基于底层数据流分析的恶意软件检测方法

韩锦荣[1,2], 张元曈[1,2], 朱子元[1,2], 孟 丹[1,2]

[1]中国科学院信息工程研究所 第五研究室 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

摘要 近些年来, 层出不穷的恶意软件对系统安全构成了严重的威胁并造成巨大的经济损失, 研究者提出了许多恶意软件检测方案。但恶意软件开发中常利用加壳和多态等混淆技术, 这使得传统的静态检测方案如静态特征匹配不足以应对。而传统的应用层动态检测方法也存在易被恶意软件禁用或绕过的缺点。本文提出一种利用底层数据流关系进行恶意软件检测的方法, 即在系统底层监视程序运行时的数据传递情况, 生成数据流图, 提取图的特征形成特征向量, 使用特征向量衡量数据流图的相似性, 评估程序行为的恶意倾向, 以达到快速检测恶意软件的目的。该方法具有低复杂度与高检测效率的特点。实验结果表明本文提出的恶意软件检测方法可达到较高的检测精度以及较低的误报率, 分别为 98.50%及 3.18%。

关键词 恶意检测; 动态检测; 数据依赖; 底层特征
中图法分类号 TP309.5 DOI 号 10.19363/J.cnki.cn10-1380/tn.2020.07.08

Malware Detection Method Based on Low-level Data Flow Analysis

HAN Jinrong[1,2], ZHANG Yuantong[1,2], ZHU Ziyuan[1,2], MENG Dan[1,2]

[1] The 5th Lab, 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

Abstract In recent years, the growing number of malicious software poses a serious threat to system security and has caused huge economic losses. Researchers have proposed many malicious software detection schemes, but due to the presence of packers and polymorphism techniques, traditional detection solutions such as static feature matching are inadequate. In addition, the traditional application layer dynamic detection method also has the disadvantage of being easily disabled or bypassed by malware. This paper proposes an approach for malicious software detection by using the low-level data flow relationship, which monitors the data transmission of the benign program and the malicious program at the low-level, then generates the data dependence graph, extracts feature from the data flow graph to form feature vector, uses the feature vector to measure the similarity of data flow graphs, and evaluates the malicious tendencies of software behavior to achieve the goal of quickly detecting malicious software. The method has the characteristics of low complexity and high detection efficiency. The experimental results show that the malware detection method proposed in this paper can achieve higher precision and lower false positive rate, which are 98.50% and 3.18% respectively.

Key words malware detection; dynamic detection; data dependence; low-level features

1 引言

近年来, 日益增多的恶意软件对计算机系统安全构成了巨大的威胁, 同时也带来了巨大的经济损失。McAfee Labs [1]发布的威胁报告显示, 2017 年第三季度新恶意软件的样本数量已经上升至 5760 万, 与同年第二季度相比数量增加了 10%。庞大的恶意软件数量使得通过手动分析或逐一检查检测可疑软件变得几乎难以实现。因此, 如何快速且有效的检测恶意软件成为工业界和学术界研究的热点。

一方面, 面对快速增多的恶意软件, 安全防御技术相关的研究人员提出了诸多可用于区分恶意程序和良性程序的研究方案; 但与此同时, 攻击技术相关的研究人员则提出了逃避检测的手段来躲避相关的检查。例如, 许多用户依靠传统的杀毒软件来保护计算机系统, 传统杀毒软件主要依赖静态特征检

通讯作者: 朱子元, 博士, 研究员, Email: zhuziyuan@iie.ac.cn。
本课题得到中国科学院战略性先导专项项目(No.XDC02010400)和国家科技重大专项项目(No.2016ZX01035101)资助。
收稿日期: 2018-07-05; 修改日期; 2018-08-30; 定稿日期: 2020-06-19

测或虚拟沙箱技术识别系统中的恶意程序。但这些方式存在缺陷。首先, 对于静态特征检测手段, 攻击者可以通过打包加壳、字符串混淆等手段绕过静态字节模式匹配; 而对于沙盒技术, 恶意程序可以在执行恶意代码前, 通过一定方式判断当前的环境是否为沙盒, 再决定是否执行恶意操作, 从而绕过检测; 其次, 传统的杀毒软件依赖病毒特征数据库检测恶意程序, 但数据库的更新通常会落后于病毒变种的出现, 因此攻击者可以变异已有病毒程序, 在短时间内形成数百种病毒变种攻击特定的计算机系统。除此以外, 传统的杀毒软件也会对被保护的系统造成不可忽视的性能开销。同时, JavaScript, Flash, PowerShell, 电子邮件, URL 等存在的大量漏洞, 为恶意软件通过多种渠道危及系统提供了机会。

针对上述问题, 越来越多的学者开始研究新的具有区分性的程序特征从而更好地分辨恶意程序与良性程序, 通过构建高效快速的分类模型识别恶意软件, 弥补传统恶意软件检测技术存在的缺陷。最初, 安全研究人员利用程序中的数据, 如程序中的可疑字符串、应用程序编程接口(Application Programming Interface, API)信息、导入导出信息等数据进行特征提取。然而, 此类方法往往可以通过替换程序中的字符串、删除导入表内容、更改 API 调用顺序或使用管理员权限禁用应用层检测软件的方法逃避检测。

因此, 最近研究人员开始使用底层的运行时数据(如缓存缺失率、分支命中率等)进行特征提取。利用底层特征进行恶意检测的优势在于恶意软件作者难以绕过或阻止底层的数据收集。改变编译器对程序中数据传递关系的编译往往需要非常大的工作量, 而中止独立工作的硬件对运行数据的监视更是难以实现的。

针对上述问题, 本文提出了基于底层数据流分析的恶意软件检测方法, 即在计算机微体系结构层面收集程序运行时寄存器和内存间数据传递情况并进行分析, 利用数据传递情况的差异, 对良性程序和恶意程序进行分类。本文的主要创新点和贡献为:

(1) 本文提出了一种基于底层局部数据流分析的恶意软件检测方法, 利用程序运行时的底层数据传递模式特征进行有效恶意检测, 可达到较高的检测率及较低的误报率, 相比于使用机器学习进行恶意软件检测的方法具有较低的复杂度;

(2) 本文提出以系统调用为触发条件的底层数据流分析方法, 可有效减少待分析数据的数据量, 提高了待分析数据的信息价值;

(3) 本文将图的特征向量提取技术应用于底层数据流分析, 相对于传统的基于数据流图相似性比较的方法具有较低的计算开销。

本文的其余部分安排如下。第二章介绍了相关工作及背景; 第三章介绍了基于底层数据流关系进行恶意检测的主要方法; 第四章是实验和对实验结果的分析; 第五章是对本文方法的总结和展望。

2 相关工作

恶意软件检测通常依赖于程序的静态或动态特征, 针对不同的特征产生不同的检测方法。因此, 恶意软件的检测方法可以分为静态检测和动态检测两大类。静态检测的分析对象是程序的静态特征, 如程序的文本特征、导入导出信息以及字节码等; 动态检测的分析对象是程序运行中的动态特征, 如执行的系统调用、操作码以及访存地址等。

2.1 静态检测

早期的研究工作主要致力于利用静态检测技术对恶意软件加以甄别。例如, Bilar 等人[3]统计了良性软件和恶意软件中不同操作码出现频率的分布差异, 利用这些数据对比了程序中不同操作码的出现频率对于恶意检测的有效性。

静态检测从数据的使用角度大致可以分为基于序列的静态检测方法与基于图的静态检测方法。

基于序列的静态检测方法

基于序列的静态检测方法指利用程序可执行文件中的字符串序列、操作码序列、API 调用序列进行恶意软件及其变种的检测[2,4-7,9]。Griffin 等人[2]提出从给定恶意样本集中产生涵盖尽可能多恶意程序的字符串签名最小集合, 以达到最低误报率的恶意检测方法。文献[4]提出提取可执行文件中的二进制操作码序列, 利用 N 元模型(N-Gram)等方法从中选取有效特征, 并借助支持向量机(Support Vector Machines, SVM)构建良性软件行为模型的方法识别恶意程序。文献[5]比较了恶意软件和变种间不同长度的操作码序列频率的相似性, 以及恶意软件和良性软件间不同长度的操作码序列频率的相似性。Nagano 等人[6]对比了使用 API 导入信息和程序的汇编码序列进行恶意检测的效果。文献[7]分析了可执行文件的 API 调用序列, 利用序列匹配, 检测待测文件中 API 序列和样本数据库中 API 序列间的相似性。文献[9]对可执行文件汇编代码使用 N-Gram 构建文件特征, 利用 K 近邻计算未知文件与样本文件的相似性, 并达到了最高 91.25%的检测率。文献[10]根据汇编文件的控制流图结构, 按照一定规则提取相关

API 调用序列, 然后计算序列的相似性。文献[11]从API 调用图中提取 API 调用序列, 根据 API 所属家族或包的名称对 API 调用序列进行抽象, 使用马尔可夫链构建行为模型。

基于图的静态检测方法

由于基于序列的静态检测方法抗干扰性较差, 恶意攻击者可通过修改目标序列的顺序绕过检测。因此, 部分学者提出基于图的静态恶意软件检测方案, 利用程序中的系统调用图、API 调用图或控制流图进行相似性检测。例如, 文献[12]使用随机游走图核比较两个 API 调用图的路径是否相似。但是, 由于基于图的相似性匹配度量方法(如图同构、最大共同子图或图编辑距离)是一个 NP-完全问题, 计算时需要很大的时间开销。因此, 许多研究提出了提高图匹配效率的检测方案[13-20]。例如, 将图进行特征化, 转化到特征空间进行建模分析, 以降低时间和空间开销。

文献[13]根据系统调用行为将其划分为不同组, 根据系统调用及其调用顺序构建系统调用图, 挖掘良性样本集和恶意样本集中的最大频率子图集合, 从而根据待测程序是否存在最大频率子图集合中包含的子图可将待测程序转化为特征向量, 在向量空间进行相似性比较。文献[14]根据恶意软件中的敏感API(访问敏感信息的 API)及其敏感系数缩小函数调用图的规模, 形成敏感 API 调用子图, 从图中提取相关特征组成特征向量, 并利用机器学习方法进行分类。由于同家族恶意软件所调用的敏感 API 具有相似性, 文献[15]对提取到的敏感 API 调用子图进行了拆分, 计算局部敏感 API 调用子图中各节点间的相似性及子图相似性, 对子图进行聚类, 得到各家族频繁出现的子图特征。Shafiq 等人[16]根据 API 序列中各 API 调用的先后顺序生成 API 关系图, 然后对图中的节点级特征(如入度、出度、紧密中心性等)、子图级特征和图级特征三个层次进行特征提取, 并利用朴素贝叶斯进行分类。实验论证了图中节点级特征在恶意检测方面也可以达到较好的检测效率和较高的检测精度。文献[17]将 API 调用和系统资源作为节点提取了程序的整合 API 调用图, 使用贪心算法计算待测程序和已有程序的图编辑距离, 实现图的模糊匹配。Hu 等人[18]提取恶意软件的函数调用图, 构建图相关数据库, 使用优化的匈牙利算法进行最近邻搜索, 实现在多项式时间内求解二分图匹配问题, 得到图编辑距离估计值。

部分研究提出利用程序的控制流图计算程序间相似性[8,19,20]。文献[8,20]使用逆向工程将程序的控制流图转换为字符串, 比较字符串之间的相似性, 实现快速控制流图匹配。文献[19]使用中间语言对程序的控制流图中各指令进行注解, 通过注解形成注解控制流图, 通过匹配两个注解控制流图及其中各节点注解是否相同, 从而衡量两图相似性。

基于序列或基于图的静态检测方法的优势是可以在不执行可疑程序的前提下获取程序的执行路径信息, 从而避免了动态检测时, 程序执行未知恶意行为给系统带来的潜在威胁。但静态检测方法也存在着一定的局限性, 首先, 二进制源代码的反汇编易受干扰, 例如, 恶意程序编写者喜欢使用一些垃圾字节来扰乱 IDA Pro(Interactive Disassembler Professional Pro)等反汇编器对程序二进制源码的精确反汇编, 这也是当前恶意检测研究领域关注的问题之一。另外, 攻击者可以使用加壳和多态等静态混淆技术阻止静态分析。加壳技术指通过特殊的算法对可执行文件的资源进行压缩处理的一种手段, 程序源码在磁盘中以加密形式存在, 压缩后的文件可独立运行并在内存中完成对程序字节的解压过程。该技术通过隐藏程序真正的入口点来阻止分析者对其程序内容进行反汇编分析, 想要脱壳必须知道被加壳的程序使用的加壳算法, 否则反汇编器无法找到程序的真正入口点。许多合法的软件开发者也使用加壳算法对软件进行版权保护, 故不能根据程序是否被加壳直接判断程序是否具有恶意。繁多的加壳算法以及加壳算法的不断演进, 导致检测人员无法在加壳算法未知的情况下完成脱壳[21]。一项调查显示, 在 98801 个恶意软件样本中, 40.2%的软件无法被最常用的脱壳工具 PEID 识别[22]。多态技术则指利用特殊的加密技术随机选择加密密钥对程序主体代码进行加密从而改变程序静态特征的一种手段。多态程序每一次执行后会选择新的加密密钥对程序内容进行加密, 故程序每次执行后拥有新版本的静态特征, 仅当在内存中运行时进行解压缩及恢复运行时代码。

因此, 攻击者可以使用对抗反汇编技术扰乱对程序源码的分析, 或借助加壳或多态技术绕过静态检测 [23]。

2.2 动态检测

为了一定程度上缓解静态检测方法存在的问题, 分析人员转向于使用动态检测方法进行恶意软件检测。动态检测主要依据程序在运行中表现出的行为判断其是否为恶意程序。例如, 程序在运行期间将执行一系列 API 调用或系统调用函数从而与不同的系统资源(例如系统文件、注册表和网络通信)进行交互,

所以程序执行期间发生的相关 API 调用或系统调用成为软件执行过程中显著的行为特征。此外, 最近许多研究人员证明了利用程序运行时的底层信息如缓存缺失率、访存距离、操作码频率等也可以进行恶意软件检测。故从检测所在层的不同可以分为高层动态检测和底层动态检测。下面我们将对两个动态检测方向进行介绍。

2.2.1 高层动态检测

高层的动态检测技术指在应用层或操作系统层收集程序执行时表现出的行为事件, 通过对这些事件进行序列相似性或图相似性分析, 判断其是否为恶意程序的检测技术。

基于序列的动态检测方法

基于序列的动态检测指利用程序运行中获取到的相关系统调用、API 调用或操作码等运行时数据的序列信息进行相似性比较的一种检测方法。

同一个恶意家族的不同变种在运行期间执行的 API 调用序列具有一定相似性。Ahmed 等人[24]使用程序运行时发生的 API 调用的相关参数及返回值构建空间信息特征, 使用 API 调用序列构建时间信息特征, 通过机器学习方法进行训练、分类, 检测准确度可达到 98%。Cho 等人[25]通过匹配不同程序的 API 调用序列最长共同子序列来检测不同程序间的相似性。Bojan 等人[26]提出利用深度学习对程序运行时系统调用序列进行分析及训练, 提出的模型对多个病毒家族的平均检测精度为 85.6%。

相同恶意家族的不同变种执行的系统调用及其序列同样具有相似性, 故一些研究提出了利用系统调用序列进行动态恶意软件检测方法。GuardOL[27]使用某些关键系统调用的频率推导恶意程序的行为语义。Dahl 等人[28]在恶意软件识别过程中使用了特殊字符串、API 调用序列, 各系统调用及其输入参数三种特征, 利用集成神经网络进行分类, 错误率仅为 0.42 %。Canzanese 等人[29]使用 TF-IDF(Term Frequency-Inverse Document Frequency)计算了不同系统调用序列的频率, 利用 SVD(Singular value decomposition)、LDA(Linear discriminant analysis)进行特征集降维, 对比不同机器学习算法对系统调用序列频率特征进行分类时的实时检测效果。

然而基于序列的动态检测方法与基于序列的静态检测方法相同, 其抗干扰性较差, 恶意攻击者通过在程序中添加不相关的调用或利用等效的调用序列替换原先序列, 可绕过基于序列的动态检测方法。

基于图的动态检测方法

针对基于序列的动态检测方法存在的问题, 一些学者提出使用污点分析技术或符号执行技术追踪程序执行时的系统调用间的依赖关系, 进而从依赖关系中提取依赖关系图, 利用图匹配算法进行相似性检测[30-36]。这种基于图的动态检测方法可以缓解使用调用序列进行检测时容易被干扰的问题。

文献[30]提出了系统调用切片与等价性检查方法, 可用于识别两个程序的执行轨迹间的细粒度相似性或差异。Kolbitsch 等人[31]第一次提出利用可监控的环境记录可疑程序与系统间的交互行为并进行相似性检测的方法, 即通过程序运行时发生的系统调用及相互的依赖关系构建系统调用依赖图, 将生成的依赖图与数据库中已有的依赖图进行对比, 寻找最相似的依赖图, 从而判定被测程序是否是恶意程序。但是, 由于该方案需要一个存储大量调用依赖图的数据库, 故存储要求较高, 且搜索耗时较长。因此, 如何进行图的相似性匹配及减少图匹配时的计算复杂度成为研究热点。

Ding 等人[32]对程序运行时的各系统调用的参数依赖关系进行污点分析, 构建程序的系统调用关系依赖图, 根据不同家族的依赖图特点, 对图进行剪枝, 构建该恶意家族最大共同行为子图, 然后利用图的相似性分析对待测恶意软件进行分类。但是图的剪枝及相似性匹配需要较高的计算开销, 实际应用会给计算机带来严重的计算负载。Park 等人[33]计算不同恶意软件的系统调用行为图中的最大公共子图。在文献[34]中, 作者提出利用污点分析记录程序执行过程中发生的所有系统调用, 将具有相似功能的系统调用划分为不同组, 根据系统调用组所生成的图的各节点的入度和出度相关特征, 分析计算两个图的相似性。Jazi 等人[35]提出利用模拟退火算法计算图编辑距离, 降低图相似性匹配的复杂度。文献[36]提取了系统调用图的拓扑结构特点和图中的系统调用节点属性, 如平均入度中心性、节点出入度为 1 的节点数量等特征, 使用机器学习方法构建了分类模型。

高层动态恶意检测技术在一定程度上克服了静态检测容易被绕过的问题。但是, 高层动态检测方案中, 恶意软件可以通过陷入内核从而劫持内核控制流, 以高权限禁用应用层的检测程序, 或通过检测当前运行环境来决定是否执行恶意行为的手段绕过高层动态检测。例如, 某些应用层检测软件通过沙盒运行可疑程序, 分析程序在沙盒中的行为, 从而根据行为对其分类。而当某些恶意软件检测到运行环境为沙盒时, 恶意软件可以暂时不执行恶意行为[37]。

2.2.2 底层动态检测

为了应对高层检测易被绕过的局限性, 近年来许多学者提出利用系统底层特征进行恶意软件检测的技术。

Demme 等人[38]证明了可以利用机器学习对执行期间从 ARM 性能计数器收集的数据进行有效的恶意软件分类。Tang 等人[39]利用硬件性能计数器(Hardware Performance Counter, HPC)周期性地从系统底层收集加载指令数、存储指令数、分支指令数、返回指令数及缓存缺失数等数据, 形成特征向量, 使用无监督机器学习, 构建良性程序的基准模型, 可检测被恶意软件感染的程序在执行中发生的偏差。

Patel 等人[40]在 Linux 上利用 HPC 周期性地收集 cache 缺失数及分支数等运行时数据, 使用机器学习进行建模及训练, 得到最好精确度为 89.62%。Xu 等人[41]利用程序执行期间的访存地址数据检测良性程序是否被恶意软件感染。Ozsoy 等人[42]统计了程序执行期间执行的每条指令的类型、相关操作码的频率、内存读写频率、访存地址频率和分支命中情况等数据, 使用逻辑回归和神经网络模型进行训练, 并在 X86 处理器上进行了仿真。

但是, 目前通过底层进行数据收集的方法存在的缺点是, 由于周期性的数据采集策略具有较大的随机性, 故采样数据的有效性和可靠性难以保证。如果通过采用提高采样频率的方式弥补这一缺点, 则会造成待分析数据量的提升, 增加了系统的运行开销。另外, 机器学习以及深度学习的建模方法复杂度较高, 后续实现成本较高。因此, 本文提出利用底层局部数据流关系进行动态恶意软件检测的方法, 选取局部数据进行分析, 降低采样数据量, 弥补传统的底层动态检测方法在数据采样量方面的不足, 同时降低模型复杂度。

3 基于底层数据流分析的动态恶意软件检测方法

本文中, 底层数据流的定义是: 处理器微体系结构中寄存器之间, 或寄存器与内存之间数据传递情况。

经过分析, 功能相似的程序通常具有相似的底层数据流, 功能差异大的程序其底层数据流的差异亦较大。例如, 同一家族的不同恶意软件变种, 执行过程中在系统目录下创建新文件, 并替换原有文件, 故二者数据流具有明显相似性。相比之下, 一个执行系统关键位置文件替换的恶意软件与常用的办公软件由于执行行为不同, 故数据流具有较大差异性。因此, 底层数据流的相似度可以作为程序功能差异性比较的依据。本文重点关注待测程序的底层数据的传递模式与已有恶意程序样本的底层数据传递模式是否具有相似性, 即二者的数据流间的相似性比较, 而不关注于特定的数值或变量是否具有相似性。

通常, 研究人员使用数据流图匹配对数据流的相似性进行刻画。因此, 本文通过构建底层数据的数据流图(Data Flow Graph, DFG), 以实现对不同数据流的相似度比较。数据流图是描述程序运行时数据传递路径的一种基本模型, 可行的表示方式为 $DFG=(V,E)$。其中, V 表示由指令构成的数据流图顶点集, 边集 $E\subseteq V\times V$ 表示指令间的读写关系, e_{ij} 表示顶点 v_j 读了 v_i 写的内容, 即 $v_i\rightarrow v_j$。

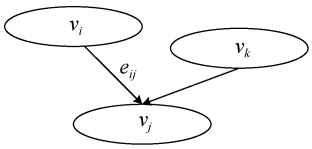

图 1 数据流图
Figure 1 Data flow graph

通过如上定义的数据流图, 可以对程序运行时的底层数据流进行分析。例如, 同一家族的两个恶意程序 W32/HLL.ow.20480a 与 W32/HLL.ow.24590 在执行中修改同目录下其他可执行文件, 导致某些程序不可执行。在这个过程中, 两个恶意程序会读取同目录下的其它可执行文件, 并覆盖其内容。图 2(a)(b)为上述两个恶意软件在执行系统调用 NtCreateFile 时的局部数据流图, 其局部数据流图显示, 两个图的拓扑结构、图中各节点数、各节点的出度、入度等信息均具有明显相似性。但 VirusTotal 扫描显示, 某商用扫描引擎甚至没有检测出 W32/HLL.ow.24590 是恶意软件。因此, 本文尝试寻找一种可行的图相似性匹配算法, 计算不同程序中的数据流图的相似性, 从而判断待测程序与已有样本集中的恶意程序及良性程序间的相似度。

由于图的相似性匹配为 NP-完全问题, 为了降低图的相似性比较的计算复杂度, 提高检测效率, 本文通过提取数据流图的特征向量对数据流图进行量化, 利用特征向量间的比较代替图的相似性匹配, 降低了运算开销。

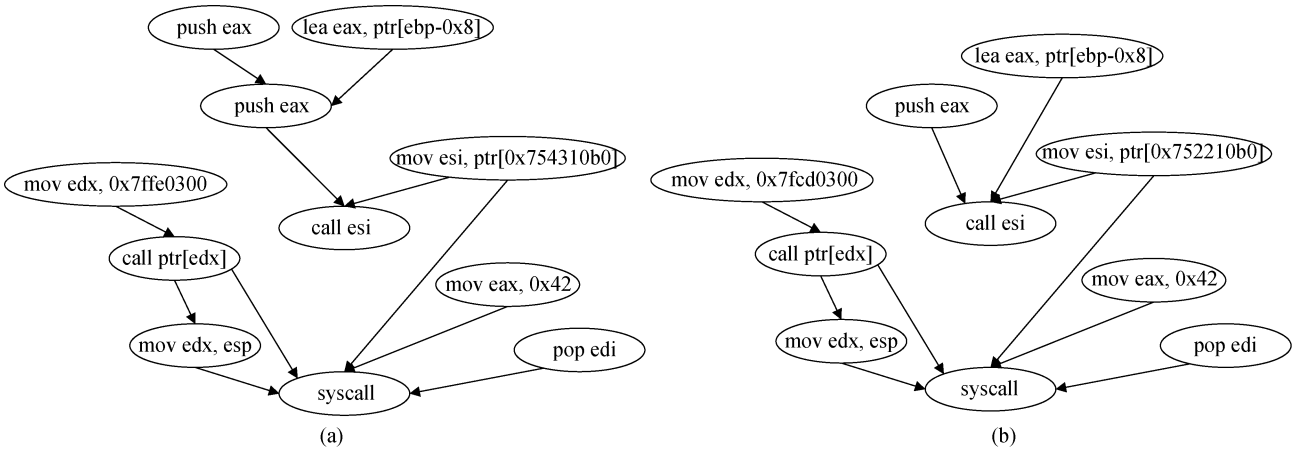

图 2　同一家族的两个恶意程序的局部数据流图对比

Figure 2　Comparison of local data flow graphs of two malicious programs of the same family

本文提出的基于底层数据流关系的恶意软件检测方法整体框架如图 3 所示, 主要包括两个阶段: “(a) 带权特征向量集生成阶段”与“(b) 恶意软件检测阶段”。共包括四个关键模块: 模块 1 用于局部数据流图构建, 模块 2 用于数据流图特征向量提取, 模块 3 用于带权特征向量集的生成, 模块 4 用于待测程序与带权特征向量集中的特征向量相似性匹配。其中, 模块 1 和模块 2 是带权特征向量集生成和恶意软件检测两个部分的共用模块。

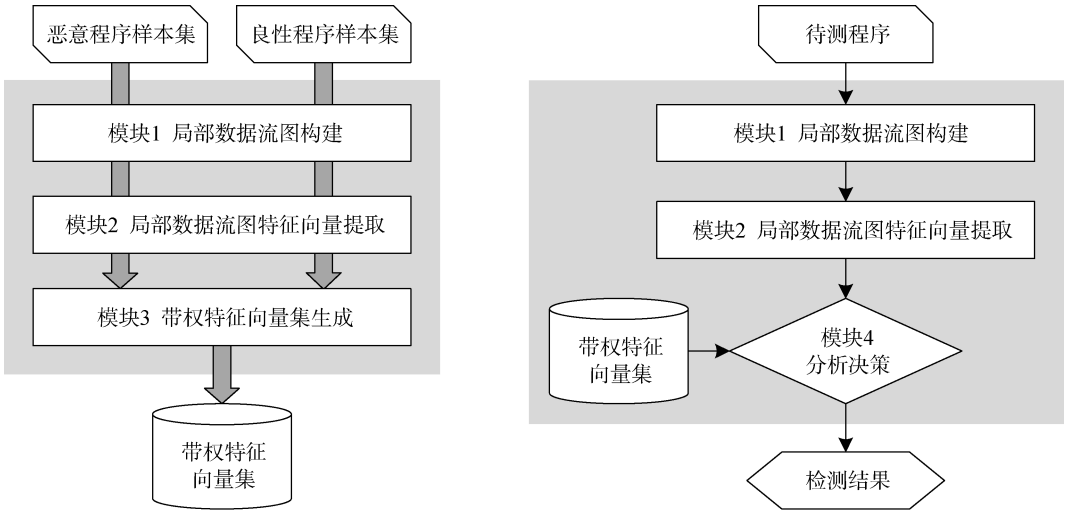

图 3　基于底层数据流分析的恶意软件检测步骤

Figure 3　The steps of the malware detection based on low-level data flow analysis

(a) 带权特征向量集生成阶段: 通过运行已有程序样本中的恶意可执行文件与良性可执行文件, 利用模块 1 分析其局部数据流关系, 构建相应的局部数据流图, 利用模块 2 提取运行程序的局部数据流图的特征向量, 在模块 3 中, 分析良性程序和恶意程序在运行过程中生成的所有局部数据流图特征向量集合, 结合特征筛选方法构建已有程序样本的带权特征向量集。该带权特征向量集用于与待测程序的特征向量进行匹配。

(b) 恶意软件检测阶段: 运行待测程序, 在运行过程中利用模块 1 构建待测程序相应的局部数据流图, 利用模块 2 提取待测程序的各局部数据流图的特征向量, 在模块 4 中利用提取到的待测程序的特征向量与阶段(a)中生成的带权特征向量集中的特征向量进行相似性匹配, 根据特征向量的匹配结果实时计算程序运行时的局部数据流的恶意倾向估值, 从而判断待测程序在执行中是否存在疑似的恶意行为。

下面我们将通过详细介绍各个模块的关键技术, 阐述本文提出的检测方法。

3.1 局部数据流图构建

局部数据流图的构建包括两个步骤: 底层待分析数据的筛选与局部数据流图的生成。

3.1.1 底层待分析数据的筛选

本文利用程序运行时的底层数据生成程序的数据流图。但是, 程序运行时的底层数据量非常庞大, 记录分析所有的数据会给系统带来严重的计算及时间开销。因此, 对底层数据进行有效筛选可以降低系统负载, 提高计算效率。

基于以上原因, 本文选取更加重要的数据(即系统调用发生前的部分数据传递关系)进行分析以达到保证整个系统的运行效率的目的。原因如下:

系统调用是应用程序(低特权级)访问系统资源(高特权级)的接口。通常, 出于安全考虑, 处理器具有用户模式和内核模式两种状态, 会对系统资源造成直接影响的指令均被限制在内核模式下执行, 否则处理器将产生异常进而终止当前进程。当用户模式的进程想要与系统进行交互时, 需要调用系统调用使得 CPU 从用户模式切换到内核模式, 此时系统内核将根据传递进来的系统调用号调用相应的处理函数, 最终将执行结果返回给发起调用的进程。一般情况下, 恶意程序想要进行提权、进程切换、网络通信等非法操作, 需要通过系统调用执行相关行为达到目的。例如, 某后门恶意软件启动时调用 NtCreateProcess 创建新的进程, 该系统调用的参数包含了拟创建进程的相关信息。我们追踪参数的来源, 发现参数来源于从远处服务器接收的相关命令字符串。即远程攻击者可以通过网络发送一个数据包从而启动受害系统上的某可执行文件。

由于程序执行系统调用之前, 会构造相应的调用参数以表明某些行为需求, 故系统调用相关参数的数据传递特征蕴含了不同的行为特征, 具有重要分析价值。本文选取程序运行中执行系统调用之前的局部数据进行样本采集, 分析系统调用相关参数的数据传递模式, 从而达到减少底层待分析数据的目的。如表 1 所示, 我们从现有样本中随机抽取 6 个程序, 统计对比其在筛选前后的底层待分析数据的数据量。从表中可以看出, 筛选后的局部数据量为原数据量的 5%以下。故使用局部数据的筛选方法可有效减少底层待分析数据量。

因此, 本文以系统调用为触发点, 抓取发生系统调用前的局部底层数据流, 降低数据采集的时间开销和数据处理的计算开销。

表 1 筛选前后的数据量对比

Table 1 Comparison of the amount of data before and after screening

序号	原数据量 (单位: 条指令)	筛选后数据量 (单位: 条指令)
程序 1	15000000	177700
程序 2	15000000	91500
程序 3	4692830	48300
程序 4	5454225	52700
程序 5	15000000	170400
程序 6	15000000	475200

3.1.2 局部数据流图的生成

利用 3.1.1 所述方法, 我们可以获取一个程序运行时的底层局部行为数据记录(假设该记录由 $\sum$ 表示)。$\sum$ 包含了该程序运行过程中在系统调用发生前的局部指令内容及各指令对寄存器及内存相关的读写信息。在传统基于序列的动态恶意检测中, 许多学者利用局部数据直接进行序列的相似性比较, 这种方法存在被攻击者在序列中加入无关项(如 NOP 指令等)绕过检测的可能。由于无关项通常不会对原有数据流产生修改, 故本文根据一个程序运行时的局部数据行为记录 $\sum$ 中的各指令对寄存器及内存的读写记录, 分析各指令间的数据传递关系, 建立指令间的联系, 生成该程序的局部数据流图。局部数据流图生成的算法如算法 1 所示。

算法 1. 局部数据流图的生成.

输入: 系统调用发生时的指令计数器的值 *syscall_ip*; 系统调用发生前的程序对寄存器和内存读写关系及相关指令的一系列记录 $\sum$

输出: 依据 $\sum$ 中的记录生成的局部数据流图 G

1: *node* ← NULL
2: *ip* ← 0
3: *rlist* ← ø
4: *wlist* ← ø
5: *readlist* ← {‘eax’: {*syscall_ip*}, ‘ebx’: {*syscall_ip*}, ‘ecx’: {*syscall_ip*}, ‘edx’: {*syscall_ip*}, ‘esi’: {*syscall_ip*}, ‘edi’: {*syscall_ip*}, ‘esp’: {*syscall_ip*}, ‘ebp’: {*syscall_ip*}}
6:
7: REPEAT
8: *record* ← 从 $\sum$ 倒序读取下一条记录
9: IF *record* 是指令信息记录 THEN
10: IF *wlist* $\wedge$ *readlist.keys* $=\varnothing$ THEN
11: CONTINUE

```
12:       END
13:
14:       ip = get_ip(record)
15:       node ← createNode(ip)
16:       将 node 加入 G
17:       FOREACH  w ∈ wlist  DO
18:         IF  w ∉ readlist  THEN
19:           CONTINUE
20:         END
21:         FOREACH  rd ∈ readlist[w]  DO
22:           reader = G.getNode(rd)
23:           reader.addWriter(node)
24:         从 readlist 删除 w
25:       END
26:
27:       FOREACH  r ∈ rlist  DO
28:         将 node 加入 readlist[r]
29:       END
30:     END
31:
32:     清空 rlist 和 wlist
33:
34:     ELSE IF record 是读写信息记录 THEN
35:       IF record 是读记录 THEN
36:         将读的寄存器或内存地址放入 rlist
37:       ELSE IF record 是写信息记录 THEN
38:         将写的寄存器或内存地址放入 wlist
39:     END
40:   UNTIL  ∑ 中的记录全部被处理
41:
42:   OUTPUT G
```

算法 1 采取倒序追踪的方法构建局部数据流图, 即从处理触发的系统调用指令开始, 递归的寻找当前指令读目标的写指令。

算法的输入包括发生系统调用时的 IP(即处理器的指令计数器)值 *syscall_ip* 以及此时缓冲区内局部数据行为记录 ∑。输出即产生的局部数据流图 *G*。

算法存在的几个关键变量如下:

(1) *readlist*: 存储读依赖未被满足的指令节点

(2) *rlist*: 暂存当前指令的读信息记录

(3) *wlist*: 暂存当前指令的写信息记录

算法 1 首先从 ∑ 中倒序读入一条记录, 并判断记录是指令信息还是读写信息, 分别对应算法伪代码的第 9 行和第 34 行。伪代码的第 35 行至 39 行将读记录和写记录分别暂存到 *rlist* 和 *wlist*, 等待处理。当遇到一条指令信息记录时, 由于是倒序读取记录, 此时 *rlist* 和 *wlist* 中暂存的读写信息记录即对应着该指令信息记录中描述的指令。伪代码第 10 至 12 行检查该指令的写操作是否能满足 *readlist* 中的读依赖, 如果不能, 则该指令和后续指令没有数据传递, 丢弃该节点。当能满足 *readlist* 中某一条或多条依赖时, 创建该节点 *node* 并加入数据流图 *G* 中, 对应伪代码第 14 至 16 行。第 17 至 25 行依据当前指令的写记录数据, 构建当前指令与能够被满足的读指令节点间的数据流边, 能够被满足的读依赖被从 *readlist* 中删除。第 27 至 29 行将当前指令的读依赖存入 *readlist*。如此迭代, 直至读依赖全部被满足或记录处理完毕。

3.2 局部数据流图特征向量提取

判断待测程序运行过程中生成的数据流图是否具有恶意倾向时, 需要衡量待测程序的数据流图与已知恶意或良性数据流图之间的相似性。

图的相似性比较大多数都被认为是 NP-完全问题, 例如, 判断图是否同构、寻找两个图的最大公共子图或判断子图是否同构。此类方法需要不断遍历两个图中的各节点, 利用回溯和剪枝等手段判断两个图中各节点间的映射关系是否相同, 其计算开销与图的规模正相关。而利用底层数据流进行恶意软件检测时, 我们需要降低计算复杂度, 提高检测效率。因此, 在底层数据流的相似性匹配问题上, 我们将图的特征向量提取技术应用于恶意软件的底层数据流检测, 将数据流图的相似性匹配转换到特征空间进行, 利用特征向量标识图的特征, 从而提高底层数据流图的相似度匹配效率。

图的特征通常可以利用图中的节点及其出入度进行衡量, 我们结合常用的有向图结构衡量指标, 选取如下易于统计计算的指标表示数据流图的特征:

总节点数 n**:** 图 DFG 中的总节点数。

$$n = |V|$$

平均度 δ_i**:** 图 DFG 中的各节点的平均度。

$$\delta_i = \frac{\sum_{i=1}^{n} \left| \bigcup_{\forall k=i} e_{jk} \right|}{n}$$

终端点数 γ**:** 图 DFG 中所有终端节点(入度为 0 的节点)的个数。

平凡节点数 ε**:** 图 DFG 中所有出度与入度均为 1 的节点数。

最大入度 $\max_{in\_\deg ree}$**:** 图 DFG 中所有节点的最大入度值。

最大出度 $\max_{out\_\deg ree}$**:**图 DFG 中所有节点的最大出度值。

组合上述介绍的局部数据流图 DFG 的各属性, 可构造出局部数据流图 DFG 对应的特征向量 f_{vector}, f_{vector} 定义如下:

$$f_{\text{vector}} = \left(n, \delta_i, \gamma, \varepsilon, \max_{in\_\text{deg}ree}, \max_{out\_\text{deg}ree}\right)$$

通过对比待测程序的数据流图特征向量 f_{vector} 和已有程序样本生成的数据流图的特征向量, 我们可以快速衡量待测程序的局部数据流图与已有程序的局部数据流图的相似性。若待测程序的局部数据流图生成的特征向量与已有恶意程序的局部数据流图生成的特征向量具有相似性, 则认为待测程序的局部数据传递关系与已有恶意程序的局部数据传递关系具有相似性, 即待测程序发生了与已知恶意程序相似的数据传递过程。

3.3 带权特征向量集生成

我们利用前文所述的特征提取方法对被执行程序进行特征提取, 得到被执行程序的所有局部数据流图的特征向量。程序的数据流表明了程序执行中数据的传递模式。程序运行中发生的一些数据传递模式可以反映程序的恶意或良性特征; 反之, 另一些数据传递模式在恶意或良性程序中并不具有明显的区分性, 此类数据传递模式产生的特征向量对分析精度没有帮助, 甚至存在被攻击者作为对抗检测的手段。为排除这些特征向量对结果的影响, 我们需要选取具有区分度的特征向量, 组成带权特征向量集, 从而利用带权特征向量集对被执行程序的数据传递模式进行检测。

本文利用信息增益方法对有效特征向量进行选取。信息增益的目标是选取一些较好的特征, 使得分类时的信息量尽可能的减小, 即降低分类的不确定性[43]。信息增益的计算公式如下:

$$\begin{aligned} IG(x_i) &= H(C) - H(C/x_i) \\ &= -\sum_{i=1}^{n} p(C_i)\log_2 p(C_i) + p(x_i) \\ &\quad \sum_{i=1}^{n} p(C_i/x_i)\log_2 p(C_i/x_i) \\ &\quad + p(\overline{x}_i)\sum_{i=1}^{n} p(C_i/\overline{x}_i)\log_2 p(C_i/\overline{x}_i) \end{aligned}$$

在上述公式中, C 表示分类的类别, 取值范围为 $C_1, C_2, \cdots, C_n$, n 的取值范围越大, 说明可选的分类数越多, 分类问题具有的信息量越大。$H(C)$ 表示信息熵, 表示分类问题的总信息量。$H(C/x_i)$ 表示条件熵, 表示当特征 x_i 出现或不出现时, 分类问题具有的信息量。$p(x_i)$ 表示特征 x_i 出现的概率, $p(\overline{x}_i)$ 表示特征 x_i 没有出现的概率。

$IG(x_i)$ 表示信息熵与条件熵的差值, 代表特征 x_i 对于分类的增益值, 即对分类做出的贡献。$IG(x_i)$ 越大, 表明特征 x_i 在某类别 C_i 中频繁出现, 而在其他类别中几乎没有出现, 因此, 特征 x_i 可较好地区分 C_i 与其他类。

本文利用信息增益筛选有效的特征向量并生成带权特征向量集的过程如图 4 所示, 通过运行已有程序集 P 中的各良性程序和恶意程序, 记录各程序发生系统调用时的相关数据, 生成各系统调用附近的局部数据流图并提取相应的特征向量 f_i, 得到已有程序集 P 的特征向量集 P_f。对于所有的待考察特征向量 $f_i \in P_f$, 我们分别计算进行分类时使用与不使用特征向量 f_i 时, 有多少程序可以被正确分类, 即特征向量 f_i 对分类结果不确定性的降低程度, 计算二者的差值即为特征向量 f_i 对于分类问题的增益值 $IG(f_i)$。在本文中, 特征向量 f_i 的信息增益值 $IG(f_i)$ 代表了特征向量 f_i 可正确区分的程序数量占总数量的比重, 其值越大, 表明特征向量 f_i 对于恶意或良性行为越具有区分性。因此, 我们根据信息增益筛选出 P_f 中 $IG(f_i)$ 值较大的特征向量 f_i 组成 P_f', f_i 对应的信息增益值 $IG(f_i)$ 作为特征向量 f_i 的权重值, 共同组成带权重的特征向量集 $F = \left\{ \left\langle f_i, IG(f_i) \right\rangle \middle| i \in \mathbb{N}, f_i \in P_f' \right\}$。其中, 在选取信息增益值 $IG(f_i)$ 较大的特征向量 f_i 组成带权特征集 F 时, 特征向量选取的数量与检测精度存在直接关系, 选取的特征向量的数量过多将增加本方法实现的开销, 而选取的特征向量的数量过少则会导致检测精度偏低。因此, 带权特征向量集 F 的大小 s 为检测系统的重要参数。

同时, 为区分良性特征向量和恶意特征向量, 并便于恶意评分的计算, 我们将带权重的向量特征集 F 中良性程序中出现概率高的特征向量的信息增益值设为正值, 在恶意程序中出现概率高的特征向量信息增益值设为负值。从而根据权重值的正负可以判断待测程序的相关特征向量所具有的恶意或良性倾向性。

通过上述方法, 我们得到了对现有程序集 P 中的良性软件和恶意软件具有区分性的数据流关系组成的带权特征向量集 F。

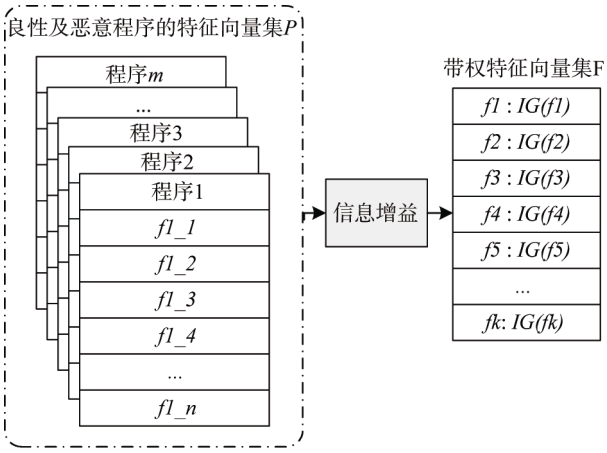

图 4　带权重的特征向量集的生成

Figure 4　Generation of weighted feature vector sets

3.4　分析决策

分析决策过程是通过匹配待测程序运行中生成的数据流图的特征向量与带权特征向量集 F 中的特征向量相似度而进行检测的一个过程。如图 5 所示, 恶意软件检测过程主要分为三步:

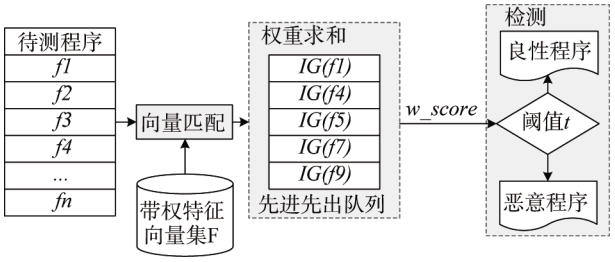

图 5　恶意软件检测过程

Figure 5　Malware detection process

(1) 特征向量匹配: 余弦相似度常被用于衡量两个向量差异的大小。因此我们使用余弦相似度衡量待测程序运行中产生的特征向量与带权特征向量集 F 中的特征向量的相似度。例如, 当待测程序产生的特征向量 f_i 与带权特征向量集 F 中的带权特征向量 f_j 间的余弦相似度的值接近 1 时, 表明两个特征向量相似。则将 f_j 相应权重值 $IG(f_j)$ 作为待测程序特征向量 f_i 的分值 f_i_score:

$$f_i\_score=\begin{cases}IG(f_j), & f_i\approx f_j\\ 0, & f_i\neq f_j\end{cases}\quad \text{其中，}\ f_j\in F$$

(2) 权重求和: 为实现对程序的连续性监控, 本文利用先进先出(First Input First Output, FIFO)队列结构记录程序运行时带权特征向量集 F 中的特征向量与待测程序特征向量匹配成功的相应权重值。如图 5 所示, 队列初始为空, 假设当前待测程序的运行中产生的特征向量 f_1 与带权特征向量集 F 中的特征向量匹配成功, 则在队列中插入匹配成功的特征向量 f_1 对应的 $IG(f_1)$ 值, 然后待测程序运行时产生的 f_2 与带权特征向量集匹配失败, 则忽略 f_2。依次类推, 每次更新时, 在队列尾部插入匹配成功的特征向量 f_j 对应的 $IG(f_j)$ 值, 如果更新后队列被填满, 则计算队列中权重值之和。假设当前 f_1,f_4,f_5,f_7,f_9 与带权特征向量集 F 匹配成功, 队列已满, 则计算队列中所有特征向量权重值之和 w_score。若再次插入新值时, 则先删除队列头部的数据。先进先出队列在逻辑上是一个滑动窗口模型, 队列的长度为滑动窗口的大小, 窗口内权重值之和 w_score 计算如下, 其中, m 为滑动窗口的长度:

$$w\_score=\sum_{i=1}^{m}f_i\_score$$

对于滑动窗口大小 m 而言, 窗口过大会造成计算负载的增加, 过小容易因为使分析结果收到特殊特征的干扰。因此窗口大小 m 为检测系统的重要参数。

(3) 检测: 根据窗口内权重分值求和结果 w_score 与阈值 τ 间的关系判定待测程序是否为恶意程序, 当窗口内的各特征向量的权重值之和大于阈值 τ 时, 我们认为程序行为趋于良性, 程序正常运行; 反之, 当窗口内的特征向量权重之和小于阈值 τ 时, 我们认为待测程序的多个数据传递关系与已知恶意数据集中的恶意程序的数据传递关系相似, 故判定其为恶意程序, 终止其运行。

$$label=\begin{cases}1, & w\_score<\tau\\ 0, & w\_score\geqslant\tau\end{cases}$$

其中, $label$ 为检测结果, 我们定义检测结果 $label$ 为 1 时, 代表检测结果为恶意程序, 否则为良性程序。

综上, 本文以“局部数据流图构建”、“局部数据流图特征向量提取”、“带权特征向量集生成”、“分析决策”为主要内容, 阐述了基于底层数据流分析的恶意软件检测方法的思路和关键技术。

4　实验及评估

本章节通过构建一个自动化数据采样与分析平台环境, 对基于底层数据流分析的恶意软件检测方法的有效性进行了验证。下面主要对实验中使用的数据集、平台架构、程序采集方法、实验结果及数据对比等进行描述。

4.1 数据集

实验中使用的样本数据集分为恶意样本和良性样本两个部分。恶意数据集来源于 2010 年的恶意样本数据库 VX Heaven[44]与 VirusShare[45]中 2017~2018 年间被创建的新恶意软件样本, 共计 1288 个 Windows 恶意软件, 使用 VirusTotal 对所有恶意样本进行归类, 样本包括蠕虫、病毒、后门程序、特洛伊木马及恶意下载等类别的程序, 各类程序对应的数量如表 2 所示。

表 2 实验中使用的恶意软件数据集

Table 2 The malware data set used in the experiment

恶意软件类型	数量
Virus	143
Backdoor	279
Trojan -SPY	310
Trojan-Downloads	224
Trojan-Banker	132
Worm	136
HackTool	64

对于良性程序, 我们收集了常用办公软件、文本编辑器、SPEC2006 基准程序、小游戏、操作系统自带的应用程序, 如系统目录下的可执行程序以及媒体播放器等, 共计 625 个良性样本程序, 使用 VirusTotal 对所有良性程序进行扫描验证, 以验证其不具有恶意行为。

4.2 平台搭建与数据采集

利用如章节 4.1 所示数据集, 我们进行了程序执行时的平台搭建及底层数据采集工作。

平台搭建方面, 我们在 VMWare 虚拟机中搭建了如图 6 所示的程序运行时相关底层数据收集平台, 操作系统选择目前常用的 32 位 Windows 7 操作系统。此外, 为避免系统安全工具对运行程序的影响, 我们关闭了防火墙和所有安全服务。

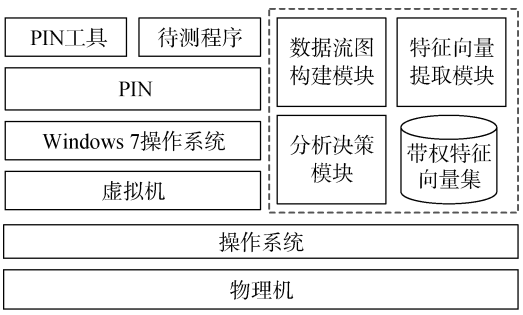

图 6 实验平台架构

Figure 6 The architecture of experimental platform

我们使用 PIN[46]进行程序运行时的数据采集工作。PIN 是一个动态二进制插桩框架, 它可以对二进制程序执行即时编译, 实现在不重编译源代码的情况下, 对二进制程序动态加入插桩代码以分析程序行为的功能。我们基于 PIN 框架编写了用于采集程序运行时各指令的信息、程序运行中各指令对寄存器和内存的读写数据信息的 PIN 工具。我们利用该工具动态采集程序运行时处理器寄存器与寄存器之间以及寄存器与内存之间的底层数据传递关系。如图 7 所示, 采集的数据在逻辑上可根据程序运行时各条指令分成相对应的一系列指令信息块。每个信息块包括一行指令信息和若干行的读写信息。指令信息包括指令的虚地址以及指令对应的汇编码; 每条读写信息以冒号分隔的二元组形式出现, 描述了该指令对寄存器和内存的读写行为。冒号前的内容统一以 X_Y 的形式存在, X 的取值包括 R 或 W, 分别对应了读和写; Y 的取值包括 reg 和 mem, 分别表示寄存器和内存。冒号后的内容表示对应的目标寄存器名称或访存目标的虚地址。

```
ip: 0x77125ab1 a: push 0xf
    R_reg: esp
    R_reg: ss
    W_reg: esp
    W_mem: 0x0012f81c
ip: 0x77125ab3 a: call 0x77106918
    R_reg: eip
    R_reg: esp
    R_reg: ss
    W_reg: eip
    W_reg: esp
    W_mem: 0x0012f818
ip: 0x77106918 a: mov eax, 0x177
    W_reg: eax
ip: 0x7710691d a: mov edx, 0x7ffe0300
    W_reg: edx
ip: 0x77106922 a: call dword ptr [edx]
    R_reg: edx
    R_reg: ds
    R_reg: eip
    R_reg: esp
    R_reg: ss
    W_reg: eip
    W_reg: esp
    R_mem: 0x7ffe0300
    W_mem: 0x0012f814
ip: 0x771070b0 a: mov edx, esp
    R_reg: esp
    W_reg: edx
ip: 0x771070b2 a: sysenter
    W_reg: eip
    W_reg: eflags
@ip 0x771070b2: sys call 177(0xf, 0x12f84c, 0xa0, 0x12f84c, 0xa0, 0x15e1254)
Success: errno=0
```

图 7 使用 PIN 采集到的底层数据

Figure 7 Low-level data collected using PIN

4.3 十折交叉验证

为验证本文提出的恶意软件检测方法的效果, 我们使用 10 折交叉验证(10-fold cross-validation)方法进行了评估。10 折交叉验证是一个用于测试算法准确性的评估方法, 具体方法是将整个样本集中的所有程序随机平分为 10 份, 每次实验逐一将其中一份作为测试数据, 而其余九份作为训练数据, 对十次实验得出的检测精度等数据取平均值。此验证方法可以对算法准确性进行公平的衡量, 避免了因为训练集和测试集划分时的偶然因素对实验结果的影响。

对于本文的实验来说, 我们分别将恶意程序集和良性程序集中的样本随机平分为十份, 然后依次将恶意样本和良性样本中的一份组合, 得到十份恶意和良性程序的组合样本。在十轮验证测试中, 每轮验证过程逐一选取其中一份作为测试集, 其余的九份作为训练集, 利用本文的方法对训练集生成带权特征向量集, 再依据生成的带权特征向量集对测试集中的程序进行检测。

4.4 实验结果

本文提出的恶意检测系统存在两个重要参数, 如章节 3.3.和章节 3.4 所述, 分别是带权特征向量集大小 s 和滑动窗口长度 m, 在前文已经说明两个参数的选取会对检测结果产生的相关影响。因此, 本节首先分析两个参数对于结果的影响。

令 s 为带权特征向量集大小, m 为滑动窗口长度。$f(s, m)$ 为检测准确度, 我们计算并比较了十折交叉验证下采用不同大小的特征向量集以及不同长度的窗口情况下的平均检测准确度。如图 8 所示, 结果表明当窗口长度大于 2 时, 窗口大小对于检测准确度影响不明显。而当带权特征向量集中的特征数量为 35~45 时, 检测准确度可相对较优的稳定在 98.43%~98.85%之间。

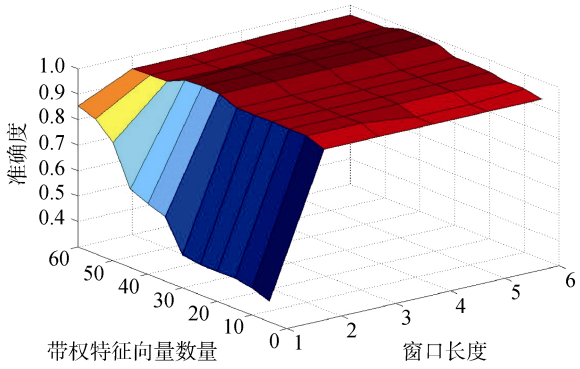

图 8 不同的带权特征向量数量及窗口长度下的平均准确度

Figure 8 The average accuracy with different numbers of the weighted feature vectors and window sizes

我们采用以下指标衡量检测结果: 真正率(True Positive Rate, TPR), 误报率(False Positive Rate, FPR), 精度(Precision), 准确度(Accuracy, ACC)。我们定义对于恶意软件被正确识别为恶意软件为 TP (True Positive), 良性软件被正确识别为良性软件为 TN(True Negative), 良性软件被错误识别为恶意软件为 FP(False Positive), 恶意软件被错误识别为良性软件为 FN(False Negative)。因此, 正确率、准确度、精度及误报率定义如下:

$$TPR = TP/(TP + FN)$$

$$\mathrm{ACC} = (TP + TN)/(TP + FN + TN + FP)$$

$$\mathrm{Precision} = TP/(TP + FP)$$

$$FPR = FP/(TN + FP)$$

表 3 为在十折交叉验证下使用上述指标进行评估的检测结果。其中, 滑动窗口长度设置为 4, 带权特征向量集大小设置为 40。

表 3 恶意软件检测结果

Table 3 Malware detect report

衡量指标	检测结果
平均正确率	0.9984
平均准确度	0.9885
平均精度	0.9850
平均误报率	0.0318

实验结果表明, 本文提出的以程序运行时的局部数据流关系进行恶意软件检测的方法可达到较高的检测精度 98.50%, 较低的平均误报率 3.18%。

4.5 与同类研究的比较

本文提出的恶意检测方法与近几年提出的部分相关恶意检测方法的检测结果对比如表 4 所示。

表 4 检测结果对比

Table 4 The comparison of detection result

提出方法	正确率(TPR)	误报率(FPR)	准确率(ACC)	精度(Precision)	数据源
Ozsoy[42], 2016	1.0	0.09~0.16		-	Offensive Computing, 系统程序
Patel[40], 2017	-	-	0.8962	-	恶意程序来源未提及、系统程序
Ding[32], 2018	0.896	-	0.964	-	Malware.lu,
Bojan[26], 2016	0.894	-	0.894	0.856	VirusShare、Maltrieve、个人收集
Menahem[47], 2013	-	-	0.90	-	VX-heaven、系统程序
本文提出方法	0.9984	0.0318	0.9885	0.9850	Vx Heaven、VirusShare、系统程序

在模型检测效果方面, Ozsoy 等人[42]使用底层架构特征进行恶意检测, 所使用的良性样本来源于系统文件、文本编辑程序及 SPEC2006 基准程序的 476 个良性程序, 恶意样本来源于 offensive Computing 网站, 共计 1087 个恶意程序。作者从系统底层对每一万条运行时指令的相关操作码频率及访存地址距离频率等信息进行统计, 使用逻辑回归及神经网络等进行训练。虽然其结果表明在使用操作码频率作为特征时, 利用具有较高计算负载的神经网络模型进行检测可达到最高 100%的恶意识别率, 但其具有 9%的误报率。对于低计算负载的 LR 模型来说, 使用操作码频率做特征时灵敏度稍差, 但同时达到 16%的高误报率。相比之下, 本文提出方法的正确率为 99.84%, 而误报率为 3.18%, 表明本文提出方法在保证较高识别率的前提下可以有效降低误报率。

Bojan 等人[26]提出利用深度学习对程序运行时执行的系统调用序列进行训练, 方法检测准确率达到了 89.4%左右, 但深度学习模型复杂度较高。

由于许多程序存在代码重用, 故 Menahem[47]等人分析了二进制文件中的函数的创建日期, 从而推导出用于恶意软件检测的新信息特征, 实验达到了 90%的平均准确度。Ding 等人[32]对各家族的最大公共子图进行了提取。虽然准确度可以达到 96.4%, 但是, 其对恶意软件的敏感度较差, 对恶意软件的漏报率达到了 10.4%。相比之下, 本文提出的检测方法在维持较高的检测准确率的前提下, 实现对于恶意软件较低的漏报率(False Negative Rate, FNR = 1-TPR)约为 0.16%。

此外, 先前利用数据流分析进行恶意软件检测的方法主要通过追踪程序运行时的每个系统调用及其参数间的关系, 构造整个程序的系统调用关系图, 但追踪整个程序的数据传递过程会造成严重的时间损耗, 因此实用性较差。本文与此类研究的差别在于, 本文仅关注恶意程序和良性程序在运行时系统调用发生前的局部数据传递模式的差异, 量化了数据流图的恶意或良性相似度, 可以做到不追踪程序运行时全局的系统调用依赖关系, 仅记录底层局部数据流的传递模式差异, 减少了待分析数据的数据量, 降低了空间和时间的开销, 增加了本方法的实用价值。

以上研究对比进一步表明了底层数据流关系对于恶意软件检测的有效性和可用性。

5 总结

本文提出了一种基于程序运行时局部数据传递关系进行恶意软件检测的方法。这种方法通过将局部数据流图特征向量化, 达到快速分析程序行为的恶意倾向性。此外, 本文提出以系统调用为采样点进行局部数据分析, 在确保信息价值的情况下, 有效降低了待分析数据的数据量。

本文的优势在于, 基于系统底层数据流分析的恶意软件检测方法可以有效避免应用层或操作系统层检测方法存在的容易被恶意软件禁用或被绕过检测的缺点; 直接收集底层程序运行时的数据减少了恶意软件逃避检测的可能性。该方法相对于现有的底层恶意软件检测方法减少了待分析数据的数据量, 具有较低的复杂度。

在下一步研究工作中, 我们将对以下方向进行探索。首先, 我们将研究如何在硬件上集成我们的检测模型, 实现低开销的快速实时检测。其次, 我们将探索如何进一步减少对 CPU 中运行数据的采样量, 同时保持较高的检测精度, 进一步增强我们的模型健壮性。

参考文献

[1] McAfee Labs threats report. https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-dec-2017.pdf

[2] Griffin K, Schneider S, Hu X, et al. Automatic Generation of String Signatures for Malware Detection[M]. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009: 101-120.

[3] Bilar, D. Opcodes as predictor for malware[M]. Inderscience Publishers, 2007.

[4] Zolotukhin M, Hamalainen T. Detection of zero-day malware based on the analysis of opcode sequences[C]. *Consumer Communications and Networking Conference (CCNC'14)*, 2014: 386–391.

[5] Santos, I., Brezo, F., Nieves, J., et al. Idea: opcode-sequence-based malware detection[C]. *International Conference on Engineering Secure Software and Systems*, 2010: 35-43.

[6] Nagano Y, Uda R. Static Analysis with Paragraph Vector for Malware Detection[C]. *The 11th International Conference on Ubiquitous Information Management and Communication*, 2017: 256-268.

[7] J.-Y. Xu, A. H. Sung, P. Chavez, et al. Polymorphic malicious executable scanner by api sequence analysis[C]. *Hybrid Intelligent Systems (HIS'04)*, 2004: 378–383.

[8] Cesare, Silvio, Yang Xiang. Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs[C]. *Trust, Security and Privacy in Computing and Communications (TrustCom'11)*, 2011: 181–189.

[9] Santos, I., Penya, Y. K., et al. N-grams-based File Signatures for Malware Detection[C]. I*nt'l Conf. Enterprise Information Systems*

(ICEIS'09), 2009: 317–320.

[10] Iwamoto K, Wasaki K. Malware Classification Based on Extracted API Sequences Using Static Analysis[C]. Proceedings of the Asian Internet Engineeering Conference, 2012: 31-38.

[11] Mariconti, Enrico, Lucky Onwuzurike, et al. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models[C]. *Network and Distributed Systems Security Symposium (NDSS'17)*, 2017: 869-872.

[12] Dam K H T, Touili T. Malware Detection Based on Graph Classification[C]. *The 3rd International Conference on Information Systems Security and Privacy*, 2017: 455-463.

[13] Hellal, Aya, Lotfi Ben Romdhane. Maximal Frequent Sub-Graph Mining for Malware Detection[C]. *Intelligent Systems Design and Applications (ISDA'15)*, 2015: 31–39.

[14] Fan M, Liu J, Wang W, et al. DAPASA: Detecting Android Piggybacked Apps through Sensitive Subgraph Analysis[J]. *IEEE Transactions on Information Forensics and Security*, 2017, 12(8): 1772-1785.

[15] Fan M, Liu J, Luo X P, et al. Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis[J]. *IEEE Transactions on Information Forensics and Security*, 2018, 13(8): 1890-1905.

[16] Shafiq Z, Liu A. A Graph Theoretic Approach to Fast and Accurate Malware Detection[C]. *2017 IFIP Networking Conference (IFIP Networking)*, 2017: 1-9.

[17] Elhadi A A E, Maarof M A, Barry B I A, et al. Enhancing the Detection of Metamorphic Malware Using Call Graphs[J]. *Computers & Security*, 2014, 46: 62-78.

[18] Hu X, Chiueh T C, 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] Alam S, Traore I, Sogukpinar I. Annotated Control Flow Graph for Metamorphic Malware Detection[J]. *The Computer Journal*, 2015, 58(10): 2608-2621.

[20] Cesare S, Xiang Y, Zhou W L. Malwise—an Effective and Efficient Classification System for Packed and Polymorphic Malware[J]. *IEEE Transactions on Computers*, 2013, 62(6): 1193-1206.

[21] Peter Ferrie. Anti-Unpacker Tricks[C]. *International CARO Workshop*, 2008:34-58.

[22] Oberheide, M. Bailey, F. Jahanian. Poly Pack: An Automated Online Packing Service for Optimal Antivirus Evasion[C]. *Usenix Conference on Offensive Technologies (WOOT'09)*, 2009: 9-16.

[23] Moser, A., Kruegel, C., Kirda, E. Limits of Static Analysis for Malware Detection[C]. *Twenty-Third Annual Computer Security Applications Conference (ACSAC'07)*, 2007: 421–430.

[24] Ahmed F, Hameed H, Shafiq M Z, et al. Using Spatio-temporal Information in API Calls with Machine Learning Algorithms for Malware Detection[C]. *The 2nd ACM workshop on Security and artificial intelligence*, 2009: 55-62.

[25] Cho, In Kyeom, Serv, et al. Malware Similarity Analysis Using API Sequence Alignments[J]. *Internet Serv. Inf. Secur.*, 2014, 4(2): 103–114.

[26] Kolosnjaji B, Zarras A, Webster G, et al. Deep Learning for Classification of Malware System Call Sequences[M]. AI 2016: Advances in Artificial Intelligence. Cham: Springer International Publishing, 2016: 137-149.

[27] Das S, Liu Y, Zhang W, et al. Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware[J]. *IEEE Transactions on Information Forensics and Security*, 2016, 11(2): 289-302.

[28] Dahl G. E., Stokes J. W., Deng L., et al. Large-scale malware classification using random projections and neural networks[C]. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*, 2013: 3422-3426.

[29] Canzanese R., Mancoridis S., Kam M. Run-time classification of malicious processes using system call analysis[C]. *Malicious and Unwanted Software (MALWARE'15)*, 2015: 21-28.

[30] Ming Jiang, Dongpeng Xu, Yufei Jiang, et al. BinSim: Trace-Based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking[C]. *USENIX Security Symposium (USENIX Security'17)*, 2017: 253–270.

[31] Kolbitsch C, Comparetti PM, Kruegel C, et al. Effective and efficient malware detection at the end host[C]. *USENIX Security Symposium (USENIX Security'09)*, 2009: 351-66.

[32] Ding Y X, Xia X L, Chen S, et al. A Malware Detection Method Based on Family Behavior Graph[J]. *Computers & Security*, 2018, 73: 73-86.

[33] Park Y, Reeves D, Mulukutla V, et al. Fast Malware Classification by Automated Behavioral Graph Matching[C]. *The Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, 2010: 25-36.

[34] Nikolopoulos S D, Polenakis I. A Graph-based Model for Malicious Code Detection Exploiting Dependencies of System-call Groups[C]. *The 16th International Conference on Computer Systems and Technologies*, 2015: 228-235.

[35] Jazi H. H., Ghorbani A. Dynamic graph-based malware classifier[C]. *Annual Conference on Privacy, Security and Trust (PST'16)*,2016: 112–120.

[36] Jang J W, Woo J, Yun J, et al. Mal-netminer: Malware Classification Based on Social Network Analysis of Call Graph[C]. *The 23rd International Conference on World Wide Web*, 2014: 731-734.

[37] Peter Ferrie. Attacks on virtual machine emulators[C]. *In: AVAR Conference*, 2006: 365-374.

[38] Demme J., Maycock M., Schmitz J., et al. On the feasibility of online malware detection with performance counters[C]. *International Symposium on Computer Architecture (ISCA'13)*, 2013: 559-570.

[39] Tang A, Sethumadhavan S, Stolfo S J. Unsupervised Anomaly-Based Malware Detection Using Hardware Features[M]. Research in Attacks, Intrusions and Defenses. Cham: Springer International Publishing, 2014: 109-129.

[40] Patel N, Sasan A, Homayoun H. Analyzing Hardware Based Malware Detectors[C]. *The 54th Annual Design Automation Conference 2017*, 2017: 25.

[41] Xu Z X, Ray S, Subramanyan P, et al. Malware Detection Using Machine Learning Based Analysis of Virtual Memory Access Patterns[C]. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, 2017: 169-174.

[42] Ozsoy M., Khasawneh K. N., Donovick C., et al. Hardware-based malware detection using low-level architectural features[J]. *IEEE Trans. Computers*, 2016, 65(11): 3332-3344.

[43] Shannon C E. A Mathematical Theory of Communication[J]. *Bell System Technical Journal*, 1948, 27(3): 379-423.

[44] VX Heaven [Online]. Available: http://vxheaven.org/, Oct, 2016.

[45] VirusShare [Online]. Available: https://virusshare.com/, Aug, 2018.

[46] Luk, Chi-Keung, Robert S. Cohn, et al. Pin:building customized program analysis tools with dynamic instrumentation[C]. *Acm Sigplan Conference on Programming Language Design & Implementation (PLDI'05)*, 2005: 190-200.

[47] Menahem E, Shabtai A, Levhar A. POSTER: Detecting Malware through Temporal Function-based Features[C]. Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13, 2013: 1379-1382.

韩锦荣 于 2016 年在东北师范大学软件工程专业获得学士学位。现在中国科学院大学网络空间安全学院网络空间安全专业攻读硕士学位。研究领域为计算机系统安全。研究兴趣包括: 系统安全防护、恶意软件检测等。Email: hanjinrong@iie.ac.cn

张元瞳 于 2015 年在东北师范大学软件工程专业获得学士学位。现在中国科学院大学网络空间安全学院计算机系统结构专业攻读博士学位。研究领域为计算机系统安全。研究兴趣包括: 静态分析技术、内存安全。Email: zhangyuantong@iie.ac.cn

朱子元 于 2010 年在同济大学控制理论与控制工程专业获得博士学位。现任中国科学院信息工程研究所研究员。研究领域为网络空间安全、计算机系统结构。研究兴趣包括: 安全芯片技术、处理器安全技术、系统安全理论与技术。Email: zhuziyuan@iie.ac.cn

孟丹 于 1995 年在哈尔滨工业大学计算机体系结构专业获得博士学位。现任中国科学院信息工程研究所所长。研究领域包括计算机系统安全, 大数据与云计算等。研究兴趣包括: 计算机系统安全, 云计算安全等。Email: mengdan@iie.ac.cn