软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(2):378−393 [doi: 10.3724/SP.J.1001.2012.03953]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于语义的恶意代码行为特征提取及检测方法*

王 蕊[1,2+], 冯登国[1,3], 杨 轶[3], 苏璞睿[3]

[1](中国科学院 研究生院,北京 100049)

[2](信息安全国家重点实验室(中国科学院 信息工程研究所),北京 100029)

[3](中国科学院 软件研究所,北京 100190)

Semantics-Based Malware Behavior Signature Extraction and Detection Method

WANG Rui[1,2+], FENG Deng-Guo[1,3], YANG Yi[3], SU Pu-Rui[3]

[1](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

[2](State Key Laboratory of Information Security (Institute of Information Engineering, The Chinese Academy of Sciences), Beijing 100029, China)

[3](Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

+ Corresponding author: E-mail: wangrui@is.iscas.ac.cn

Wang R, Feng DG, Yang Y, Su PR. Semantics-Based malware behavior signature extraction and detection method. *Journal of Software*, 2012,23(2):378−393. http://www.jos.org.cn/1000-9825/3953.htm

Abstract: This paper proposes a semantic-based approach to malware behavioral signature extraction and detection. This approach extracts critical malware behaviors as well as dependencies among these behaviors, integrating instruction-level taint analysis and behavior-level semantics analysis. Then, it acquires anti-interference malware behavior signatures using anti-obfuscation engine to identify semantic irrelevance and semantically equivalence. Further, a prototype system based on this signature extraction and detection approach is developed and evaluated by multiple malware samples. Experimental results have demonstrated that the malware signatures extracted show good ability to anti obfuscation and the detection based on theses signatures could recognize malware variants effectively.

Key words: malware; semantics; behavior signature extraction; malware detection

摘 要: 提出一种基于语义的恶意代码行为特征提取及检测方法,通过结合指令层的污点传播分析与行为层的语义分析,提取恶意代码的关键行为及行为间的依赖关系;然后,利用抗混淆引擎识别语义无关及语义等价行为,获取具有一定抗干扰能力的恶意代码行为特征.在此基础上,实现特征提取及检测原型系统.通过对多个恶意代码样本的分析和检测,完成了对该系统的实验验证.实验结果表明,基于上述方法提取的特征具有抗干扰能力强等特点,基于此特征的检测对恶意代码具有较好的识别能力.

关键词: 恶意代码;语义;行为特征提取;恶意代码检测

中图法分类号: TP309 **文献标识码**: A

* 基金项目: 国家自然科学基金(60703076, 61073179); 国家高技术研究发展计划(863)(2007AA01Z451, 2009AA01Z435)
收稿时间: 2010-04-12; 修改时间: 2010-09-10; 定稿时间: 2010-10-11

近年来,恶意代码已经成为威胁互联网安全的主要因素之一.据赛门铁克(Symantec)统计:2008 年,Symantec 在全球范围内监测到的恶意代码样本高达 1 656 227 个,约占 2002 年~2008 年间共监测到的总数的 60%[1].仅由赛门铁克的监测数据,已可见恶意代码数量的日益庞大及其威胁的日益严重.由于技术的局限性,仍有大量恶意代码无法有效监测.恶意代码层出不穷,是恶意代码防范形势日益严峻的主要原因.

恶意代码变种的检测问题是目前恶意代码防范的难点和重点.现有的恶意代码变种在实现上可大致分为两类:一类是基于基础技术的共用,恶意代码开发人员通过重用基础模块实现变种;一类是恶意代码专门针对现有防范技术而设计开发的混淆技术.混淆技术按实现机理可分为两类[2]:一类是干扰反汇编的混淆,使反汇编无法得到正确结果,从而阻碍进一步分析;另一类是指令/控制流混淆,此类混淆技术通常采用垃圾代码插入、寄存器重分配、等价指令替换及代码变换等方式,改变代码的语法特征,隐藏其内部逻辑关系.

恶意代码检测方法可以分为基于启发式(heuristic-based)的检测和基于特征(signature-based)的检测两大类[3].基于启发式的检测方法根据预先设定的规则判断恶意代码存在的可能性,其优势在于可检测新恶意代码样本;但其规则的生成依赖于分析人员的经验,在应用中易引发高误报及漏报率,因此在实际检测系统中应用较少.基于特征的检测方法则根据由恶意代码中提取的特征进行检测,与基于启发式的检测方法相比,具有效率高、误报率低等优点,因此被广泛应用于恶意代码检测工具之中,是目前恶意代码检测的主流方法.

特征的描述能力是决定基于特征的检测方法的检测能力和检测效率的主要因素.如何更有效地提取恶意代码的本质特征,降低混淆技术的干扰,从而对恶意代码变种进行准确、有效地检测,是目前恶意代码防御的研究热点.传统的特征检测大多采用基于代码特征的检测方法,大量商用杀毒软件即使用此类方法.该方法从已有恶意代码样本中提取代码语法(syntactic)特征用于检测,此类特征通常精确匹配单一样本,恶意代码使用简单混淆方法即可绕过相应检测[4].并且,不同变种需使用不同特征进行描述,特征库数据量往往十分庞大,且在出现变种时需及时、不断地更新.此后,研究者提出了基于行为特征的检测方法[5],通过提取恶意代码执行的行为作为特征进行检测.此方法着眼于恶意代码的实际行为,因此可避免仅针对代码的混淆方法的影响.然而,基于行为的特征仍局限于语法特性,无法抵御等价行为替换等行为层混淆方法的干扰.如 Sekar 等人[6]提出了一套攻击方法,通过垃圾行为注入使攻击行为模拟正常行为序列,从而绕过以行为序列为特征的入侵检测系统.基于语义的检测方法的提出为更好地对抗混淆技术、检测变种提供了可能.通过分析混淆技术原理发现,混淆技术具有改变代码语法特征但仍保留行为语义的特点.基于语义的检测方法即利用这一特点,通过抽象语义特征实施检测来提高对变种的检测能力.目前,已有一些工作从语义角度对特征提取及检测方法进行了一定程度的改进[7−10],其中比较有代表性的是,Christodorescu 等人[8]结合语义分析提取代码特征,提高了对部分混淆技术的抗干扰能力,但其方法在等价指令替换等方面的抗干扰能力较弱.目前,其他基于语义的特征提取工作也大多基于静态代码层分析,在行为层抗混淆方面仍有待提高.

本文提出了一种基于语义的恶意代码行为特征提取及检测方法,使用经语义抽象的行为特征图描述恶意代码行为特征.基于此特征,通过计算特征值的方法对恶意代码进行检测.本方法通过指令层的动态污点传播分析,提取恶意代码的关键行为及行为之间的数据依赖关系和控制依赖关系,在此基础上,结合行为层语义分析进行抗混淆处理,从而提高了对常用混淆技术的抗干扰能力;本方法提取的特征可融合同类多个恶意代码变种的特性,从而提高对恶意代码变种的检测能力,并具备对部分通过增加或修改行为的方式而产生的恶意代码新种的检测能力;同时,在一定程度上缩减了特征库数据量.

本文的贡献主要包括如下 3 个部分:

- 建立了一种基于行为依赖图的特征描述方法.本方法从行为层次抽象特征,以行为之间的数据依赖关系和控制依赖关系结合的方式表示行为逻辑,建立行为特征图.该特征可表示一类恶意代码的行为特征,因此可提高对恶意代码变种的检测能力,同时降低了特征库规模,从而改进检测效率.
- 提出了一种基于语义的抗混淆特征提取方法,从语义无关和语义等价两方面进行抗干扰处理.该方法通过创建语义等价序列库将行为相同的序列预归为相应的语义等价序列集合,然后建立了相应的特征图构造算法和特征匹配算法,解决了等价行为替换等混淆代码的检测问题.

- 完成了原型系统开发,实现了基于语义的恶意代码行为特征提取和检测等功能,并完成了基于 SdBot, NetSky 等实例样本的实验.实验结果表明,基于本文所述方法提取的特征具有抗干扰能力强等特点,基于此特征的检测对恶意代码变种具有较好的识别能力.

本文第 1 节介绍相关工作.第 2 节概述本文特征提取及检测的基本思路.第 3 节详细介绍行为特征的提取过程和方法.第 4 节说明如何根据提取的行为特征对恶意代码进行检测.第 5 节介绍系统设计及实验结果.第 6 节进行总结.

1 相关工作

本文的主要研究内容是恶意代码行为特征提取及检测方法.为了实现行为特征提取及检测,首先要对恶意代码进行分析,在此基础上,研究恶意代码行为特征提取及描述方法,并基于提取及描述的特征,实现基于恶意代码行为特征的检测.因此,本文从恶意代码分析、恶意代码特征描述和恶意代码检测这 3 个方面介绍相关研究工作.

恶意代码分析一般分为静态分析和动态分析两种.静态分析首先对可执行程序进行反汇编,在此基础上,分析并提取代码的特征信息.静态分析可对代码进行全面的分析,无须实际执行代码,因此不会对系统产生危害;但由于所分析的代码不一定是最终执行的代码,可能消耗大量时间于无用代码.同时,静态分析对反汇编技术的依赖也导致了其局限性.恶意代码可使用各种混淆技术阻碍反汇编分析[4],一些恶意代码通过加密、压缩等方式,其完整代码只有在实际运行中才释放,导致静态分析难以得到正确结果.为了减少混淆技术的影响,国内外学者进行了一系列的研究工作,如 Christodorescu 等人[11]针对代码重排、加壳、垃圾代码插入等 3 种混淆方式提出了相应的处理方法,试图将其恢复为混淆前的代码,但未能解决寄存器重分配、等价替换等其他混淆技术的干扰问题.动态分析则是在代码执行过程中进行分析,所分析的代码即实际执行的代码;但动态分析一次执行过程只能获取单一路径行为,而一些恶意代码存在多条执行路径.针对这一问题,Moser 等人[12]提出通过建立系统快照递归探索多执行路径的方法,改善了动态分析方法的单一执行路径问题.最初的动态分析借助调试工具进行,依赖于分析者的经验,其断点等操作易被恶意代码察觉并采取反制手段.此后,国内外学者开发了一系列的动态分析工具辅助人工分析,比较有代表性的是基于虚拟机和基于硬件模拟器的方法.基于虚拟机(VMWare)的动态分析工具如 CWSandbox[13],采用挂钩 API 的方法监控程序行为.基于虚拟机的方法依然有一部分指令在主机上执行,由于执行速度改变而易被恶意代码察觉;基于硬件模拟器的分析方法如 Bayer 等人开发的 TTAnalyze[14]系统,在 QEMU[15]的基础上开发,监控恶意代码与系统交互的 API 调用序列并生成报告输出.由于静态分析和动态分析各有优缺点,因此产生了两者结合的混合分析方法.如,Kirda 等人[16]基于间谍软件(spyware)的行为特征,利用动态分析监控组件与浏览器的交互确定代码区域,然后静态分析检查该代码区域,识别系统调用信息.

恶意代码的特征描述方法是影响检测能力的重要因素.传统的恶意代码特征大多使用序列描述法.如,Kirda 等人[16]将特征描述为系统调用序列,Bailey 等人[17]用系统消息序列描述特征.序列描述法针对代码或行为的先后次序,易受代码混淆手段的干扰.常用的还有控制流程图(control flow graph,简称 CFG)描述法[18].它以代码的执行流程描述特征,但因局限于代码执行顺序,易受顺序无关操作调换等混淆方法的干扰.Christodorescu 等人[19]将病毒源代码转换为自动机描述特征,并处理了垃圾代码插入及代码顺序变换等问题,但未能解决等价代码替换的干扰问题.其他描述方法,如 Bilar[20]使用操作码频率分布等信息描述特征,此类通过频率分布来描述的特征,垃圾代码插入即可对其形成干扰.近年来,研究者开始使用依赖关系描述代码内部逻辑关系,如 Christodorescu 等人[21]使用系统调用及系统调用间的数据依赖图描述特征,动态挖掘恶意代码与正常代码的差异子图来表示恶意代码特征,但其未考虑控制依赖关系的影响,且存在比较样本的选取和差异子图的连续性问题.

恶意代码检测可分为基于启发式的检测和基于特征的检测.基于启发式的检测,如 Rootkit Revealer[22]系统,通过比较系统上层信息和取自内核的系统状态来识别隐藏的文件、进程及注册表信息.还有一些研究工作通过

监控系统特定资源来识别恶意代码,如 Strider Gatekeeper[23]系统检测恶意自启动项信息,VICE[24]检测各种恶意代码常用钩子.基于特征的检测方法是目前的主流方法,传统的基于代码特征的检测具有速度快、效率高的优点,但易受代码混淆技术的干扰.基于行为特征的检测从代码行为出发提取特征,如,Kirda 等人[16]根据间谍软件获取敏感信息,再将数据泄露的行为特点进行检测,但其方法只局限于一类恶意代码.基于语义的特征检测从语义角度抽象特征,如 Christodorescu 等人[8]通过静态分析代码,结合指令语义信息,使用三元操作符构造特征来进行检测,可有效对抗指令重排、垃圾代码插入、寄存器重分配等混淆技术的干扰.但由于是指令级提取,对行为层混淆技术的抗干扰能力较弱,且其匹配判定过程比较复杂.Kinder 等人[10]利用形式化方法,通过模型检验(model-checking)来检测恶意代码. Sathyanarayan 等人[25]同样从语义角度出发,使用关键系统调用之间的相互关系,静态分析挖掘一类代码的共同特征,采用统计学比较的方法进行检测,可对抗部分代码混淆技术的干扰.然而,对等价系统调用替换等混淆方法未能很好地解决,且存在静态分析方法的局限性.

2 基本思路

现有恶意代码变种主要基于混淆技术实现.混淆技术是恶意代码针对当前特征检测方法使用的程序变换技术.从混淆技术产生作用的层面,可将其分为代码层混淆和行为层混淆两类:代码层混淆通过变形、压缩等方式,模糊、隐藏或改变原有代码特征,从而使基于代码特征的检测失效;行为层混淆则通过垃圾行为插入、执行顺序变换及等价行为替换等方式,改变行为序列或执行流程,使基于行为序列或流程图的检测失效.

如图 1 所示,图 1(a)为 ForBot 的原始样本代码,图 1(b)为混淆代码.混淆代码在第 8 行插入了 CreateFile 垃圾调用;在第 11 行、第 28 行插入了 ReadFile 垃圾调用.需要特别说明的是,混淆代码在第 14 行打开了 log.dat 文件,但实际上,该文件以只读形式打开,因此在后期执行时,第 17 行的写操作操作返回值必定为失败.因此,该操作并不会引起操作系统的完整性变化,也是垃圾调用.在循环中,第 26 行使用 ReadFile 等价替换了初始代码中的内存赋值操作,并且在循环中将读写粒度由 68 减为 30 以增加循环次数.在原始的代码中,在第 10 行、第 14 行和第 16 行分别使用了 CreateFile,CreateFileMapping,MapViewOfFile 的行为序列将文件映射进内存后进行读操作.而在混淆代码中,则是直接在第 20 行和第 26 行采用了 CreateFile,ReadFile 序列实现同样的读操作.

```
1  HANDLE hFile; char szPath[MAX_PATH];
2  DWORD dwSize; HKEY hKey;
3  dwSize=800;
4  RegOpenKeyEx(HKEY_CURRENT_USER,
5    "Software\\Microsoft\\WAB\\WAB4\\Wab File Name",
6    0, KEY_ALL_ACCESS, &hKey);
7  if(!hKey) return true;
8  RegQueryValueEx(hKey, "", 0, 0, (unsigned char*)szPath, &dwSize);
9  RegCloseKey(hKey);
10 hFile=CreateFile((char*)szPath, GENERIC_READ, \
11     FILE_SHARE_READ, NULL, OPEN_ALWAYS, \
12     FILE_ATTRIBUTE_NORMAL, NULL);
13 char *pszBuf=NULL;
14 HANDLE hFileMap=CreateFileMapping(hFile, 0, \
15   PAGE_READONLY, 0, 0, 0);
16 pszBuf=(void*)MapViewOfFile(FILE_MAP_ALL_ACCESS,0,0,0);
17 if(!hFileMap) { CloseHandle(hFile); return true; }
18 if(!pszBuf) { CloseHandle(hFileMap); CloseHandle(hFile); return true; }
29 char szA1[300]; int ii, j=0, iLen;
20 for(iLen=0; iLen<(iNos*68); iLen+=68) {
21   for (ii=0;ii<=68;ii++) {
22     szA1[ii]=*(pszBuf+dwAdd+j+iLen); j+=2;
23   }
24   szA1[68]='\0'; j=0;
25   SendTo(pMsg->cilent, pMsg->bNotice,\
26     pMsg->sReplyTo.Str(), \
27     "Found EMail (%s).", szA1);
28 }
29 CloseHandle(hFile);
30 UnmapViewOfFile(pszBuf);
31 CloseHandle(hFileMap);
32
33
34
```

(a)

```
1  HANDLE hFile; char szPath[MAX_PATH];char szTemp[MAX_PATH];
2  DWORD dwSize; HKEY hKey;HANDLE hTemp;
3  dwSize=800;
4  RegOpenKeyEx(HKEY_CURRENT_USER,
5      "Software\\Microsoft\\WAB\\WAB4\\Wab File Name",
6      0, KEY_ALL_ACCESS, &hKey);
7  if(!hKey) return true;
8  hTemp=CreateFile("C:\\test.dat",GENERIC_READ, \
9    FILE_SHARE_READ, NULL, OPEN_ALWAYS, \
10     FILE_ATTRIBUTE_NORMAL, NULL);
11 ReadFile(hTemp,NULL,0,NULL,NULL);
12 CloseHandle(hTemp);
13 RegQueryValueEx(hKey, "", 0, 0, (unsigned char*)szPath, &dwSize);
14 hTemp=CreateFile("C:\\log.dat",GENERIC_READ, \
15     FILE_SHARE_READ, NULL, OPEN_ALWAYS, \
16     FILE_ATTRIBUTE_NORMAL, NULL);
17 WriteFile(hTemp,szPath,dwSize,&dwWriteSize,NUL);
18 CloseHandle(hTemp);
19 RegCloseKey(hKey);
20 hFile=CreateFile((char*)szPath, GENERIC_READ, \
21     FILE_SHARE_READ, NULL, OPEN_ALWAYS, \
22     FILE_ATTRIBUTE_NORMAL, NULL);
23 char *pszBuf=NULL;
24 char szA1[200]; int ii, j=0, iLen;
25 for(iLen=0; iLen<(iNos*30); iLen+=30) {
26   ReadFile(hFile,szA1,29,0,0);
27   szA1[30]='\0'; j=0;
28   ReadFile(hTemp,szTemp,MAX_PATH);
29   SendTo(pMsg->bSilent, pMsg->bNotice,\
30       pMsg->sReplyTo.Str(), \
31       "Found EMail (%s).", szA1);
32 }
33 CloseHandle(hTemp);
34 CloseHandle(hFile);
```

(b)

Fig.1 Code segment of ForBot

图 1 ForBot 源码示例

以上手段使混淆代码和原始代码在代码特征上发生了很大的变化,并打乱了行为序列,使其行为序列产生了很大的差别.由这两段代码可知,代码和行为混淆技术简便易行并且效果明显.恶意代码普遍使用各种混淆手段对代码进行处理,为恶意代码检测带来了挑战.

当前,基于特征的检测方法主要存在以下 3 个方面的问题:其一是特征匹配能力单一,无法检测同类变种,影响检测准确性;其二是现有特征多为代码特征或行为序列不能充分描述恶意代码的本质特性,易受混淆技术干扰;其三是大量变种需要不同特征进行描述,特征库需要不断更新,从而造成特征库规模庞大,对性能产生影响.以上问题的根本原因在于,用作检测的恶意代码特征未能很好地表征恶意代码的行为本质及意图.Christodorescu 等人[21]也曾在研究中指出:目前检测工具的效果不理想,主要是由恶意代码的特征描述和实际行为之间存在差距导致的.现有的基于序列、流程图的描述方法不能直接反映恶意代码的内在逻辑,而目前使用依赖图的特征描述中,也仅关注到数据依赖或只补充少部分控制依赖,对行为逻辑关系的描述能力有限.

基于以上分析,本文的目标是从恶意代码行为之间的逻辑关系出发提取特征,提高恶意代码特征的描述能力和抗混淆能力,从而提高恶意代码的检测能力,特别是对恶意代码变种的识别能力.我们的基本思路是,将指令层分析与行为层分析相结合,利用动态污点传播分析方法,提取恶意代码的关键行为及依赖关系,从语义的角度抽象出能够识别同类恶意代码变种的行为特征,使用基于行为依赖图的方法进行描述;最后,以该特征作为恶意代码检测依据,构建一套适于该特征的检测算法来提高检测能力.

系统调用是应用程序与系统交互的接口,用户空间的应用程序通过这组接口获取系统内核服务,应用程序的行为目标几乎都是通过系统调用实现的.因此,本文使用恶意代码执行的系统调用(system call)及系统调用之间的依赖关系来描述恶意代码的行为特征.由于系统调用种类、数目繁多,本文只关注关键系统调用(critical system call),即所有可能导致安全问题的系统调用[21,25].据分析,几乎所有的恶意代码为实现目标而对敏感数据的操作都与关键系统调用相关[3],因此,监控关键系统调用可以使恶意代码的行为分析更有针对性,有利于清晰地理解恶意代码的目标并提高效率.为了描述恶意代码行为之间的关系,我们使用关键系统调用之间的数据依赖关系和控制依赖关系共同创建依赖图.依赖关系能够描述行为逻辑,不受执行流变换的混淆技术干扰.与以往使用依赖关系的描述方法不同的是,除了数据依赖关系以外,我们还提取控制依赖关系.由于恶意代码可使用控制依赖影响数据依赖而逃避检测,我们提出同时考虑数据依赖和控制依赖的行为依赖图描述方法,对恶意代码的行为特征进行更为准确、全面的描述.在依赖关系的提取方面,我们在现有动态污点传播方法的基础上进行扩展,实现对关键行为的细粒度跟踪和依赖关系的准确提取.

为了对抗混淆技术的干扰,我们通过语义分析,由行为依赖图抽象出行为特征.在恶意代码变种的过程中,其核心行为是保持不变的,即其主要功能部分的代码语义是保持不变的[7].基于这一特点,我们创建了基于语义的抗混淆引擎,目标是从一个恶意代码样本中提取能够识别该类恶意代码变种的特征.针对常用混淆方法,本文通过行为层分析,从语义无关和语义等价两个方面对行为依赖图进行处理.由此形成的恶意代码行为特征图,对垃圾行为插入、等价行为替换等各类行为层混淆技术的抗干扰能力明显增强.

为了验证所提取行为特征的准确性和有效性,我们设计了一套基于该特征的检测方法对其进行验证,并完成了特征提取及检测的原型系统,系统框架如图 2 所示.

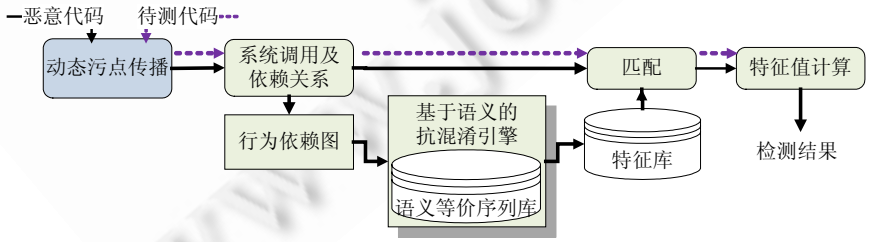

Fig.2 Framework of malware signature extraction and detection system

图 2 恶意代码特征提取及检测系统框架图

系统基本流程如下:

(1) 在可控环境中,利用可回溯的污点传播技术分析恶意代码的执行流程,识别并记录其中的关键系统调用、系统调用之间的数据依赖关系和控制依赖关系以及与污点数据相关的指令信息.

(2) 结合指令信息,根据系统调用、数据依赖关系和控制依赖关系构建恶意代码行为依赖图.

(3) 通过行为层的语义分析,对行为依赖图进行抗混淆处理,识别并处理其中的语义无关调用和语义等价调用,得到基于语义的恶意代码行为特征图.

(4) 基于所提取的特征对恶意代码进行检测.根据操作敏感度对行为特征图中的节点和边赋予相应的权值,设定加权特征值的计算方式及检测阈值;对待测代码进行污点传播分析,根据其执行的系统调用及依赖关系与特征库中特征的匹配情况计算特征值.当特征值大于检测阈值时,判定该代码为恶意代码.

3 行为特征提取

基于语义的恶意代码行为特征提取过程是在获得恶意代码样本之后,通过对其运行过程的动态跟踪,分析其行为及语义,提取行为特征.这部分主要包括 3 个方面的内容:

- 一是对目标代码的数据流分析,提取并分析关键数据和属性.此部分基于动态污点传播分析方法实现.通过可回溯的动态污点传播技术,跟踪分析代码的数据处理流程,提取行为分析相关的关键数据.
- 二是基于提取的动态数据和相关属性构造行为依赖图,结合污点传播过程和数据流分析,构造包含数据依赖关系和控制依赖关系的恶意行为依赖图.在本文中,主要是对依赖图的描述方法进行了扩展,并提出了一套基于可回溯的动态污点传播的依赖图构造方法.
- 最后,基于构造的恶意行为依赖图,通过语义抽象提取恶意行为特征,并生成相应的特征描述.在本文中,主要解决了语义无关调用和语义等价调用的混淆问题.

3.1 动态污点传播分析

为了提取恶意代码的行为特征,首先要提取描述恶意代码行为特征的关键数据.在本文中,需要提取的关键数据可分为数据元素和关系元素两类.数据元素包括恶意代码执行的关键系统调用及其参数,我们将系统调用描述为 $F=\{Ret,Parameter_{in},Parameter_{out}\}$.其中,$Ret$ 表示系统调用返回值,$Parameter_{in}$ 表示系统调用的传入参数,$Parameter_{out}$ 表示传出参数.关系元素包括数据依赖关系和控制依赖关系.数据依赖关系是由数据的定义和使用而形成的关系,对于两个系统调用 F_1,F_2,若 F_2 使用了由 F_1 定义的数据,且在 F_1 到 F_2 的执行路径中没有其他系统调用重新定义该数据,则 F_2 数据依赖于 F_1.控制依赖关系是指对于两个系统调用 F_1,F_2,若 F_2 能否被执行由 F_1 的执行状态决定,则 F_2 控制依赖于 F_1.同时,为了辅助后续分析过程,我们在提取上述关键数据的过程中提取了执行过程中与关键行为流程相关的指令信息.

本文中,这些关键数据的提取是通过对恶意代码样本进行动态污点传播分析来完成的.污点传播分析是一种有效的分析攻击行为、提取攻击信息的方法[26].在污点传播分析中,通过标记敏感数据为污点并编写污点传播规则,然后根据污点传播规则对代码执行过程中的各种系统调用和指令执行流程进行污点传播计算,构建相关行为之间的依赖关系,并记录相关数据.

本文使用可回溯的污点传播引擎进行分析.我们对 Dawn Song 的执行流回溯[25]方法进行扩展,实现污点操作流回溯.由于本文分析关注的是代码执行的关键系统调用之间的数据依赖关系和控制依赖关系,均为与污点相关的操作之间的关系,而执行流回溯包括所有执行到的指令,为了提高分析的针对性,排除无关指令的干扰,避免冗余数据量,我们根据本文的应用场景对回溯过程进行修改,针对敏感数据,在污点操作流上进行回溯.我们使用双链表链接污点传播节点,使其可以回溯污点源和污点传播流程.

污点传播分析过程包括污点标记、污点传播和污点漂白这 3 个方面.首先是标记污点数据,在分析过程中,我们将系统调用的返回值(若未作特殊说明,所指系统调用返回值均包括 in-out 类型参数返回的数据)作为污点源.我们选择性地构建了需要作为污点源监控的系统调用列表,并对其作了初步分类,主要包括文件、网络、注册表、进程这 4 类,可描述为 $Func_{sensitive}=\{FileTaint,RegistryTaint,NetworkTaint,ProcessTaint\}$.其中,每类污点源集合包含了此类需要监控并标记的系统调用,如 $File_{taint}=\{CreateFile,OpenFile,ReadFile,...\}$.当污点源集合中的系

统调用发生时,根据该污点源的类型,标记其返回值为污点.如,CreateFile 应标记返回的 Handle 为污点,ReadFile 标记读出内容的内存范围为污点.为了方便描述,我们将污点源系统调用列表中的系统调用统称为敏感系统调用.为了描述污点数据,我们构建影子内存(shadow memory)*Taint*={*Address*,*Length*,*Status*,*Type*}来保存污点数据记录.其中,*Address* 表示污点的起始地址,即在进程虚拟地址中的偏移;*Length* 表示污点数据的长度,以字节为单位; *Status* 表示污点数据的状态(执行、改写等);*Type* 表示污点的类型,分为 Regs 普通寄存器、EFLAGS 标志寄存器和 Mem 内存数据 3 种.污点数据包括两种情况:一种是内存中的数据,另一种是寄存器.对于内存中的数据,我们根据内存地址、数据的长度和当前内存状态创建影子内存.对于寄存器,同样标记影子内存.由于控制依赖关系分析的需要,我们对标志寄存器 EFLAGS 创建单独的记录.

当污点标记完成后,我们对恶意代码进行单步指令分析,通过污点传播规则计算污点传播路径,提取数据依赖关系和控制依赖关系.污点传播规则包括针对系统调用的传播规则和针对指令的传播规则.系统调用的传播规则根据不同的函数编写,对于系统调用 F={*Ret*,$Parameter_{in}$,$Parameter_{out}$},若传入参数 $Parameter_{in}\in Taint$,则将返回值 *Ret* 和传出参数 $Parameter_{out}$ 也标记为污点.指令的污点传播规则根据指令类型编写.在我们的分析中,主要关注内存操作指令、运算指令和控制流转移指令.在污点传播过程中,如果系统调用 F_1 产生的污点源 T_1 传播至 F_2 的传入参数,则 F_2 数据依赖于 F_1,记录数据依赖关系.在污点分析过程中,控制依赖表现为污点数据传播至 EFLAGS,进而影响控制流转移的过程.因此,当控制流转移指令将被执行,同时,该控制流转移指令依赖于带有污点的标志寄存器时,对当前执行的指令和其后的指令进行反汇编,通过计算后必经节点判断当前控制流转移指令的控制范围[26].如果系统调用 F_1 产生的污点源 T_1 传播至影响控制转移的指令 i,则指令 i 控制范围内的所有系统调用{$F_{i,1}$,$F_{i,2}$,…}均控制依赖于 F_1,记录控制依赖关系.

污点漂白的过程即污点被无关数据改写的过程.对于污点数据记录 *Taint*、数据 D 和路径 L,如果$\exists D,D\in L$,且 $D\notin Taint$,则污点数据 *Taint* 中以 $Address_D$ 为起点、$Length_D$ 长度的内容被漂白.在污点漂白时,对污点传播流程进行回溯,提取行为之间的依赖关系.

通过可回溯的污点传播分析,我们获取到恶意代码执行的关键系统调用、系统调用之间的数据依赖关系和控制依赖关系以及污点数据相关的指令信息,将其作为描述恶意代码行为特征的关键数据.

3.2 行为依赖图构建

构造行为依赖图的过程即描述恶意代码执行的关键系统调用之间的数据依赖和控制依赖关系的过程.根据可回溯的污点传播分析提取的恶意代码执行的关键行为数据,构建行为依赖图.

我们将行为依赖图 $G_{initial}$ 描述为 $G_{initial}$={*Entry*,*Exit*,*N*,*C*,*D*,*Code*},其中,$G_{initial}$ 表示行为依赖图,*Entry* 表示图的入口节点,*Exit* 表示出口节点,*N* 表示其他节点,*C* 表示控制依赖边,*D* 表示数据依赖边,*Code* 表示在该图执行过程中访问的指令记录.由于在污点传播分析中以敏感调用操作的返回值作为污点源,即图的入口,在分析中可能出现多个行为依赖图,我们将这些图的集合标记为 T_{init}={G_{init1},G_{init2},…,G_{initn}},$n\in N$.

行为依赖图的构造过程包括入口节点的产生、节点的添加、数据依赖边和控制依赖边的添加以及构造结束的判断.

在对恶意代码开始分析时,行为依赖图集合 T_{init} 为空.我们以产生污点的关键系统调用作为入口节点 *Entry* 生成图 $G_{initial}$,同时更新影子内存记录为当前污点的起始地址、长度、状态和类型信息,以便后续污点传播分析使用.此后对恶意代码的单步处理中,分析执行的每一条指令,计算污点传播流程.

当新系统调用发生时,解析其传入参数,如果包含污点数据,则将该系统调用作为一个新节点加入到依赖图 $G_{initial}$ 中.需要注意的是,当文件映射进内存之后发生读写指令时,我们需要将其转译为相应的系统调用节点.例如,在恶意代码的文件访问操作中,常使用 CreateFile,CreateFileMappint,MapViewOfFile 将文件映射进内存后进行读写,此时,相关的读取指令并非使用传统的 ReadFile,而是内存读写类的指令(如 memcpy).以往的分析方法在这种情况下将缺失本应记录的系统调用及依赖关系,为了解决此问题,我们采取将此类指令转译为系统调用的方法.这里的难点在于如何判定转义指令.我们注意到,转义指令之前往往有为该转义指令提供执行环境的系统调用序列,因此,我们使用 DFA 匹配系统调用序列.该 DFA 由手工定义得到,其状态节点包括系统调用和指令.

当映射系统调用状态匹配之后,根据污点操作指令,将后续匹配的内存操作指令替换为与其等价的系统调用.以上述文件映射读操作为例,相应 DFA 的前几个状态表示连续的 CreateFile,CreateFileMappint,MapViewOfFile 操作,在前面 3 个状态匹配之后,开始匹配污点数据操作指令,如果出现对污点数据进行读取的指令,则将其替换为 ReadFile.

依赖边的添加是通过在添加新节点时回溯计算污点传播过程完成的,包括数据依赖边和控制依赖边两种情况.对于数据依赖边的添加,在新节点加入时,回溯污点传播路径,查询当前污点数据与产生污点的系统调用节点之间的数据依赖关系,在相关节点和当前节点之间添加数据依赖边.由于控制依赖分析的目标是标志寄存器引起的控制流转移,因此控制依赖边的添加需要通过识别污点数据对于标志寄存器的改变和计算控制范围实现.若一个系统调用节点产生的污点数据影响了标志寄存器 EFLAGS,同时在污点传播分析中发现标志寄存器被作为控制流转移的判断条件,则计算当前控制流转移指令的后必经节点得到其控制范围.当新的系统调用产生时,判断其是否在某个污点数据的控制范围内,如果是,则在该节点和对应污点源之间添加控制依赖边.在实际分析中我们发现,代码编译时可能存在多种控制流转移结构,如 switch 分支语句、if else 条件语句、while 条件循环等.我们还发现,不同的编译器会生成不同的二进制代码,例如,在判定 eax 时,若判定是否为–1,则代码被编译为 inc eax;若是其他值,则被编译为 cmp eax,x(其中 x 代表所要判定的值或者其所在寄存器)的形式.基于以上问题,控制依赖分析的难点在于,如何全面地分析影响控制流转移的指令和如何在动态分析的过程中获取控制依赖的范围.对于分析影响控制流转移的指令,我们通过针对所有可能改变标志寄存器的指令编写处理规则来解决;对于控制依赖范围的判定,由于 switch,if, while 最终会被翻译为包含 1 个或多个 jnz 类的二进制代码,而动态分析每次只分析 1 条路径,因此仅通过动态执行代码的方法无法判定控制范围.控制依赖的范围判定可通过使用不同编译器产生的代码特征来识别特殊结构实现[27],然而,由于编译器的多样性和不同编译优化结果的可能,我们并没有采用此方法.我们在系统中整合了反汇编引擎,当发生控制流转移时,递归地反汇编其后继指令,并使用 Sreedhar 等人[26]提出的计算后必经节点的方法计算该指令的控制范围,以确定控制依赖范围.该方法存在一定的局限性,例如,使用指针数组进行的控制流转移是使用一个数组结构记录代码指针,当需要执行相应功能时,计算数组索引取出函数指针地址并跳转到该地址执行,该过程中没有使用标志寄存器而实现了控制流转移;使用我们的方法无法获取其控制依赖关系,此问题将在我们下一步的工作中改进.由于该类代码一般由手工编写而非编译器生成,很少使用,因而对于实际分析的效果影响不大.

需要说明的是,在两个系统调用节点之间,可能既存在数据依赖关系又存在控制依赖关系.对于此种情况,我们将在两节点之间同时添加数据依赖边和控制依赖边,即在两节点之间同时存在两条依赖边.

行为依赖图构建的结束有两种情况:第 1 种是当已经产生的污点记录全部被漂白时,在其后的系统调用不可能产生相关的操作,因此当前污点的行为依赖图构建结束;第 2 种情况是当恶意代码执行完成时,行为依赖图创建结束.

3.3 基于语义的抗混淆特征生成

行为依赖图可以对恶意代码行为之间的依赖关系进行描述,但是考虑到恶意代码在行为层仍会采用混淆技术进行干扰,我们从语义角度出发,针对行为层混淆技术构建抗混淆引擎,在行为依赖图的基础上进一步处理,提取出有一定抗混淆能力、可用于识别一类恶意代码变种的行为特征图.

通过对行为层混淆技术的分析,我们从语义角度将其分为语义无关调用插入和语义等价调用变换两类.语义无关调用插入是指恶意代码中常加入一些与其意图无关没有无实际作用的系统调用,目的是改变代码特征,我们从语义角度将其称为语义无关调用.语义等价调用变换主要是指通过等价系统调用替换、循环变换等方式改变执行行为的系统调用序列,从而改变特征.等价系统调用序列是指行为相同的不同系统调用序列,将其互相替换,便可在不改变行为的前提下改变特征.循环变换即指改变操作粒度,将同样的行为分割为不同循环而产生不同图特征的混淆方式.这两种混淆方法的共同特点是,在保持恶意代码行为意图即代码语义的条件下,通过等价替换改变特征,我们将此类变换统称为语义等价调用变换.

对于语义无关调用,我们通过分析污点传播过程是否对系统状态产生影响来对其进行识别并删除.我们认

为,引起操作系统完整性变化或与其他终端及软件进行信息交互的行为改变了操作系统的执行状态,即使系统状态改变可认为是改变操作系统的完整性和内部数据的安全性.我们定义可以使系统状态改变的系统调用集合 $StateChange$={*WriteFile*,*SetRegValue*,*SendTo*,…},当该集合中的系统调用返回正确即成功执行时,我们认为其改变了系统状态.

从污点传播分析的角度,语义无关调用的特点是:在污点传播过程中,没有使系统状态产生改变的节点.根据此特点,对于污点 *Taint* 传播路径上的系统调用 N_{taint1},其后的污点传播路径为 L_N,N_{taint1} 产生时 L_N=∅.根据所编写的污点传播规则,当 N_{taint1} 产生传播到 N_{taint2} 时,L_N={N_{taint1},N_{taint2}},以此类推.语义无关调用的判断条件为,当污点产生后,若满足以下两条件之一:(1) L_N=∅,(2) L_N≠∅,但∀N∈L_N,均有 N∉$StateChange$,则认为 L_N 中所有节点为语义无关调用,将其删除(若 L_N=∅,则删除 N_{taint1}),并删除与之相关的控制依赖边和数据依赖边.

对于语义等价调用的处理分为等价调用序列处理和循环处理两个方面.等价调用序列替换的混淆方法一直是特征提取及检测方法中的难点问题.恶意代码常使用功能相同的不同系统调用序列相互替换.例如,在文件的读写过程中,系统调用序列 CreateFile→CreateFileMapping→MapViewOfFile→ReadFile(其中,ReadFile 为指令转译生成)和 CreateFile→ReadFile 所执行的行为相同,常互相替换.针对这种混淆方法,本文使用语义等价序列库对等价系统调用进行识别,并创建集合节点对其进行替换.

我们创建语义等价序列库集合 F={L_{equ1},L_{equ2},…,L_{equn},*Index*}.该集合包含一系列语义等价序列库,每一个语义等价序列库是一个预设序列集合,可表示为 L_{equi}={Seq_{i1},Seq_{i2},…,Seq_{in}},集合中的所有系统调用序列的语义相同,可相互替换.同时,构造所有语义等价序列库的索引 *Index*={E_a,E_b,E_c,…}.其中,E_i 表示库中序列的首节点名称,每个节点对应一个集合,保存以该节点为首节点的所有序列的所在库信息,记为 E_i={L_{equ1},L_{equ2},…,L_{equn}}.其中,L_{equn} 表示以 E_i 为首节点的序列所在的库名即集合节点名.一个集合节点在行为特征图中代表一个语义等价序列库,其内部存储一个链接,链接到其代表的语义等价序列库中所有序列的首节点.即,将同一语义等价序列库中的多个系统调用序列映射为相同的内容,用相应的集合节点进行表示.例如,在上述读文件的例子中,所有读文件的系统调用序列均属于一个语义等价序列库,以集合节点 $READ_{equ}$ 表示.

在特征提取过程中,对等价系统调用序列的识别是一个序列匹配过程.首先在库集合 F 的索引 *Index* 中查找相应节点,然后依照该节点保存的库信息到相应库中找到该节点为首的序列向下匹配,若有一个库中的一条序列匹配成功,则将行为依赖图中该序列缩减为代表该库的集合节点 N',并将原有的数据依赖边和控制依赖边都指向该集合节点.

语义等价调用处理的另一种情况是循环变换的处理.为了应对等价循环变换对图特征的干扰,我们将行为依赖图中的循环缩减,替换为一次执行信息.本文分析中所关注的循环操作是针对污点进行的,因此可根据污点传播记录分析循环.在每次污点调用操作中,我们记录污点的操作范围信息,对于每一条指令都有污点起始地址和操作长度的记录.针对循环中的指令,我们设置记录结构 $Instruction_i$={*TaintOpAddress*,*TaintOpLength*},循环可通过判定行为节点的链表中是否有回边来识别,当发现有循环时,从循环的入口节点开始自动比较该循环中的系统调用发生地址.设前一次循环中污点操作节点为 $SysCall_{Prior}$,后一次循环中相同系统调用地址的污点行为节点为 $SysCall_{Succ}$,计算 $SysCall_{Prior}$ 和 $SysCall_{Succ}$ 之间的污点操作范围.如果有 $SysCall_{succ}$.*TaintOpAddress*=$SysCall_{prior}$.*TaintOpAddress*+$SysCall_{prior}$.*TaintOpLength*,则将 $SysCall_{Succ}$ 与 $SysCall_{Prior}$ 合并为 $SysCall_{Prior}$,并将其污点操作范围修改为二者污点操作范围之和.

基于以上提取过程,我们将恶意代码行为特征图描述为 G={*Entry*,*Exit*,N,N',C,D},其中,*Entry* 表示图的入口节点,*Exit* 表示出口节点,N 表示一般系统调用节点,N'表示集合节点,C 表示控制依赖边,D 表示数据依赖边.由于在实际分析中可能存在多个污点源而产生多个行为特征图,恶意代码特征为行为特征图的集合:

$$T=\{G_1,G_2,\ldots,G_n\},\ n\in N.$$

我们以 ForBot 样本中的一段代码为例,由行为依赖图提取行为特征图的过程如图 3 所示.其中,图 3(a)为通过可回溯的污点传播分析构建的行为依赖图.首先识别其中的语义无关调用并将其删除,如图中 CreateFile→ReadFile,其过程未引起系统状态改变,根据本文的方法判定其为语义无关调用并将其删除,语义无关调用处理

的结果如图 3(b)所示;然后将 ReadFile,SendTo 循环的各分支缩减,替换为 ReadFile→SendTo,如图 3(c)所示;最后识别出序列 CreateFile→ReadFile 在语义等价序列库 $READ_{equ}$ 中,将此序列替换为集合节点 $READ_{equ}$,得到行为特征图,如图 3(d)所示.

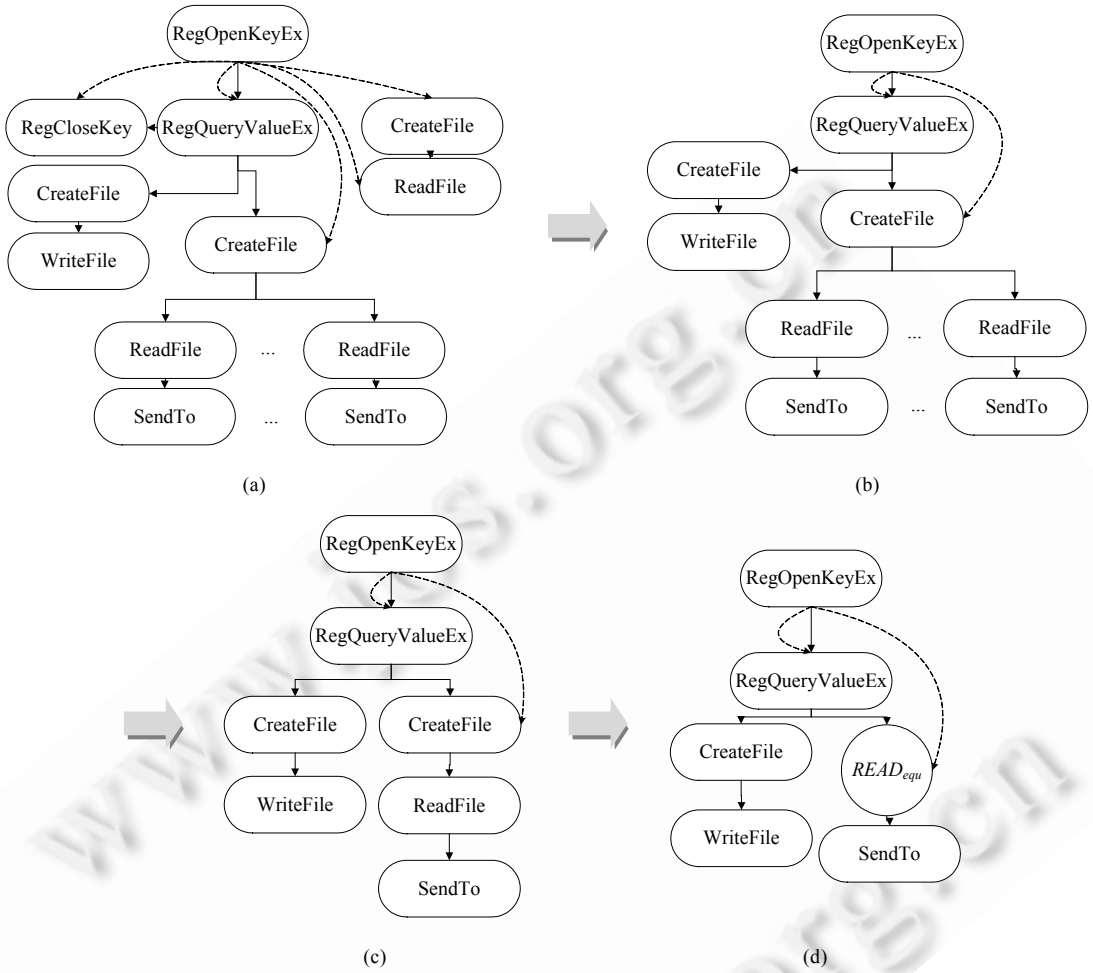

Fig.3 Example of anti-obfuscation process of forbot code segment

图 3 ForBot 样本代码片段经抗混淆引擎处理过程示例

4 基于行为特征图的检测

在完成恶意代码特征提取之后,我们利用行为特征图对恶意代码进行检测.我们对待测代码的执行流程进行污点传播分析,识别其中的关键系统调用、数据依赖关系和控制依赖关系,与特征库中已构造的行为特征图进行比较,计算匹配度对其进行判定.

我们使用 *TGInput* 描述备选特征集合,$TGInput=\{T_1,T_2,\ldots,T_n\}$,$n\in N$,其中,$T_x=\{G_{x1},G_{x2},\ldots,G_{xn}\}$.为每个特征 T_x 分配一个计数集合 $M_x=\{n_N,n_{N'},n_C,n_D\}$,用于在检测过程中表示待测代码与特征 T_x 匹配的节点和边的计数.其中,n_N 表示已匹配的一般节点数,$n_{N'}$表示已匹配的集合节点数,n_C 表示已匹配的控制依赖边数,n_D 表示已匹配的数据依赖边数.在初始状态下,*TGInput* 为特征库中的所有行为特征.跟踪待测代码的执行流程,根据污点传播获得的系统调用及依赖关系信息与行为特征图进行匹配.首先是入口点的匹配,在遇到第 1 个系统调用时,从特征库中查找 *Entry* 为该系统调用的行为特征图 G_{xy},其中,$G_{xy}\in T_x$.此时得到的行为特征图可能是 1 个或多个,将这些图

G_{xy}所在的特征 T_x保留在 *TGInput* 中等待进一步判断,并在每个 T_x对应的计数集合 M_x中开始相应计数,其他无关特征从 *TGInput* 中删除.然后,以该系统调用的返回值为污点开始污点传播计算.此后,代码执行系统调用时,查询其参数使用的数据是否为污点数据以及该污点数据的污点源.如果该系统调用引用的参数不是污点,则将其舍弃,继续污点传播过程;如果该系统调用引用的参数为污点,则开始在备选特征集合 *TGInput* 中各 T_x的行为特征图 G_{xy}中查找相应的节点和依赖边,删除不包含上述节点和依赖边的特征图.当一个特征 T_m中没有能够匹配的行为特征图时,将 T_m从备选特征集合中删去,缩小备选特征集合 *TGInput*;对于匹配的行为特征图所在特征,更新其计数集合 M 中对应的计数值.

本文检测方法的核心是待测代码与行为特征图的匹配.在匹配过程中,通过判断待测代码的执行流程,对其中执行的系统调用及依赖关系进行图匹配算法.匹配过程主要包括:一般系统调用节点的匹配、集合节点的匹配、边的匹配和匹配计数.

行为特征图中节点的匹配分为一般节点的匹配和集合节点的匹配.一般节点表示单个系统调用,其匹配内容包括系统调用名称的匹配和系统调用参数的匹配两部分.系统调用名称的匹配是根据字符串比较的方法判定两个系统调用是否相同;系统调用参数的匹配则需要根据不同参数类型分别考虑.我们把系统调用参数分为句柄、文字、枚举和结构 4 种类型.句柄类型用来判断系统调用之间的调用返回值是否有关联,从而判断是否构成数据依赖关系;文字类型和枚举类型使用字符串比较和数值比较的方法判定两参数是否一致;结构类型则根据结构中数据类型的不同,进行字符串、数值等方式的比较.需要注意的是,在系统调用的匹配中,我们并不要求所有的域都相同.例如,打开文件时,通过参数确定为打开操作即可,至于该文件是否共享读写,则不属于关注范围.为此,我们为不同类别的系统调用设定不同的匹配规则,当系统调用的名称和在匹配规则中关注的参数都一致时,认为两个系统调用匹配,同时更新 M 中的一般调用计数值为 n_N+1.

集合节点表示一个语义等价序列库.当匹配过程中遇到集合节点时,保存此时的各数据状态,然后转到相应的语义等价序列库中进行匹配.当库中有序列的首节点被匹配时,继续匹配剩余的系统调用部分,不在原行为特征图中向下匹配.语义等价序列库的匹配方法如下:首先选取首节点和待匹配系统调用相同的序列作为备选行为序列,其余的序列从当前比较集合中移除.当待测代码发生新的系统调用时,在备选序列中继续匹配.在此过程中,需要注意判断当前产生的系统调用是否还在语义等价序列库中.由于语义等价序列库中各序列的系统调用之间必然存在数据依赖关系,因此可根据污点传播过程回溯得到的恶意代码数据依赖关系判定新发生的系统调用是否仍在库中.继续匹配过程,若库中有一条序列匹配成功,则认为该集合节点匹配完成,更新相应 M 中集合节点计数为 $n_{N'}$+1,然后回到原特征图中继续匹配.

边的匹配与节点的匹配同时进行,当特征图中的节点被匹配时,自动回溯计算当前节点和其余已匹配节点之间的依赖关系,根据得到的数据依赖关系及控制依赖关系判定两个节点之间的数据依赖边或控制依赖边是否相同,并更新相应的依赖边计数值.

需要注意的是,对于恶意代码,在前期的分析中提取的特征是行为特征图的集合,可能出现多个图之间系统调用有重叠的情况.对于有系统调用重叠的情况,我们通过在构建依赖图时记录系统调用地址来识别重叠,在检测时对于已匹配的重叠节点不做重复计算.

在匹配过程完成后,根据匹配结果计算加权特征值.我们为行为特征图中的节点和边定义相应的权值,根据匹配结果中 M 集合的相应计数值计算匹配行为的加权值,将其称为特征值.我们根据系统调用、数据依赖关系和控制依赖关系对恶意行为的影响程度不同为它们赋予不同的权值,其中,一般系统调用的权值为 0.8,集合节点的权值为 1,控制依赖边的权值为 0.1,数据依赖边的权值为 0.2.特征值的加权计算公式为

$$E=\frac{\sum_{x=N,N',C,D} p_x n_x}{\sum_{x=N,N',C,D}{}_{Graph}\, p_x n_x},$$

其中,E 表示特征值,p_x为权值,n_x为相应节点或边的个数,N,N',C,D 分别代表一般节点、集合节点、控制依赖边和数据依赖边.

该公式计算的是待测代码与特征库中的特征相匹配的节点和边的权值之和与该特征中所有节点和边的权值之和的商.我们认为,这个商表示了待测代码与特征库中恶意代码特征的匹配程度.在检测中,对特征值设定阈值 *Threshold*,*Threshold*∈[0,1].当特征值计算结果 $E \geqslant Threshold$ 时,判定待测代码为恶意代码.在实际分析中发现,当 *Threshold*=0.7 时效果最好,我们在实验中即采用了该阈值.基于特征值的检测避免了二元判定的绝对性带来的漏报和误报问题.检测阈值可以根据检测需要进行设置.例如,当主要关注低漏报率时,可将阈值适度降低,此时,将可一并检测到部分通过修改或增加行为而产生的恶意代码新种.

5 系统设计及实验

5.1 系统设计

恶意代码特征提取及检测系统是在 WooKon 平台的基础上实现的.WooKon 是一个恶意代码动态分析平台.该平台基于硬件模拟器 QEMU[15]开发,在其中运行 Guest OS 虚拟系统.对在虚拟系统中运行的进程进行数据采集及动态分析,可实现逐一指令的细粒度分析,剖析所运行代码的机理及实现细节.该平台将分析模块与分析目标处于不同环境,可实现对分析目标的高度透明,不易被恶意代码察觉.该平台具有可扩展能力,通过增加模块实现不同需求.我们通过扩展 WooKon 平台的污点传播分析模块,添加特征提取模块和检测模块来实现恶意代码特征提取及检测原型系统.

我们的原型系统主要由硬件模拟环境、污点传播分析模块、特征提取模块和检测模块这 4 部分组成,如图 4 所示.

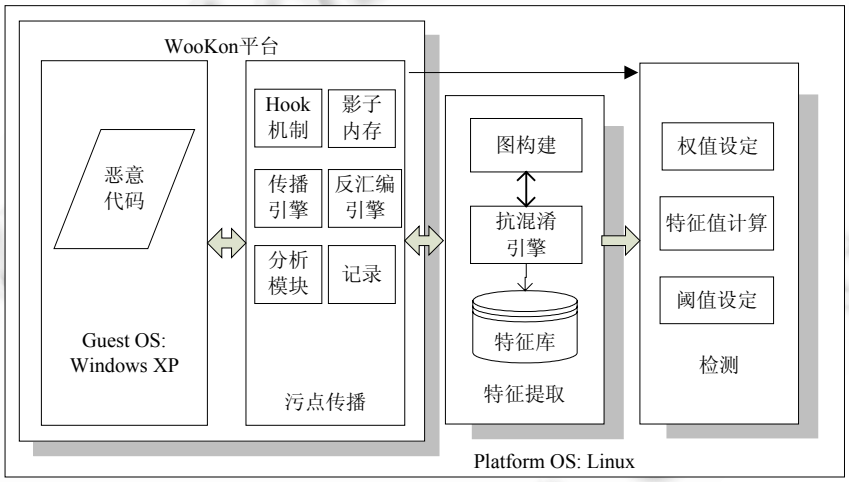

Fig.4 Prototype system framework

图 4 原型系统框架图

我们在硬件模拟环境中运行 Windows XP 虚拟系统,在其中运行并监控恶意代码;污点传播分析模块主要在 WooKon 平台的污点传播分析功能的基础上扩展实现,提高了回溯分析能力和依赖关系的提取能力;特征提取模块主要根据污点传播分析模块采集的数据构建行为依赖图,并利用抗混淆引擎进行处理,提取恶意代码行为特征;检测模块中,根据待测代码的污点传播分析数据,匹配特征库中的恶意代码行为特征图集合,通过特征值计算结果对待测代码进行判定.

5.2 实 验

为了验证所提取的恶意代码特征对于真实样本的效果,我们在实验中选取一些较有代表性的恶意代码及其变种样本进行了测试.在实验中,我们选取单个恶意代码样本提取行为特征,利用此特征对剩余变种进行检

测,对实验结果进行分析评估.

在本文中,我们以 NetSky 及 SdBot 样本的实验结果为例,说明特征提取及检测结果.NetSky 是一种蠕虫病毒,经电子邮件传播.当使用者下载及开启病毒邮件的附件后,该蠕虫自动扫描所有电子邮件地址并发送邮件进行传播.该蠕虫首个版本出现两日后即出现了变种.SdBot 是一种集 IRC 后门、蠕虫功能于一体、通过网络共享进行传播的病毒,会在感染的电脑上打开后门接收攻击者发出的指令,然后连接特定的 IRC 服务器通知攻击者.

文中用于实验的真实样本来自 VX Heavens[28]网络病毒数据库,库中样本由该网站部署的蜜罐和捕获工具获取,是恶意代码研究中重要的样本来源之一.本文实验中的 NetSky 样本全部由病毒数据库 VX Heavens 中获取,在完全真实的环境下测试特征检测效果.同时,为了测试我们的特征检测方法在不同混淆及加壳强度下的效果,对 SdBot 样本,除了从 VX Heavens 中选取部分测试样本(后缀为 ori,bx 和 by 的样本)以外,我们还通过手工混淆方法生成了 SdBot.ma 样本,并分别使用加壳工具 UPX,EXECryptor 和 Asprotect 对混淆过的样本进行处理,得到不同加壳强度下的 3 个混淆样本(后缀分别为 up,ex 和 as 的样本).

在 NetSky 的实验中,首先由 NetSky.ad 样本提取行为特征作为检测基准,然后对其余多个变种进行检测,计算特征值,检验匹配效果.实验结果见表 1.

Table 1 Match result and eigen value of netsky variants

表 1 NetSky 变种匹配情况及特征值计算结果

样本名称	系统调用匹配数(总数)	集合节点匹配数(含系统调用数)	一般节点匹配数	控制依赖匹配数	数据依赖匹配数	特征值
NetSky.ad	27	5 (15)	12	43	31	—
NetSky.aa	21 (3079)	4 (12)	9	30	27	0.78
NetSky.af	25 (1093)	5 (15)	10	38	28	0.89
NetSky.c	23 (188)	5 (15)	8	36	23	0.78
NetSky.f	24 (844)	5 (15)	9	35	25	0.82
NetSky.r	23 (47)	4 (12)	11	30	23	0.81
NetSky.t	19 (8220)	3 (9)	10	27	21	0.71

在表 1 中,NetSky.ad 一行的数据为提取出的行为特征中所含的系统调用及依赖边的个数,在提取特征的过程中,分析 NetSky.ad 得到的行为依赖图节点数为 53 个,最终提取的行为特征图节点为 27 个,这是由于经过抗混淆处理缩减了一定的数据量.

由表 1 可以看出,在 NetSky 各变种样本中监控到的系统调用个数远远大于其匹配的系统调用个数.经分析,其原因是 NetSky 代码中含有大量垃圾调用和循环操作,而我们在构造特征图时对其进行了化简.例如,在实验中作为基准提取行为特征的 NetSky.ad 样本为了将自身复制到 C:\WINDOWS\Jammer2nd.exe 文件,使用了连续 ReadFile 和 WriteFile 操作,在提取过程中我们对该循环进行了缩减.在检测中发现,混淆方法最突出的是 NetSky.t 样本.该样本在将自身复制为 C:\WINDOWS\base64.tmp 时循环次数高达 4 229 次.此类循环和等价替换操作给基于行为序列的检测带来了很大的干扰,然而对我们的行为特征检测方法没有影响.

在 SdBot 样本的实验中,为了验证特征提取和检测方法对混淆代码的识别能力,我们运用手工混淆方法对 SdBot 样本进行了部分代码特征的去除.我们采用直接修改 SdBot.ori 源码的方式,使变形之后的恶意代码不会被杀毒软件查杀.手工混淆使用的方法包括:加入垃圾系统调用、加入无用的数学运算指令和等价系统调用替换等.在此基础上使用加壳工具进行处理,由 SdBot.ori 样本产生了 4 个变种(SdBot.up,SdBot.ex,SdBot.as 和 SdBot.ma).根据由 SdBot.ori 样本提取的行为特征,对这些变种以及从 VX Heavens 库中获取的 SdBot.bx 和 SdBot.by 样本进行特征值计算,实验结果见表 2.

SdBot.ori 一行的数据为提取的行为特征中所含的相应系统调用及依赖边个数.在提取过程中,由污点传播分析得到的行为依赖图节点数为 293 个,经抗混淆提取后的行为特征节点数为 42 个.可见,与依赖图相比,行为特征图缩减了数据量.

以变种 SdBot.bx 为例,在分析中可知,SdBot.bx 产生了大量循环代码,在调用 RegSetValueExW 后连续调用 6 次 connect 连接远程地址,若不成功则循环该操作.由于我们在特征提取中进行了循环缩减处理,这样的循环并没有影响我们的分析结果.

Table 2 Match result and eigen value of SdBot variants

表 2 SdBot 变种匹配情况及特征值计算结果

样本名称	系统调用匹配数(总数)	集合节点匹配数(含系统调用数)	一般节点匹配数	控制依赖匹配数	数据依赖匹配数	特征值
SdBot.ori	42	10 (28)	14	103	61	—
SdBot.up	39 (155)	10 (28)	14	87	52	0.92
SdBot.ex	33 (187)	7 (22)	11	73	41	0.72
SdBot.as	37 (176)	8 (24)	13	78	53	0.84
SdBot.m	42 (1 653)	10 (28)	14	82	56	0.93
SdBot.bx	41 (95)	10 (28)	13	76	50	0.87
SdBot.by	33 (1769)	10 (28)	11	84	48	0.84

为了检验本文的行为特征检测方法的误报率,我们使用一些正常程序进行对比检测.在实验中,我们选取了比较有代表性的 IE 浏览器程序、FTP 程序、计算程序和记事本程序.根据上述例子中提取的恶意代码特征(NetSky.ad 特征及 SdBot.ori 特征)对其进行检测,计算特征值,并将计算结果与恶意代码变种的特征值计算结果进行比较,如图 5 所示.由图可以看出,基于本文的特征提取及检测方法,正常代码的特征值计算结果均在 0.4 以下;相比之下,恶意代码变种的特征值均在 0.7 以上.可见,本文提取的行为特征及检测方法对正常代码及恶意代码变种有较好的辨别能力.

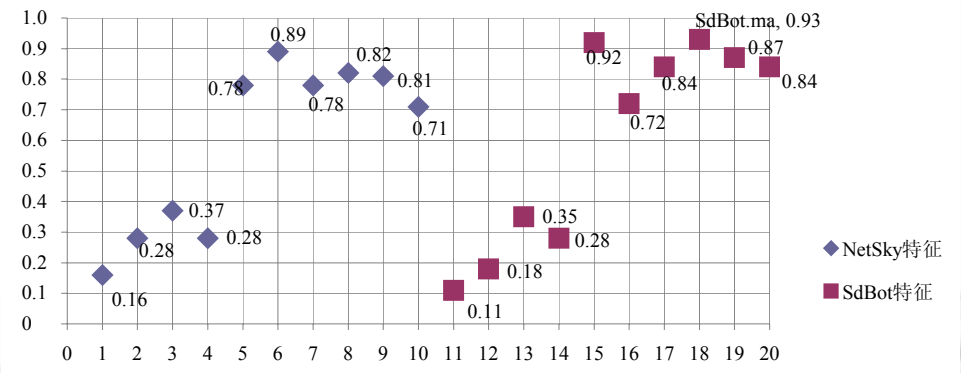

Fig.5 Comparison of eigen value of normal program and malware variants

图 5 正常程序及恶意代码变种的特征值计算结果比较

为了比较测试结果,我们使用常用杀毒软件及工具 NOD32、诺顿(Norton)、卡巴斯基(KAV)及 BitDefender 对恶意代码及其变种进行检测和比较分析.仍以 NetSky 和 SdBot 为例,检测效果见表 3.

Table 3 Comparison of detection result

表 3 检测结果对比

样本名称	检测工具				
	NOD32	Norton	KAV	BitDefender	Ours
NetSky.ad	YES	YES	YES	YES	YES
NetSky.aa	YES	YES	YES	YES	YES
NetSky.af	YES	YES	YES	YES	YES
NetSky.c	YES	YES	YES	NO	YES
NetSky.f	YES	YES	YES	YES	YES
NetSky.r	YES	YES	YES	NO	YES
NetSky.t	YES	YES	YES	YES	YES
SdBot.ori	YES	YES	YES	YES	YES
SdBot.up	NO	NO	NO	NO	YES
SdBot.ex	NO	NO	NO	NO	YES
SdBot.as	NO	NO	NO	NO	YES
SdBot.ma	NO	NO	NO	NO	YES
SdBot.bx	YES	YES	YES	YES	YES
SdBot.by	YES	YES	YES	YES	YES

经分析发现,大部分杀毒软件使用的仍然是代码特征和序列特征匹配技术.NOD32 杀毒软件由于使用了启发式扫描技术,普通插入垃圾调用的混淆方法难以绕过该杀毒引擎.但由于我们在 SdBot 手工混淆的程序入口加入了运算开方的垃圾代码,导致 NOD32 无法检测相应的行为,从而绕过了该杀毒引擎.通过比较分析可见,相对于传统方法,本文提出的特征提取和检测方法具有一定的优势.

5.3 讨 论

通过对多种恶意代码样本进行实验测试,验证了本文提出的恶意代码行为特征及检测方法的有效性.本文提出的特征提取及检测方法提高了对恶意代码变种的检测能力,对正常程序与恶意代码有较好的识别度,且在一定程度上降低了特征数据量.在实验中我们发现,检测系统的时间效率并不十分理想,这是由目前污点传播分析技术的效率问题造成的.本文使用可回溯的动态污点传播分析方法对恶意代码进行分析及检测,这种方法需要对代码进行逐条指令地分析,因此不可避免地带来系统效率的降低.对于此问题,我们拟在以后的工作中对检测部分的污点传播分析进行改进研究,提高时间效率.另外,由于动态分析的局限性,在恶意代码一次执行的监控中只分析到 1 条路径,可能无法获取全面的执行路径.对于此问题,在接下来的工作中可采用多路径挖掘方法[11]进行改善.

6 总 结

本文提出了一种基于语义的恶意代码行为特征提取及检测方法,通过在硬件模拟环境下监控恶意代码执行,利用可回溯的污点传播技术对恶意代码进行指令层分析,提取关键系统调用、依赖关系及相关指令信息,根据这些信息结合行为层分析抽象语义,利用抗混淆引擎提取恶意代码行为特征.在此基础上,对恶意代码及变种进行检测.通过原型系统上的实验,验证了本方法提取的特征具有较好的抗混淆能力,基于此特征的检测方法对恶意代码变种有较好的识别能力.

在下一步的工作中,我们将结合多路径分析方法提取全面路径的特征信息,并研究多样本特征融合的问题;同时,改进检测过程中污点传播分析对时间效率的影响,进一步提高检测能力.

References:

[1] Symantec global Internet security threat report, trends for 2008. Vol.14, 2009. http://www.symantec.com/business/theme.jsp?themeid=threatreport

[2] Li Y, Zuo ZH. An overview of object-code obfuscation technologies. Journal of Computer Technology and Development, 2007, 17(4):125−127 (in Chinese with English abstract).

[3] Yin H, Song D, Egele M, Kruegel C, Kirda E. Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proc. of the 14th ACM Conf. on Computer and Communications Security. Alexandria, 2007. [doi: 10.1145/1315245.1315261]

[4] Christodorescu M, Jha S. Testing malware detectors. In: Proc. of the 2004 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2004). Boston, 2004. 34−44. [doi: 10.1145/1007512.1007518]

[5] Jacob G, Debar H, Fillol E. Behavioral detection of malware: From a survey towards an established taxonomy. Journal in Computer Virology, 2008,4(3):251−266. [doi: 10.1007/s11416-008-0086-0]

[6] Parampalli C, Sekar R, Johnson R. A practical mimicry attack against powerful system-call monitors. In: Proc. of the 2008 ACM Symp. on Information, Computer and Communications Security. New York, 2008. 156−167. [doi: 10.1145/1368310.1368334]

[7] Sathyanarayan VS, Kohli P, Bruhadeshwar B. Signature generation and detection of malware families. In: Proc. of the 13th Austalasian Conf. on Information Security and Privacy. Wollongong, 2008. 336−349. [doi: 10.1007/978-3-540-70500-0_25]

[8] Christodorescu M, Jha S, Seshia SA, Song DX, Bryant RE. Semantics-Aware malware detection. In: Proc. of the 2005 IEEE Symp. on Security and Privacy. 2005. 32−46. [doi: 10.1109/SP.2005.20]

[9] Preda MD, Christodorescu M, Jha S, Debray S. A semantics-based approach to malware detection. In: Proc. of the Symp. on Principles of Programming Languages. New York: ACM Press, 2007. 377−388. [doi: 10.1145/1190216.1190270]

[10] Kinder J, Katzenbeisser S, Schallhart C, Veith H. Detecting malicious code by model checking. In: Proc. of the 2nd Int'l Conf. on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA 2005). LNCS 3548, Vienna: Springer-Verlag, 2005. 174−187. [doi: 10.1007/11506881_11]

[11] Christodorescu M, Kinder J, Jha S, Katzenbeisser S, Veith H. Malware normalization. Technical Report, #1539, Madison: University of Wisconsin, 2005.

[12] Moser A, Kruegel C, Kirda E. Exploring multiple execution paths for malware analysis. In: Proc. of the 2007 IEEE Symp. on Security and Privacy. 2007. 231−245. [doi: 10.1109/SP.2007.17]

[13] Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using CWSandbox. IEEE Security and Privacy, 2007, 5(2):32−39. [doi: 10.1109/MSP.2007.45]

[14] Bayer U, Kruegel C, Kirda E. TTAnalyze: A tool for analyzing malware. In: Proc. of the EICAR 2006. 2006. 180−192.

[15] Bellard F. Qemu, A fast and portable dynamic translator. In: Proc. of the USENIX 2005 Annual Technical Conf. on FREENIX Track. 2005. 41−46.

[16] Kirda E, Kruegel C, Banks G, Vigna G, Kemmerer RA. Behavior-Based spyware detection. In: Proc. of the 15th Conf. on USENIX Security Symp. Springer-Verlag, 2006. 273−288.

[17] Bailey M, Oberheide J, Andersen J, Mao ZM, Jahanian F, Nazario J. Automated classification and analysis of Internet malware. In: Proc. of the 10th Symp. on Recent Advances in Intrusion Detection (RAID 2007). 2007. 178−197.

[18] Bergeron J, Debbabi M, Desharnais J, Erhioui MM, Lavoie Y, Tawbi N. Static detection of malicious code in executable programs. In: Proc. of the Symp. on Requirements Engineering for Information Security. 2001.

[19] Christodorescu M, Jha S. Static analysis of executables to detect malicious patterns. In: Proc. of the 12th USENIX Security Symp. 2003.

[20] Bilar D. Statistical Structures: Tolerant fingerprinting for classification and analysis. In: Proc. of the Black Hat USA 2006. Las Vegas, 2006.

[21] Christodorescu M, Jha S, Kruegel C. Mining specifications of malicious behavior. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE). 2007. [doi: 10.1145/1287624.1287628]

[22] Cogswell B, Russinovich M. Rootkit revealer. 2006. http://www.microsoft.com/technet/sysinternals/Utilities/RootkitRevealer.mspx

[23] Wang YM, Roussev R, Verbowski C, Johnson A, Wu MW, Huang YN, Kuo SY. Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In: Proc. of the 18th Systems Administration Conf. (LISA 2004). 2004. 33−46.

[24] Butler J, Hoglund G. VICE—Catch the hookers! In: Proc. of the Black Hat USA 2004. 2004. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf

[25] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the 12th Annual Network and Distributed System Security Symp. (NDSS). 2005.

[26] Sreedhar VC, Gao GR, Lee YF. Identifying loops using DJ graphs. ACM Trans. on Programming Languages and Systems (TOPLAS), 1996,18(6):649−658. [doi: 10.1145/236114.236115]

[27] Hex-Rays. http://www.hex-rays.com

[28] VX heavens. http://vx.netlux.org/

附中文参考文献:

[2] 李勇,左志宏.目标代码混淆技术综述.计算机技术与发展,2007,17(4):125−127.

王蕊(1981－),女,北京人,博士,助理研究员,CCF 会员,主要研究领域为网络与系统安全.

杨轶(1982－),男,博士,助理研究员,主要研究领域为恶意代码分析与防范.

冯登国(1965－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为密码学,信息安全.

苏璞睿(1976－),男,博士,副研究员,主要研究领域为恶意代码分析与防范.