统一分析控制依赖关系和数据依赖关系, 提出了基于控制依赖图和数据依赖图构建程序依赖图(Program Dependence Graph, PDG), 若 $G_c$=(*V*, *C*)和 $G_d$=(*V*, *D*)分别为程序 *P* 的控制依赖图和数据依赖图, 则程序依赖图 $G_p$=(*V*, *E*), 其中 *E*=*C*∪*D*∪*X*, 其中 *X* 表示程序中的补充依赖关系[14-17]。

## 2.4 程序切片算法

程序切片关注指令序列对特定语句和变量的影响, 因此基于特定语句和变量制定切片准则, 用于执行程序切片[18-19]。切片准则包含两个要素, 即切片目标变量和切片初始代码位置。程序 *P* 的切片准则可以描述为一个二元组<*n*, *V*>, 其中 *n* 是程序中一条语句或基本块的编号, *V* 是切片所关注的变量集合, 该集合是 *P* 中变量的一个子集。

执行程序切片的一般步骤:

**1)程序依赖关系提取**: 包括控制依赖分析和数

据依赖分析, 以及程序依赖图的构建。

**2)切片准则构建**: 基于程序分析需求设计一组切片准则, 可以是程序中的变量定义、变量引用、过程调用的特定参数、常量、以及应用程序接口(Application Programming Interface, API)。

**3)程序切片生成:** 选取合适的程序切片算法, 基于已构建的切片准则, 执行程序切片算法, 提取程序的代码片段。

**算法 4 基于图可达性切片算法**

**输入**: $CFG=\{V,E,Entry,Exit\}$
$PDG=(V_p,E_p)$
$SlicingCriterion=(n,\{variable\})$
**输出**: $Slice\rightarrow\phi$
**过程**:
1: *ReachableGraphSlice*(Node *n*)
2: IF *n.visited*==false
3: *n.visited*=true
4: *Slice*=∪{*n*}
5: FOR *s* IN *n.children*
6: *Slice*=∪*ReachableGraphSlice*(*s*)
7: RETURN *Slice*

算法 4 描述了基于图可达性的程序切片算法, 其中, 输入参数包括控制流图 *CFG*、程序依赖图 *PDG* 以及切片准则 *SlicingCriterion*, 通过递归遍历程序依赖图, 生成程序依赖图的一个子图, 即为针对该切片准则的一个程序切片。

## 3 二进制代码切片改进

由于程序切片技术最初用于软件工程领域的程序调试和排错, 因此经典程序切片算法几乎都是针对高级程序设计语言作为程序切片对象。然而在软件安全分析领域, 如恶意代码检测和安全漏洞分析方面, 二进制可执行程序(Binary Executables)是主要的分析对象, 二进制代码与高级语言相比, 存在信息丢失(如变量的数据类型信息丢失)和单语句/指令更复杂(如一条指令同时改变通用寄存器、状态寄存器和内存数据)等差异, 因此已知的经典程序切片算法对二进制代码的切片应用仍存在很多挑战[20]。尽管面向高级语言的程序切片技术得到了广泛研究, 但是很少有面向二进制代码切片的深入研究[21-22]。本文基于二进制代码特性和恶意代码检测应用场景, 改进了静态程序切片算法, 主要包括如下 2 个特性:

1)过程间切片(Interprocedural Slicing)改进: 对高级语言系统依赖图进行优化, 识别过程间调用时形参和实参的输入和输出在系统依赖图中的节点, 减少图中冗余的节点和边, 降低图计算复杂度。改进过程将在 3.1 节中详细描述。

2)切片粒度(Slicing Granularity)优化: 针对二进制代码多赋值指令——一条指令中隐式包含多个输入和输出的情形, 将该指令转换为单赋值的中间表示, 并对该指令对应的系统依赖图的节点进行扩展, 将该指令的中间表示作为一个子过程调用, 转化为过程间调用的依赖关系分析问题, 从而改善汇编代码的多赋值特性带来的切片粒度问题。改进过程将在 3.2 节中详细描述。

### 3.1 过程间切片

本文第 2 章详细描述了由 M. Weiser 提出的经典程序切片算法以及一些扩展算法研究, 这些算法的作用范围通常是在一个过程或函数中的切片——过程内切片(Intraprocedural Slicing)。随着计算机程序越来越复杂, 许多应用领域要求实现更为复杂的过程间切片辅助理解程序的结构、功能、行为和逻辑设计。过程间切片与过程内切片相比, 除了需要提取程序中每一个过程的控制依赖关系和数据依赖关系以外, 还需要提取过程调用关系和过程间参数传递依赖关系, 因此过程间切片的作用范围是整个程序的依赖关系[23-26]。

S. Horwitz 等提出了基于系统依赖图的过程间程序切片方法, 该方法的核心思想是通过对程序依赖图进行扩展, 添加用于刻画过程调用以及参数传递的节点和边, 并生成系统依赖图, 最后应用算法 4 描述的图可达性算法实现过程间程序切片[27]。由于该方法是面向高级语言的程序切片, 因此基于高级语言的特点, 需要在程序依赖图中对每一个过程调用扩展 5 类节点和 3 类边用以描述过程间的依赖关系。表 1 中列出了高级语言过程间切片需要增加的节点和边, 由于高级语言过程调用中的实参(Actual Parameter)和形参(Formal Parameter)允许不相同的变量名称表示, 因此在程序依赖图中需要添加实参输入和输出、形参输入和输出节点用于在图中描述参数传递过程, 而在二进制代码中, 如 x64 架构下的过程调用, 前 6 个参数使用寄存器传参, 第 7 个及更多的参数使用堆栈传递, 实参和形参实际上是同一个寄存器或堆栈地址, 因此二进制代码中不需要添加额外的参数传递节点即可以描述参数传递关系, 从本质上看过程间的参数传递与 2.2 节中描述的“定义—引用”数据依赖关系相同, 可以保持与过程内数据依赖分析相一致的处理过程。基于上述分析, 本文对基于高级语言的系统依赖图生

成方法进行改进, 并在表 1 中进行了对比, 本文提出的改进方法减少了图中顶点的数量, 降低了图的复杂度, 并且使用数据依赖分析方法描述过程间参数传递关系。

```
1: int add(int a, int b) {
2:     return a + b;
3: }
4: int main() {
5:     int sum = 0;
6:     for (int i = 1; i < 100; i++)
7:         sum += add(sum, i);
8:     return sum;
9: }
```

```
1:      main:  push    rbp
2:             mov     rbp, rsp
3:             sub     rsp, 10h
4:             mov     [rbp+sum], 0
5:             mov     [rbp+i], 1
6:             jmp     short loc_1167
7:  loc_1151:  mov     edx, [rbp+i]
8:             mov     eax, [rbp+sum]
9:             mov     esi, edx        ; b
10:            mov     edi, eax        ; a
11:            call    add
12:            add     [rbp+sum], eax
13:            add     [rbp+i], 1
14: loc_1167:  cmp     [rbp+i], 64h ;
15:            jle     short loc_1151
16:            mov     eax, [rbp+sum]
17:            leave
18:            retn
```

```
1:      add: push    rbp
2:           mov     rbp, rsp
3:           mov     [rbp+a], edi
4:           mov     [rbp+b], esi
5:           mov     edx, [rbp+a]
6:           mov     eax, [rbp+b]
7:           add     eax, edx
8:           pop     rbp
9:           retn
```

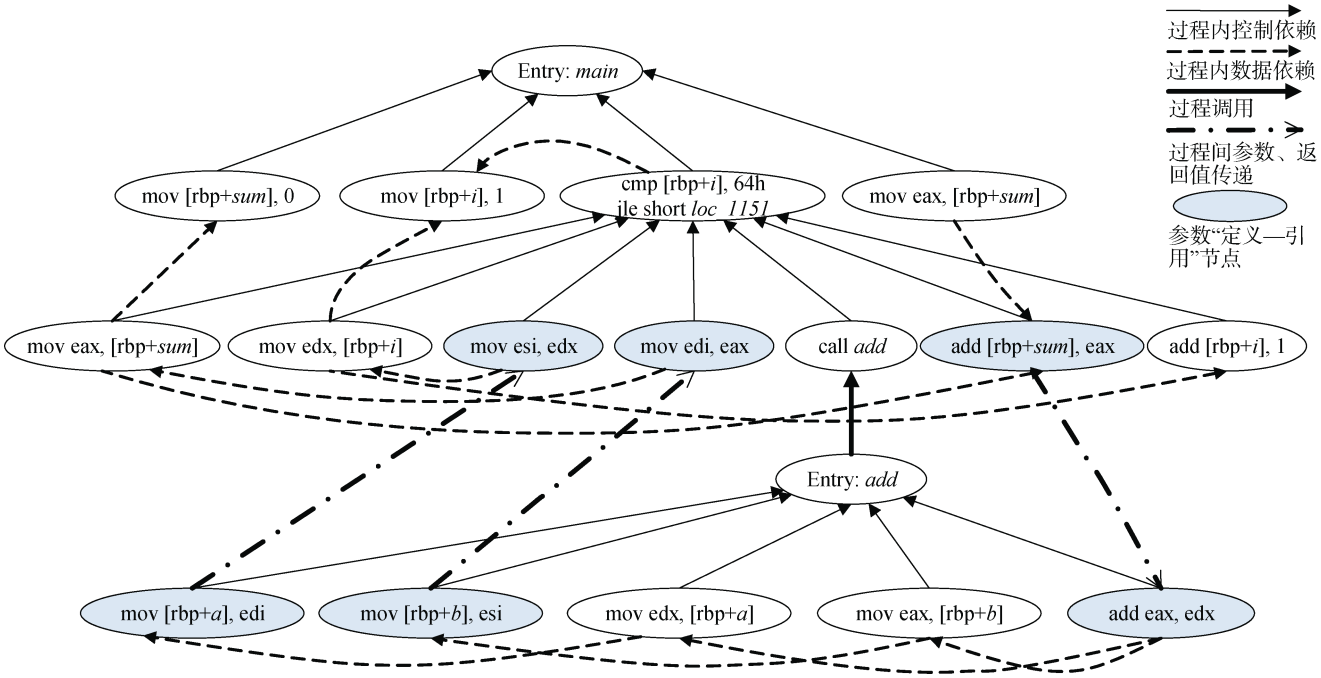


**图 1　改进的二进制代码过程间切片示例**

**Figure 1　A sample of improved interprocedural slicing of binary executables**

**表 1　二进制程序系统依赖图改进**

**Table 1　Improved binary executables' SDG**

| | 高级语言 | 二进制代码改进 |
|---|---|---|
| 基于 PDG 扩展的节点 | 过程调用点 | 过程调用点 |
| | 实参输入 | / |
| | 实参输出 | / |
| | 形参输入 | / |
| | 形参输出 | / |
| 基于 PDG 扩展的边 | 调用点→过程入口 | 调用点→过程入口 |
| | 参数传入子过程 | 寄存器/堆栈数据依赖 |
| | 实参输入→形参输入 | 定义→引用 |
| | 结果返回父过程 | 寄存器/堆栈数据依赖 |
| | 形参输出→实参输出 | 定义→引用 |

图 1 是本文提出的二进制代码过程间切片的一个示例, 图中上方分别为 C 语言编写的示例代码以及使用 IDA Pro 反汇编生成的对应 x64 架构汇编代码, 在图中的系统依赖图中, *main* 函数调用 *add* 函数时传递了 2 个参数, 保存在 EDI 和 ESI 寄存器中, *add* 函数中使用 EDI 和 ESI 寄存器中的值进行运算, 因此参数传递过程本质上是寄存器 ESI 和 EDI 基于过程调用控制流的数据依赖关系, 从引用节点向定义节点引出一条边就可以描述过程间调用的参数传递关系。

## 3.2　切片粒度

在二进制代码中, 如 x86、x64、ARM 等架构的汇编语言代码, 许多指令是多赋值的, 即一条指令中包括对多个变量的定义, 如在 PUSH EAX 指令中, 同时包含 ESP=ESP-4 和[ESP]=EAX 两条微指令(Microcode), 分别对 ESP 寄存器和[ESP]堆栈地址进行赋值。由于二进制代码的多赋值属性是一种隐式数据依赖, 因此给二进制代码分析如程序切片、符号执行和污点分析等带来了巨大的挑战[28]。

图 2 和图 3 的示例描述了二进制代码切片与高级语言程序切片的差异。在图 5 的高级语言程序示例中, 我们以 *main* 函数的返回语句和返回值作为切

片准则执行经典的后向切片(在图中以黑框标示)。在程序中, *multiply* 函数与 *main* 函数返回值不存在数据依赖关系, 因此程序切片不包含 *multiply* 函数(程序切片在图中以阴影标示)。同时, 我们对图 2 中的示例代码使用 IDA Pro 反汇编生成图 3 中的二进制汇编代码, 并使用相同的切片准则执行后向切片算法(包含在切片中的指令在图中以阴影标示), 我们发现, 切片包含 *multiply* 函数中的全部指令, 而 *multiply* 函数在对应的高级语言程序切片是不相关的代码, 这使得程序切片包含了过多冗余代码从而降低了切片的精度。为什么在二进制切片中包含了 *multiply* 函数的代码呢? 原因在于切片准则(第 29 行)中包括了 EBP 寄存器的引用, 而第 8 行指令 LEAVE 是一条多赋值指令, 语义上等价于 MOV ESP, EBP 和 POP EBP 两条指令, 并且 POP EBP 指令也是一条多赋值指令, 语义上等价于 EBP=[ESP]和 ESP=ESP+4, 即 EBP 寄存器的定义依赖于 ESP 的值指向的堆栈地址, 因此当把第 8 行 LEAVE 指令作为一个整体添加到切片时, 由于多赋值指令无法拆分为更小单元, 因此当 LEAVE 被添加到切片中时, 会将更多的冗余指令序列引入到切片中, 这就是二进制代码的多赋值属性在切片过程引发的粒度问题, 即在某些情况下, 虽然我们希望切片只包含一条指令中若干微指令的一个子集, 但切片算法必须包含整个指令。此外切片粒度问题还会引发级联效应(Cascade Effect), 不相关的指令将导致越来越多的不相关指令被切片准则命中。

```
1: int multiply(int a, int b) {
2:     int c = a * b;
3:     return c;
4: }
5: int main() {
6:     int a = 1, b = 2;
7:     int c = multiply(a, b);
8:     return a * b;
9: }
```

**图 2 C 语言后向静态切片示例**

**Figure 2 A sample of backward static slicing for C**

```
multiply:
 1: push   ebp
 2: mov    ebp, esp
 3: sub    esp, 10h
 4: mov    eax, [ebp+a]
 5: imul   eax, [ebp+b]
 6: mov    [ebp+c], eax
 7: mov    eax, [ebp+c]
 8: leave
 9: retn

main:
10: push   ebp
11: mov    ebp, esp
12: sub    esp, 10h
13: mov    [ebp+a], 1
14: mov    [ebp+b], 2
15: push   [ebp+b]     ; b
16: push   [ebp+a]     ; a
17: call   multiply
18: add    esp, 8
19: mov    [ebp+c], eax
20: mov    eax, [ebp+a]
21: imul   eax, [ebp+b]
22: leave
23: retn
```

**图 3 汇编语言后向静态切片示例**

**Figure 3 A sample of backward static slicing for assembly**

一些学者已开始研究将二进制汇编语言转换成某种中间表示, 以应对二进制代码切片引入的粒度问题, 这种中间表示必须支持一个关键属性即静态单赋值(Static Single Assignment, SSA), 它要求在一条指令中仅为一个变量赋值一次, 并且在使用一个变量之前必须定义该变量[29-31]。由于静态单赋值表示的“定义—引用”链是显式的, 因此转换为中间表示的二进制代码可以和高级语言一样运用经典的程序切片算法且不会包含过多不相关的冗余指令。

本文中, 我们选取 VEX 中间表示[32-33]作为二进制代码切片的切片表示, 并将“二进制代码—中间表示指令序列”的映射关系转换看作过程间调用关系, 再应用我们在 3.1 节提出的过程间切片算法对包含中间表示的系统依赖图执行程序切片, 进而尽可能消除切片中的不相关冗余代码。图 4 是 x86 架构下 LEAVE 指令的 VEX 中间表示, 通过 LEAVE 指令转换为 6 行中间表示代码, 并且每行中间代码都是静态单赋值的, 因此满足上述中间表示的要求。我们可将 VEX 微指令序列看作对应二进制汇编指令位置调用的子过程, 从更细的粒度上描述多赋值的二进制指令。图 5 是将二进制代码转换成中间表示并生成改进的系统依赖图示例, 我们对图 3 中 *multiply* 函数的系统依赖图进行扩展, 将图 3 第 8 行 LEAVE 指令转换为中间表示, 并把中间表示看作 LEAVE 的子过程, 最后将二进制代码和局部中间表示作为一个整体生成 EBP 寄存器的数据依赖关系。由于中间表示显式的描述了 EBP 寄存器的“定义—引用”链, 因此在 *multiply* 函数中对 EBP 寄存器执行切片时, 系统依赖图可以准确的描述 EBP 的依赖关系, 仅 MOV EBP, ESP 和 LEAVE 两条指令包含在切片中。

根据以上分析, 对于任意给定的二进制可执行程序, 应用中间表示生成的程序切片的非形式化过程如下:

```
IRSB {
  t0:Ity_I32 t1:Ity_I32 t2:Ity_I32 t3:Ity_I32

  00 | ------ IMark(0x400400, 1, 0) ------
  01 | t0 = GET:I32(ebp)
  02 | PUT(esp) = t0
  03 | t1 = LDle:I32(t0)
  04 | PUT(ebp) = t1
  05 | t2 = Add32(t0,0x00000004)
  06 | PUT(esp) = t2
  NEXT: PUT(eip) = 0x00400401; Ijk_Boring
}
```

**图 4 LEAVE 指令的 VEX 中间表示**

**Figure 4 VEX IR of LEAVE instruction**

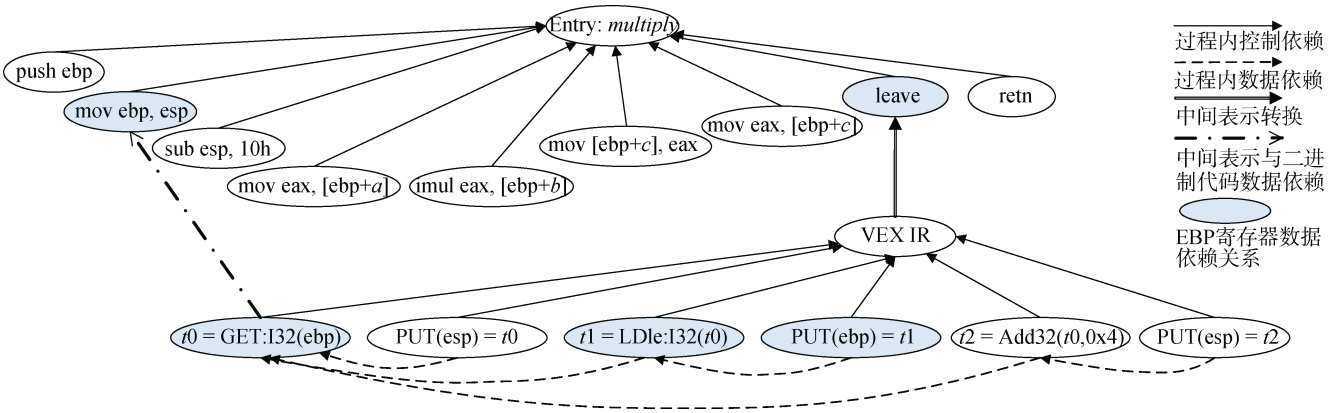


**图 5　带中间表示的改进系统依赖图**

**Figure 5　Improved SDG with IR**

1)分析二进制代码的控制依赖关系和数据依赖关系, 应用 3.1 节所述过程间切片方法生成系统依赖图;

2)对二进制指令中的多赋值指令, 如 x86 架构下的 ENTER、LEAVE、PUSH、POP 等转换为 VEX 中间表示, 将中间表示的指令序列看作子过程调用, 对系统依赖图进行局部扩展, 重新生成局部控制依赖关系和数据依赖关系;

3)运用基于图的可达性程序切片算法, 对 2)中改进的系统依赖图生成二进制代码切片。

## 4　基于二进制切片的恶意代码检测模型

恶意代码检测技术发展至今, 主要有两种检测方法:

**1)规则式检测(Rule-based Detection)**: 恶意代码检测引擎基于恶意代码特征规则库对样本进行检测, 规则库主要包括针对恶意指令的指纹特征和针对恶意行为的模式特征。该检测方法准确率较高、检测时间较短, 但需要预先定义规则, 因此对于已知样本的检测具有较好的效果, 对于未知样本的检测能力相对较弱。

**2)启发式检测(Heuristic Detection)**: 通过监视系统的活动并将其分类为正常或异常两种状态来检测样本是否具有恶意的企图。当前对异常状态的判断通常基于机器学习算法, 这需要恶意代码检测引擎进行一段时间的训练和建模。由于该检测方法基于统计特征和概率决策模型, 因此通常具有一定的误报率, 并且检测性能较低, 但对于未知样本检测相对于规则式检测具有较高的召回率。

图 6 描述了恶意代码检测一般模型, 当待检测样本文件投入到恶意代码检测引擎中, 检测引擎首先提取样本文件特征并与引擎内置规则库进行比较, 进而判断样本是否为恶意样本。若基于规则式检测无法判别结果, 引擎进一步通过启发式检测对样本多维度特征的状态进行分类, 并由专家基于分类结果进行决策, 最终输出检测报告。

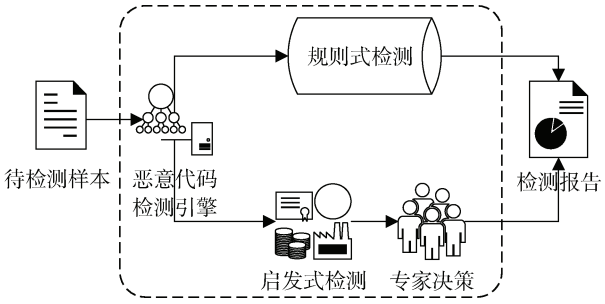


**图 6　恶意代码检测一般模型**

**Figure 6　General model of malware detection**

随着新型恶意代码的出现, 隐形恶意代码(Stealth Malware)、多态恶意代码(Polymorphic Malware)、加密恶意代码(Encrypted Malware)和多歧恶意代码(Multipartite Malware)使得基于规则的恶意代码检测方法难以有效检测, 因此恶意代码检测研究倾向于对启发式检测能力进行增强。其中, 恶意软件同源性分析(Malware Homology Analysis)是一种借助于威胁情报和大数据分析方法的恶意代码检测和分析方法[34-35], 也是当前的恶意代码分析和溯源研究的热点问题之一。网络攻击组织通常利用系统漏洞或社会工程传播恶意代码, 获得目标主机权限或窃取敏感数据。由于同一网络攻击组织通常使用相同家族的恶意代码, 或同一作者编写的恶意代码, 因此恶意样本间具有内在相关性和相似性[36]。恶意代码同源性分析采集威胁情报中网络攻击组织关联的所有已知恶意样本, 提取样本关键特征并建模, 使用该模型对待检测样本进行分类, 判断是否与已知攻击组织的恶意样本具有同源性进而判断待检测样本

是否为恶意样本。

### 4.1 恶意代码同源性检测工作流

Jieran Liu 等[37]学者提出了一种基于图嵌入网络(Graph Embedding Network)的恶意代码同源性分析方法，该方法提取程序控制流图和代码统计特征，结合了图嵌入特征表示算法和深度学习模型，对来自于同一网络攻击组织的恶意代码建立分类模型，并使用该模型对待检测样本进行分类预测。Jieran Liu 等基于该方法实现了原型系统 MCrab，系统的工作流如下:

1)采集威胁情报中与网络攻击组织相关联的恶意代码样本集。

2)提取恶意代码样本中对应的原始特征，包括汇编指令序列、函数(过程)入口点集合、和函数(过程)内控制流图等样本代码的基础特征。

3)对恶意代码样本原始特征进行处理和泛化，包括 2 个部分: 一是对函数内基本块的特征进行泛化，包括字符串常量数、内存操作指令数、转移指令数、CALL 指令数、总指令数和算术指令数共 6 类特征; 二是对函数内控制流图执行图嵌入算法，将图结构转化为 $n$ 维特征向量。

4)使用深度学习算法 LSTM 对第 3 步中经过处理和泛化的多维恶意代码特征向量进行训练，生成不同网络攻击组织的恶意代码同源性检测模型。

5)当待检测样本输入到检测模型时，模型对样本是否与某个网络攻击组织的关联恶意代码集做出相似性判断，供专家决策，并将判别结果反馈到威胁情报中，形成检测模型持续优化的闭环。

在上述 MCrab 原型系统的工作流中，首先对二进制可执行文件的所有函数提取控制流图以及基本块的指令统计特征，并对特征进行清洗和预处理，最后使用机器学模型进行训练。尽管实验表明 MCrab 具有较高的准确率，然而，对于二进制可执行程序而言，程序从源代码经过编译、链接生成可执行程序，并非所有的代码片段都用于实现程序功能，大多数编译器会在插入一些代码用于异常处理和执行优化，而恶意代码作者也可能插入无用代码对抗安全分析。因此，有必要在训练检测模型之前从样本原始特征中去除无用的代码片段，以提高恶意代码检测模型的准确率和召回率。基于经验的归纳，二进制可执行程序中存在如下 2 种无用代码片段:

1)代码片段在控制依赖关系上不可达: 在函数间调用图和函数内控制流图上均不存在从入口点到该代码片段的路径;

2)代码片段在数据依赖关系上不可达: 代码片段在函数调用参数和函数内变量上均无定义-引用关系。

为了更有效提取恶意代码特征，本文改进了 Jieran Liu 提出的恶意代码同源性检测方法，在图 7 中描述了基于 MCrab 原型系统改进的检测方法工作流，我们进一步对提取的恶意代码样本原始特征分

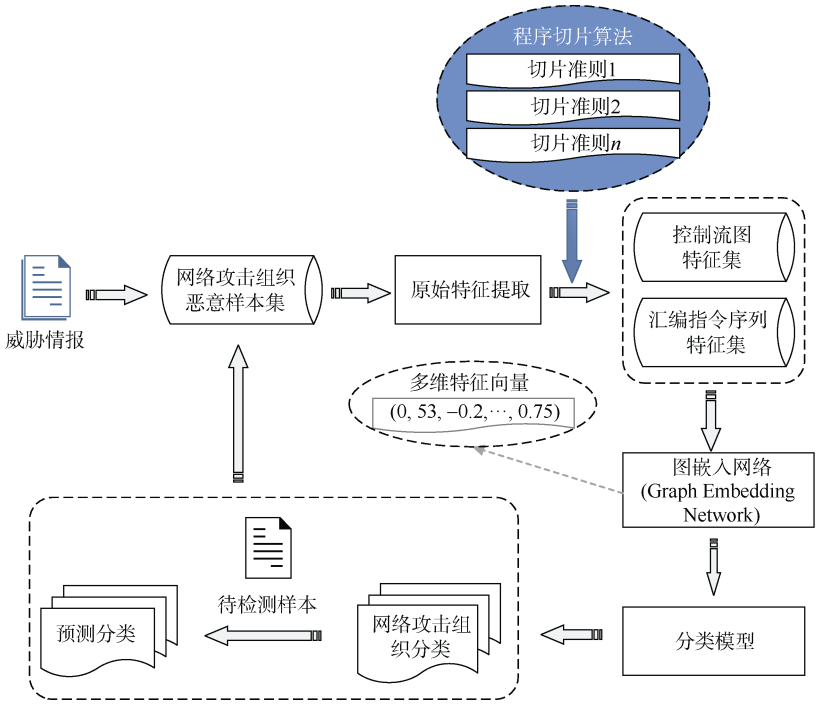


**图 7 恶意代码同源性检测工作流**

**Figure 7 Workflow of malware homology detection**

析二进制代码的控制依赖关系和数据依赖关系, 输出系统依赖图, 并根据预先设定的一组表征恶意行为的切片准则执行程序切片, 最后将每组程序切片输入到图嵌入网络中生成多为特征向量, 最后训练检测模型。本文对 MCrab 的优化所使用的程序切片算法是对经典程序切片算法面向二进制代码的改进, 并已在第 3 章中进行了阐述。

### 4.2 切片准则选取

为了对恶意代码的原始特征进行清洗和预处理, 提高分类模型的准确率和召回率, 我们通过人工分析 10 个网络攻击组织所使用的传播范围为 TOP 100 的恶意代码样本, 设计了一组被恶意代码广泛使用的切片准则[38-39], 分别为基础行为切片准则和反分析行为切片准则共两类, 切片准则清单在表 2 中列出。其中, 基础行为切片准则包括文件行为、注册表行为、网络行为、进程/线程行为等共 22 条; 反分析行为切片准则包括反虚拟机行为、反沙盒行为和反调试行为共 20 条, 合计提取 42 条关键切片准则。

依据恶意代码的恶意行为的共性特征, 本文对经典切片准则进行扩展, 将其描述为如下三元组: **<{*API, instructions*}, {*parameters*}, {*return values*}>**。其中, 第一个元素{*API, instructions*}用于定位程序切片的起始位置, 包括关键 *API* 调用或关键汇编指令; 第二个元素{*parameters*}是 *API* 的参数, 描述程序切片起始位置的后向(Backward)数据依赖关系; 第三个元素{*return values*}是 *API* 的返回值(在 IA 架构下由 EAX 或 RAX 寄存器保存)以及通过指针方式传入的参数返回值, 该元素描述程序切片起始位置的前向(Forward)数据依赖关系。表 3 以反沙盒行为切片准则为示例, 列出了本文所提取的恶意代码用于检测沙盒行为的 8 条切片准则, 其余切片准则均遵循上述的三元组进行定义, 并提取能够表示恶意代码关键行为的特征。

**表 2 切片准则清单**

**Table 2 Slicing criterion list**

| | 程序行为类型 | 程序切片数量 |
|---|---|---|
| 基础行为 | 文件行为 | 4 |
| | 注册表行为 | 4 |
| | 网络行为 | 3 |
| | 进程/线程行为 | 5 |
| | Mutex 操作 | 2 |
| | Service 操作 | 2 |
| | DLL 操作 | 2 |
| 反分析 | 反虚拟机检测 | 10 |
| | 反沙盒检测 | 8 |
| | 反调试检测 | 2 |

**表 3 反沙盒行为切片准则列表**

**Table 3 Slicing criterion list of anti-sandbox behaviors**

| 切片准则编号 | 切片准则描述 | 调用 API | 参数 | 返回值 |
|---|---|---|---|---|
| 01 | 检测光标位置变化判断是否位于沙盒环境 | call GetCursorPos | | |
| 02 | 检测用户名判断是否位于沙盒环境 | call GetUserName | | “SANDBOX” “VIRUS” “MALWARE” |
| 03 | 检测程序运行目录判断是否位于沙盒环境 | call GetModuleFileName | | “\\SAMPLE” “\\VIRUS” |
| 04 | 获取磁盘大小判断是否位于沙盒环境(方式一) | call CreateFile | 参数 1: “\\\\.\\PhysicalDrive0” | |
| 05 | 获取磁盘大小判断是否位于沙盒环境(方式二) | call GetDiskFreeSpace | | |
| 06 | 检测时间判断沙盒时间加速 | call GetTickCount | | |
| 07 | 获取 CPU 核心数判断是否位于沙盒环境 | mov eax, fs:0x18 | | |
| 08 | 检测特定模块判断是否位于沙盘(Sandboxie)环境 | call GetModuleHandle | 参数 1: “sbiedll.dll” | |

## 5 实验和评估

为验证第 4 章中所描述的基于二进制代码切片的恶意代码同源性分析方法的效果, 本文以 MCrab 数据集为实验对象, 并以 MCrab 作为评估基线, 对本文提出的方法进行评估。

### 5.1 数据集预处理

本实验使用的数据集包括从公开威胁情报中获取的 10 个具有广泛影响的网络攻击组织或安全事件所使用的恶意代码样本集合计 4053 个, 使用二进制代码分析工具 IDA Pro 对所有样本进行分析, 提取样本原始特征, 包括每个恶意代码样本的自定义函数清单, 输出每个函数的汇编指令序列和控制流图。我们的实验首先使用 MCrab 对数据集提取原始特征, 共获取自定义函数 364626 个。作为对比, 我们基于 4.2 节中定义的切片准则, 执行程序切片算法对样本的原始特征进一步预处理, 提取与恶意行为关键代码片段存在依赖关系的自定义函数, 获取

304675 个自定义函数(占原始特征中全部自定义函数的 83.6%)。表 4 列出了数据集中 10 个网络攻击组织或安全事件的恶意代码样本数和自定义函数对比情况。

**表 4 数据集预处理结果列表**

**Table 4 List of dataset preprocessing result**

| 网络攻击组织/事件 | 恶意代码样本数 | 全部函数个数 | 程序切片函数个数 | 占比(%) |
|---|---|---|---|---|
| stuxnet | 597 | 39472 | 27893 | 71 |
| hangover | 586 | 40536 | 35672 | 88 |
| patchwork | 482 | 38560 | 32390 | 84 |
| apt28 | 474 | 38868 | 33038 | 85 |
| lazarus | 389 | 41311 | 34701 | 84 |
| darkhotel | 356 | 39160 | 33678 | 86 |
| deeppanda | 331 | 38065 | 31975 | 84 |
| apt10 | 302 | 30078 | 24965 | 83 |
| gaza | 276 | 28704 | 24972 | 87 |
| turla | 260 | 29872 | 25391 | 85 |
| 合计 | 4053 | 364626 | 304675 | 84 |

在数据集预处理过程中, 程序切片算法基于 Triton[40]二进制分析框架开发, 使用 Triton 内置的抽象语法树(Abstract Syntax Tree)实现中间层语义表示, 并通过调用 Microsoft Z3 约束求解器接口实现静态符号执行, 获取系统依赖图, 最后基于图的可达性提取程序切片。此外, 我们使用 Valgrind 工具将多赋值的二进制指令转换为 VEX 中间表示。由于本实验采用静态程序切片, 因此在不确定程序实际执行路径的情况下, 我们选择了相对保守的程序切片参数, 从表 4 中看出, 10 个网络攻击组织或安全事件中, 除了 stuxnet“震网病毒”事件以外, 执行程序切片后的自定义函数个数约占全部函数个数的 83%~88%, 通过人工选取数据集中的典型样本验证, 未包含在程序切片中的函数主要是编译器自动添加的辅助代码和无交叉引用关系的函数。其中, stuxnet 事件样本的程序切片后函数个数占全部函数个数的比例为 71%, 通过人工分析 597 个 stuxnet 样本中的 10 个典型样本, 我们发现 stuxnet 中的许多样本仅包括由汇编语言编写的漏洞利用代码, 而不包括恶意代码中常见的远程控制、数据窃取、逃逸检测、持久化等通用恶意功能, 因此依据 4.2 节中设置的切片准则, 获取的程序切片较小。总的来看, 程序切片后的代码集最大的保留了恶意代码中的关键恶意行为特征。

## 5.2 实验结果分析

我们以 MCrab 实验结果作为基准, 与本文提出的使用程序切片处理后的样本特征训练的模型进行比较。同时, 我们也选取了典型机器学习算法对样本原始特征进行训练, 并与本文方法进行对比。实验步骤描述如下:

1)代码块内统计特征提取: 采用卷积神经网络(Convolutional Neural Networks, CNN)算法, 将代码块内的汇编指令序列作为文本进行处理, 输出代码块内多维特征向量;

2)函数内(基本块间)控制流图提取: 采用 Ribeiro 等提出的 Struc2Vec 图嵌入网络算法[41], 将控制流图结构转换为多维特征向量;

3)样本预测分析模型: 采用 MCrab 中提出的改进的长短期记忆时间循环神经网络(Long Short-Term Memory, LSTM), 通过该模型预测待检测样本所属的网络攻击组织或安全事件分类。

实验将数据集按照 8∶2 的比例随机分割为训练集和测试集, 分别应用不同的模型进行训练和验证。表 5 显示了文本方法与多种经典模型的对比, 其中, 除 LSTM 模型外其他模型的准确率均低于 90%, 本文在 MCrab 模型的基础上对样本原始特征执行程序切片后, 准确率达到 94.7%, 极大的改进了恶意代码样本分类的准确程度。

作为基线, 我们同时详细比较了 Mcrab 与本文方法, 图 8 描述了 *epoch* 取值 1~160 之间的准确率和损失函数值对比。其中, 图 8(a)所示, 本文的方法当迭代次数为 30 即 *epoch*=30 时的模型准确率即达到 90%, 而 MCrab 当迭代次数为 *epoch*=48 时准确率达到 90%, 经过 160 轮迭代后, 本文方法的准确率为 94.7%, 而 MCrab 的结果为 94.2%。另一方面, 在图 8(b)所示的损失函数值对比中, 本文方法比 MCrab 生成了具有更低损失函数的模型。

**表 5 经典模型实验结果对比**

**Table 5 Compare with state-of-art models**

| 模型 | 模型算法 | 准确率 (%) |
|---|---|---|
| SVM | Bigram+SVM | 76.7 |
| | Bigram+LR | 73.1 |
| | NBSVM | 82.4 |
| CNN | CNN-static | 83.1 |
| | CNN-non-static | 87.3 |
| | CNN-multichannel | 88.1 |
| LSTM | LSTM | 90.4 |
| | Bi-LSTM | 89.0 |
| MCrab | CNN-SLSTM | 94.2 |
| 本文实现 | CNN-SLSTM (程序切片预处理) | 94.7 |

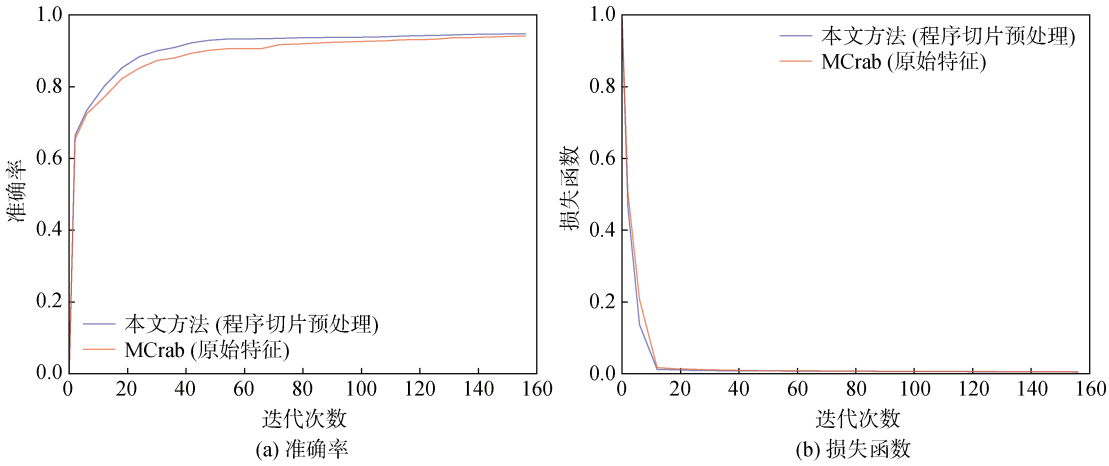


**图 8　Mcrab 实验结果对比**

**Figure 8　Comparison with experiment result of Mcrab**

### 5.3　方法局限性讨论

本文的方法由于对原始特征执行了基于程序切片的预处理, 因此无论与经典机器学习模型对比, 还是与参考基线 Mcrab 比较, 在模型训练效率和分类准确率上均有不同程度改善。然而, 通过对分类错误的恶意代码样本进行了人工分析, 本方法仍存在如下局限性:

**1)面向内存数据的数据依赖分析:** 与许多高级语言不同, 汇编语言不是强类型语言, 操作码仅可以表示操作数宽度, 并且在同一个作用域内可以通过寄存器、立即数和内存地址引用等多种方式定义或引用指定内存地址的数据, 因此这对于数据依赖分析带来了挑战。如 REP STOSB 指令, 从 ESI 寄存器指向的源地址向 EDI 寄存器指向的目的地址拷贝数据, 数据长度由 ECX 寄存器指定, 该指令的操作完成后使得[EDI]至[EDI+ECX]的内存区域被重新定义, 但对于该区域的数据依赖分析难以通过汇编代码语义的静态分析实现。其他如 MOVS、LODS 等指令亦存在类似的情形, 这降低了切片分析的准确度, 从而影响本文方法的样本原始特征的处理。

**2)间接函数调用**: 由于本文提出的方法仅针对样本原始特征执行程序切片, 而样本的原始特征是基于 IDA Pro 工具静态分析得到的汇编指令序列和函数调用图, 因此对于类似 CALL DWORD PTR [EAX]的指令无法通过静态分析函数调用关系。此外, 许多恶意代码中包含了直接使用汇编语言编写的 Shellcode, 导致 IDA Pro 无法解析, 使得程序切片算法无法构建系统调用图。图 9 中的示例中, 位于.text:00402235 的 call $+5 指令是 Shellcode 中常见的获取当前地址的方法, 但 IDA Pro 无法正确解析; 位于.text:0040223E 处的 JMP EDX 指令在静态分析时也无法确定跳转目的地址。

**3)漏洞利用样本**: 对于通过漏洞传播的恶意代码, 样本中包括特定的漏洞利用代码片段, 由于不同类型的漏洞利用方式各不相同, 因此在文本提出的 42 条切片准则中无法枚举所有的漏洞利用方式。值得说明的是, 如果恶意代码在漏洞利用后实现了远程控制、数据窃取、逃逸检测等恶意功能, 仍可通过 4.2 节中定义的切片准则提起关键特征。对于数据集中的极少数仅包括漏洞利用代码片段的恶意代码, 应用本文的程序切片后将导致漏洞利用代码的特征被裁剪, 因此无法正确检测和分类。

```
.text:0040221F      mov     ecx, 87D8358h
.text:00402224      add     bh, [ebp+3Eh]
.text:00402227      push    offset aErrorMissingAr ; "Error!\n"
.text:0040222C      call    sub_4025E0
.text:00402231      add     esp, 4
.text:00402234      push    edx
.text:00402235      call    $+5
.text:0040223A      pop     edx
.text:0040223B      add     edx, 7
.text:0040223E      jmp     edx
```

**图 9　间接调用示例**

**Figure 9　A sample of indirect invocation**

## 6　相关工作

程序理解是一种从计算机程序中获取知识信息的过程。这些知识信息可以应用于软件测试、软件安全分析、软件二次开发等方面。程序理解是软件工程领域里的一个重要部分, 运用程序理解方法可以从不同的抽象级别上建立软件的概念模型, 包括

从代码本身的模型到基本应用领域的模型。依据不同的抽象层次, 可以将程序理解分为三类: 设计模型、代码模型和二进制模型。二进制模型的研究对象是二进制代码, 在信息安全领域, 分析人员常需要提取二进制形式的软件内部算法、分析安全漏洞、解密隐蔽信息等, 同时也需要对恶意代码的恶意功能和行为进行分析和理解, 从而确定二进制代码的安全属性[42-43]。

本文讨论的程序切片技术是程序理解方法之一, 是一种通过对程序进行分解进而对程序关键代码进行深入分析的方法。此外, 学者们还针对不同的应用场景提出了 4 种程序理解方法。

**方法一: 模式匹配方法**, 一种在不同的抽象层次上对程序的各种模式进行匹配的过程。该方法按照从低到高的抽象层次建立文本模式、语法树模式、控制流/数据流图模式、概念模式等, 并对于不同的应用领域分别实现上述一个或多个抽象层次的模式匹配, 理解目标程序的结构、功能和行为。

**方法二: 概念赋值和分析方法**, 一种通过将物理世界中概念与计算机逻辑程序进行映射, 进而实现程序理解的方法。概念赋值(Concept Assignment)是一种基于语义的模式匹配, 并将可识别的概念与计算机程序建立映射的一种理解的过程。概念分析(Concept Analysis)把任何对象和内部属性之间的关系转换成代数格, 然后运用数学形式化方法来研究物理概念和逻辑程序间关系。

**方法三: 程序分析方法**, 从是否执行程序的角度看, 程序分析包括静态分析和动态分析两种方式。静态程序分析是在不执行目标程序的状态下, 仅根据选取的模型推断程序功能和行为的过程。静态程序分析包括语法分析、控制流和数据流分析、交叉引用分析等过程; 动态程序分析是在目标程序中发现运行时(Runtime)依赖的过程。它包括对象实例依赖、多态性、函数调用图、并发等分析过程。

**方法四: 智能理解方法**, 运用人工智能和专家系统技术, 提取程序内部的结构、功能、行为和逻辑设计等特征, 通过训练机器学习模型, 对程序进行分类预测或聚类, 辅助进行软件理解。

一些学者已经对方法三和方法四开展了广泛而深入的研究, 以应对网络空间中日益严峻的软件安全漏洞和恶意代码传播, 特别是 DARPA 于 2016 年举办的网络安全挑战赛 Cyber Grand Challenge(CGC)中, 二进制程序分析方法在安全漏洞自动化发现和修复等方面取得了极大的成功。许多研究已开始瞄准如何自动化、体系化、规模化开展二进制程序分析。

B. David 等[28,44]学者提出了一个二进制代码分析框架 BAP, 用于对二进制代码进行逆向工程和程序分析, BAP 将任何输入的二进制汇编代码转换成一种中间表示 BIL, BIL 显式表示了汇编指令的所有隐式特性, 这使得所有的二进制分析都可以基于 BIL 的语法实现自动化处理。BAP 内置了常见的代码表示形式, 包括控制流图、程序依赖图、带有常量合并的数据流框架、死代码消除(Dead Code Elimination)、值集分析(Value Set Analysis, VSA)等。此外, BAP 还支持 ARM, x86, x86-64, PowerPC, MIPS 多种指令架构。

S. Nick 等[45-46]学者提出了一种选择性动态符号执行的模糊测试方法, 并实现了基于模糊测试框架 AFL[47]和二进制代码分析框架 angr[48]的工具 Driller。与其他许多利用符号执行生成输入用例以获得更高代码覆盖率的模糊测试方法不同[49-50], Driller 通过监测模糊器 Fuzzer 的执行过程, 当一轮输入执行完成后未发生基本块转换时(即没有新的控制流图的边被执行), 则对一个具体的输入用例执行动态符号执行, 这种选择性符号执行极大的改善了路径爆炸的问题。其中, Driller 的动态符号执行所依赖的 angr 二进制分析框架, 将二进制代码转换为中间表示 VEX, 基于动态二进制分析框架 Valgrind[32], 实现内存异常检测、线程异常检测、缓存和分支预测、以及堆分析器等二进制代码动态分析能力。

J. Minkyu 等学者通过分析和评估二进制代码分析框架的前端组件(反编汇、中间表示)和后端组件(控制流分析、数据流分析)的依赖关系, 将注意力从大量研究后端组价如何执行二进制分析转移到前端组件中, 关注前端组件如何为后端组件提供支撑, 并实现了一个二进制带分析框架 B2R2[51]。文章提出了一种新的中间表示转换技术, 并利用多核并行计算技术极大的提高了性能, 通过与现有许多二进制分析框架的比较, B2R2 改善了中间表示转换的性能, 并具备良好的后端分析如代码可视化和动态符号执行的能力。

另一方面, 随着人工智能技术的发展, 一些学者已开始将数据挖掘和机器学习算法应用到恶意代码检测中。K. Rajesh 等学者将恶意代码文件的比特流转换为灰度图像, 并运用图像相似性算法, 结合卷积神经网络对恶意代码样本进行预测和分类[52]。F. Xiao 等学者针对 IoT 设备传播的恶意代码, 提取恶意代码 API 调用图特征(而不是传统方法中的 API 调用序列特征), 并利用自编码器网络训练恶意代码检测模型, 并实现了原型工具 BDLF[53]。J. Li 等学者

将关注点瞄准了移动 APP 恶意代码, 通过有选择的提取 APP 权限特征, 结合多种机器学习算法实现恶意代码预测和分析, 文章实现了原型工具 SigPID, 并在实验中展示了其对于大规模恶意代码的快速检测的良好效果[54]。

## 7 结论与展望

本文通过总结程序理解领域中的一种重要方法——程序切片的发展和研究现状, 并针对恶意代码检测的实际应用场景, 基于经典程序切片技术提出了面向二进制代码切片的改进方法, 结合人工分析不同类型的恶意代码样本提取了表示恶意行为特征的 42 条切片准则, 最后, 我们在实验中通过本文方法对恶意代码样本的原始特征进行预处理, 并运用机器学习算法训练模型, 用于恶意代码同源性检测。实验结果表明, 本文提出的方法在恶意代码同源性检测方面改善了检测准确率, 并且降低了检测模型训练时间和迭代次数。

然而, 在本文的实验中, 我们也看到本文提出的方法在检测间接函数调用和特定恶意代码样本(如漏洞利用样本)时仍存在一定的局限性, 未来需要在如下两个方面开展进一步的研究: (1)应用动态程序切片技术改进二进制汇编代码级别上的间接函数调用场景的控制流/数据流分析; (2)进一步完善表征恶意代码行为的切片准则库, 更准确全面地提取恶意代码中的关键恶意行为的程序切片。

**致 谢** 本文的研究得到了国家自然科学基金重点项目(U1736218)和科技部重大专项(2018YFB0804704)的资助, 本文的基础研究平台得到了国家互联网应急中心(CNCERT/CC)运行部的支持, 本文的实验数据集和对照基准参考了 CNCERT/CC 运行部网络安全研究组的科研成果 MCrab 原型系统, 在此表示衷心的感谢。

## 参考文献

[1] Paszkiewicz A, Bolanowski M. Software Development for Modeling and Simulation of Computer Networks: Complex Systems Approach[M]. Towards a Synergistic Combination of Research and Practice in Software Engineering. Cham: Springer International Publishing, 2017: 193-206.

[2] Weiser M. Program slicing[C]. *The 5th International Conference on Software Engineering*, 1981: 439-449.

[3] Weiser M. Program Slicing[J]. *IEEE Transactions on Software Engineering*, 1984, SE-10(4): 352-357.

[4] Silva J. A Vocabulary of Program Slicing-Based Techniques[J]. *ACM Computing Surveys*, 2012, 44(3): 1-41.

[5] 2019 State of Malware. https://resources.malwarebytes.com/files/2019/01/Malwarebytes-Labs-2019-State-of-Malware-Report-2.pdf. Jan. 2019.

[6] Su Purui, Ying Lingyun, Yang Yi. Software Security Analysis and Application[M]. BEIJING: Tsinghua University Press, 2017: 414.
(苏璞睿, 应凌云, 杨轶. 软件安全分析与应用[M]. 北京: 清华大学出版社, 2017: 414.)

[7] Li Bixin. Program Slicing Technique and Applications[M]. BEIJING: Science Press, 2006: 297.
(李必信. 程序切片技术及其应用[M]. 北京: 科学出版社, 2006: 297)

[8] Devkota S, Isaacs K E. CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations[J]. *Computer Graphics Forum*, 2018, 37(3): 453-464.

[9] Xu B W, Qian J, Zhang X F, et al. A Brief Survey of Program Slicing[J]. *ACM SIGSOFT Software Engineering Notes*, 2005, 30(2): 1-36.

[10] Cornelissen B, Zaidman A, van Deursen A, et al. A Systematic Survey of Program Comprehension through Dynamic Analysis[J]. *IEEE Transactions on Software Engineering*, 2009, 35(5): 684-702.

[11] Kinder J, Veith H. Jakstab: A static analysis platform for binaries[C]. *International Conference on Computer Aided Verification*, 2008: 423-427.

[12] Henderson A, Yan L K, Hu X C, et al. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform[J]. *IEEE Transactions on Software Engineering*, 2017, 43(2): 164-184.

[13] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis[M]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.

[14] Korel B. The Program Dependence Graph in Static Program Testing[J]. *Information Processing Letters*, 1987, 24(2): 103-108.

[15] Ferrante J, Ottenstein K J, Warren J D. The Program Dependence Graph and Its Use in Optimization[J]. *ACM Transactions on Programming Languages and Systems*, 1987, 9(3): 319-349.

[16] Baxter W, Bauer H R III. The Program Dependence Graph and Vectorization[C]. *The 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, 1989: 1-11.

[17] Harrold M J, Offutt A J, Tewary K. An Approach to Fault Modeling and Fault Seeding Using the Program Dependence Graph[J]. *Journal of Systems and Software*, 1997, 36(3): 273-295.

[18] Schoenig S, Ducassé M. A Backward Slicing Algorithm for Prolog[M]. Static Analysis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996: 317-331.

[19] Jaffar J, Murali V, Navas J A, et al. Path-Sensitive Backward Slic-

ing[M]. Static Analysis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: 231-247.

[20] Xue H F, Venkataramani G, Lan T. Clone-Slicer: Detecting Domain Specific Binary Code Clones through Program Slicing[C]. *FEAST '18: The 2018 Workshop on Forming an Ecosystem Around Software Transformation*. 2018: 27-33.

[21] AlAbwaini N, Aldaàje A, Jaber T, et al. Using Program Slicing to Detect the Dead Code[C]. *2018 8th International Conference on Computer Science and Information Technology*, 2018: 230-233.

[22] Park J, Jung H, Lee J, et al. An Efficient Technique of Detecting Program Plagiarism through Program Slicing[M]. Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. Cham: Springer International Publishing, 2018: 164-175.

[23] Sui Y L, Xue J L. SVF: Interprocedural Static Value-Flow Analysis in LLVM[C]. *The 25th International Conference on Compiler Construction*, 2016: 265-266.

[24] Kiss A, Jasz J, Lehotai G, et al. Interprocedural Static Slicing of Binary Executables[C]. *Third IEEE International Workshop on Source Code Analysis and Manipulation*, 2003: 118-127.

[25] Bodden E. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot[C]. *SOAP '12: The ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. 2012: 3-8.

[26] Sinha S, Harrold M J, Rothermel G. System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow[C]. *The 21st international conference on Software engineering - ICSE '99*, 1999: 432-441.

[27] Horwitz S, Reps T, Binkley D. Interprocedural Slicing Using Dependence Graphs[J]. *ACM Transactions on Programming Languages and Systems*, 1990, 12(1): 26-60.

[28] Brumley D, Jager I, Avgerinos T, et al. BAP: A Binary Analysis Platform[M]. Computer Aided Verification. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 463-469.

[29] Song D, Brumley D, Yin H, et al. BitBlaze: A New Approach to Computer Security via Binary Analysis[M]. Information Systems Security. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008: 1-25.

[30] Kim S, Faerevaag M, Jung M, et al. Testing Intermediate Representations for Binary Analysis[C]. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017: 353-364.

[31] Srinivasan V, Reps T. An Improved Algorithm for Slicing Machine Code[J]. *ACM SIGPLAN Notices*, 2016, 51(10): 378-393.

[32] Nethercote N, Seward J. Valgrind[J]. *ACM SIGPLAN Notices*, 2007, 42(6): 89-100.

[33] Shoshitaishvili Y, Wang R Y, Salls C, et al. SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 138-157.

[34] Pfeffer A, Call C, Chamberlain J, et al. Malware Analysis and Attribution Using Genetic Information[C]. *2012 7th International Conference on Malicious and Unwanted Software*, 2012: 39-45.

[35] Ruttenberg B, Miles C, Kellogg L, et al. Identifying shared software components to support malware forensics[C]. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2014: 21-40.

[36] Sæbjørnsen A, Willcock J, Panas T, et al. Detecting Code Clones in Binary Executables[C]. *The eighteenth international symposium on Software testing and analysis - ISSTA '09*, 2009: 117-128.

[37] Liu J R, Shen Y, Yan H B. Functions-Based CFG Embedding for Malware Homology Analysis[C]. *2019 26th International Conference on Telecommunications*, 2019: 220-226.

[38] Alazab M, Venkataraman S, Watters P. Towards Understanding Malware Behaviour by the Extraction of API Calls[C]. *2010 Second Cybercrime and Trustworthy Computing Workshop*, 2010: 52-59.

[39] Galal H S, Mahdy Y B, Atiea M A. Behavior-Based Features Model for Malware Detection[J]. *Journal of Computer Virology and Hacking Techniques*, 2016, 12(2): 59-67.

[40] Saudel F, Salwan J. Triton: A dynamic symbolic execution framework[C]. *Information and Communications Technology Security Symposium*, 2015: 31-54.

[41] Ribeiro L F R, Saverese P H P, Figueiredo D R. Struc2vec: Learning Node Representations from Structural Identity[C]. *The 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017: 385-394.

[42] von Mayrhauser A, Vans A M. Program Understanding: Models and Experiments[J]. *Advances in Computers*, 1995, 40: 1-38.

[43] Hoffswell J, Satyanarayan A, Heer J. Augmenting Code with in Situ Visualizations to Aid Program Understanding[C]. *2018 CHI Conference on Human Factors in Computing Systems*, 2018: 1-12.

[44] Binary Analysis Platform. https://github.com/BinaryAnalysis Platform/bap/. May. 2011.

[45] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing through Selective Symbolic Execution[C]. *The 2016 Network and Distributed System Security Symposium*, 2016: 1-16.

[46] Driller: Augmenting AFL with Symbolic Execution. https://github.com/shellphish/driller/. Jan. 2016.

[47] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Jun. 2016.

[48] Shoshitaishvili Y, Wang R Y, Hauser C, et al. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware[C]. *The 2015 Network and Distributed System Security Symposium*, 2015: 1-8.

[49] Godefroid P, Klarlund N, Sen K. DART: Directed Automated Random Testing[C]. *The 2005 ACM SIGPLAN conference on Pro-*

*gramming language design and implementation - PLDI '05*, 2005: 213-223.

[50] Godefroid P, Levin M Y, Molnar D. Sage[J]. *Communications of the ACM*, 2012, 55(3): 40-44.

[51] Jung M, Kim S, Han H, et al. B2R2: Building an Efficient Front-End for Binary Analysis[C]. *2019 Workshop on Binary Analysis Research*, 2019: 1-10.

[52] Kumar R, Zhang X S, Khan R U, et al. Malicious Code Detection Based on Image Processing Using Deep Learning[C]. *The 2018 International Conference on Computing and Artificial Intelligence - ICCAI 2018*, 2018: 81-85.

[53] Xiao F, Lin Z W, Sun Y, et al. Malware Detection Based on Deep Learning of Behavior Graphs[J]. *Mathematical Problems in Engineering*, 2019, 2019: 1-10.

[54] Li J, Sun L C, Yan Q B, et al. Significant Permission Identification for Machine-Learning-Based Android Malware Detection[J]. *IEEE Transactions on Industrial Informatics*, 2018, 14(7): 3216-3225.

**梅瑞** 于 2014 年在北京大学软件工程专业获得硕士学位。现在中国科学院大学网络空间安全专业攻读博士学位。研究领域为系统与软件安全。研究兴趣包括: 恶意代码检测、自动化软件分析。Email: meirui@iie.ac.cn

**严寒冰** 于 2006 年在清华大学计算机科学与技术专业获得博士学位。现任国家互联网应急中心(CNCERT/CC)运行部主任、教授级高级工程师、博士生导师。研究领域为网络空间安全。Email: yhb@cert.org.cn

**沈元** 于 2014 年在河北工业大学计算机专业获得硕士学位, 现在北京航空航天大学网络安全专业攻读博士学位。研究领域为软件安全分析, 网络安全。Email: shyeric@buaa.edu.cn

**韩志辉** 于 2015 年在中国科学院软件研究所获得博士学位。现任国家计算机网络应急技术处理协调中心工程师, 主要研究方向为系统安全与网络安全。研究兴趣包括: 恶意代码监测、网络攻击溯源等。Email: hzh@cert.org.cn