软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2011,22(10):2438−2453 [doi: 10.3724/SP.J.1001.2011.03888]

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

# 基于行为依赖特征的恶意代码相似性比较方法*

杨 轶[1,3+], 苏璞睿[1], 应凌云[1], 冯登国[1,2]

[1](信息安全国家重点实验室 中国科学院 软件研究所,北京 100190)

[2](信息安全国家重点实验室 中国科学院 研究生院,北京 100049)

[3](信息安全共性技术国家工程研究中心,北京 100190)

## Dependency-Based Malware Similarity Comparison Method

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

[1](State Key Laboratory of Information Security, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](State Key Laboratory of Information Security, Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

[3](National Engineering Research Center for Information Security, Beijing 100190, China)

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

**Yang Y, Su PR, Ying LY, Feng DG. Dependency-Based malware similarity comparison method. *Journal of Software*, 2011,22(10):2438−2453.** http://www.jos.org.cn/1000-9825/3888.htm

**Abstract**: Malware similarity comparison is one of the basic works in malware analysis and detection. Presently, most similarity comparison methods treat malware as CFG, or behavior sequences. Malware writers use obfuscation, packers and other means of technique to confuse traditional similarity comparison methods. This paper proposes a new approach in identifyling the similarities between malware samples, which rely on control dependence and data dependence. First, the dynamic taint analysis is performed to obtain control dependence relations and data dependence relations. Next, a control dependence graph and data dependence graph are constructed. Similarity information is obtained by comparing these two types of graph. In order to take full advantage of the inherent behavior of malicious codes and to increase the accuracy of comparison and anti-jamming capability, the loops are recued and the rubbish is removed by means of the dependence graph pre-processing, which reduces the complexity of the similarity comparison algorithm and improves the performance of the algorithm. The proposed prototype system has been applied to wild malware collections. The results show that the accuracy of the method and comparison capabilities all have an obvious advantage.

**Key words**: malware; similarity comparison; dynamic analysis; taint propagation

**摘 要**: 恶意代码相似性比较是恶意代码分析和检测的基础性工作之一,现有方法主要是基于代码结构或行为序列进行比较.但恶意代码编写者常采用代码混淆、程序加壳等手段对恶意代码进行处理,导致传统的相似性比较方法失效.提出了一种基于行为之间控制依赖关系和数据依赖关系的恶意代码相似性比较方法,该方法利用动态污点传播分析识别恶意行为之间的依赖关系,然后,以此为基础构造控制依赖图和数据依赖图,根据两种依赖关系进行恶意代码的相似性比较.该方法充分利用了恶意代码行为之间内在的关联性,提高了比较的准确性,具有较强的抗干扰

* 基金项目: 国家自然科学基金(60703076); 国家高技术研究发展计划(863)(2007AA01Z451, 2009AA01Z435)
收稿时间: 2010-01-08; 修改时间: 2010-03-30; 定稿时间: 2010-05-14

能力;通过循环消除、垃圾行为删除等方法对依赖图进行预处理,降低了相似性比较算法的复杂度,加快了比较速度.实验结果表明,与现有方法相比,该方法的准确性和抗干扰能力均呈现明显优势.

**关键词:** 恶意代码;相似性比较;动态分析;污点传播

**中图法分类号:** TP309 **文献标识码:** A

随着计算机网络的深度应用和恶意代码技术的不断发展,恶意代码带来的危害越来越大.根据微软的统计[1],全球约三分之一的计算机被安装了木马后门等各种恶意代码.该报告仅统计了可检测的样本,实际危害程度可能更高.恶意代码的发作逐渐呈现变种速度快、模块重复使用度高的特点.如 2008 年爆发的机器狗病毒,最快时每两小时产生一个新的变种.针对恶意代码的上述特点,研究人员开始使用相似性比较方法进行离线分析,识别恶意代码变种,根据比较结果快速构造恶意代码行为轮廓,并根据恶意代码的行为轮廓辅助构建该类代码的行为特征.恶意代码的相似性比较方法已成为目前国内外研究的热点和难点问题之一.

恶意代码的相似性比较主要包括静态结构相似性比较和动态行为相似性比较两类.静态结构相似性比较通过静态反汇编构造程序的控制流图(CFG),根据控制流图的结构特征进行相似性比较.Wang[2]提出的二进制文件比较方法是该方面的代表性工作.该方法首先以基本块(basic block)为单位比较代码的相似性,再根据代码中数据操作的起始地址和访问长度比较数据块.该方法需要对程序进行准确的反汇编,因此其往往对加壳代码等特殊代码无能为力.另外,控制流图的结构特征容易受到代码混淆等手段的干扰,影响比较的准确性.因此,静态相似性比较的方法虽然实现较为简单,但容易受到各种代码保护技术的干扰,具有较大的局限性.

动态相似性比较主要根据获取的行为序列进行相似性判定,其判定方法多基于行为序列的文字距离或系统调用的敏感程度进行.Bayer 等人[3]提出的可缩放的代码分析方法是该方面的代表性工作,该方法对代码执行中的数据进行污点传播分析,提取执行序列,并根据行为关联数据进行相似性比较.该方法虽然使用了污点传播手段,但其比较过程仍然针对于行为序列.动态相似性比较方法虽然可以解决静态比较方法无法解决的加壳代码等特殊代码的相似性比较等问题,但现有的方法只根据行为序列的文字距离或加权文字距离进行比较,没有充分利用程序行为内部的逻辑关系,忽视了程序代码和行为之间的依赖关系,容易被恶意代码通过系统调用重排和加入垃圾调用等方法增大行为序列差异,在比较中产生较大的误差.

针对上述问题,本文提出了一种基于行为之间控制依赖和数据依赖关系的恶意代码相似性比较方法.该方法首先对现有的控制依赖图和数据依赖图进行了改造,提出了基于带有虚拟节点的控制依赖图和扩展数据依赖图的恶意行为描述方法,提高了对行为之间依赖关系的描述能力.然后,通过动态污点传播方法分析目标程序的控制依赖和数据依赖关系,构造带有虚拟节点的控制依赖图和扩展数据依赖图,并对依赖图进行垃圾调用删除、循环缩减和行为轮廓构建等预处理,减少由于添加垃圾调用、等价替换和系统调用重排等引起的干扰,缩小了图的规模,降低了相似性比较的复杂度.最后,利用基于扩展依赖图的相似性比较方法对控制依赖图和数据依赖图进行相似性比较,确定目标程序之间的相似程度.该方法基于动态污点传播实现,不依赖于反汇编代码,不受加壳代码的干扰;充分利用了行为之间的逻辑关系,不受代码重排和垃圾行为的影响;使用代码行为轮廓和前面比较的结果作为后期比较的指引,在准确比较恶意代码行为依赖关系相似性的同时降低了比较的复杂度.该方法主要用于获取恶意代码样本后的离线分析,快速划分恶意代码的族类并构建该类恶意代码的行为轮廓.

本文的主要贡献总结如下:

(1) 改进了恶意代码行为描述方法.通过扩展现有的数据依赖图和控制依赖图,在其中加入表示行为之间可替换关系的虚拟节点,使之具有描述可替换行为的能力,从而提高了恶意代码行为描述方式的抗干扰能力.并使用依赖图预处理算法去除了代码混淆手段带来的干扰,分析并构建行为轮廓作为相似性比较算法的输入,降低了比较的复杂度;

(2) 提出了一种基于扩展依赖图的相似性比较方法.该方法使用依赖图行为轮廓信息和已经完成的相似性比较结果作为后期比较的指引,在提高比较准确性的同时,降低了行为相似性比较的复杂度;

(3) 设计实现了一套原型系统并完成了相关实验,通过对 SDBot 等恶意代码及其变种的相似性比较,验证

并评估了该方法的正确性和有效性.

本文第 1 节介绍目前国内外的研究进展.第 2 节通过一个实例描述我们要解决的问题.第 3 节讨论代码之间的依赖关系构建方法和相似性比较算法.第 4 节说明系统整体设计.第 5 节详细介绍我们的实验设计和结果.最后总结全文.

## 1 相关工作

根据分析方法的不同,可将恶意代码相似性比较方法分为静态结构比较和动态行为比较.静态结构相似性比较根据静态反汇编的结果进行比较.反汇编主要依赖于工具 Objdump,IDA Pro 和 W32Dasm 等.Wang[2]提出了比较文件中代码和数据来判断相似性的方法,该方法首先以基本块为单位比较代码,之后根据数据操作的地址和长度比较数据块,适用于代码仅有小范围改动且未作混淆和加壳的情况.Flake[4]和 Dullien 等人[5]分别以不同的顺序和条件对代码中的函数和调用图进行比较以判断相似性.DOME[6]系统根据代码中系统调用地址是否相同来判断代码相似性.BinHunt[7]系统基于控制流图进行比较并使用符号执行进行语义分析,该方法提高了相似性比较的准确性,但由于其针对基本块进行比较,因此难以分析混淆代码.静态相似性比较方法的基础是正确的静态反汇编,由于恶意代码普遍使用混淆手段干扰分析,导致其难以获得正确的结果.同时,静态结构比较针对于代码之间的次序,忽略了其中的语义特征和逻辑联系,因此在对混淆和加壳代码进行相似性比较时存在较大的局限性.

动态行为相似性比较则通过调试工具或系统监控工具对恶意代码的操作进行监控,在运行程序的同时观察其状态和执行流程的变化,获得行为数据进行相似性比较.动态行为相似性比较具有不受代码混淆和加壳的干扰,可以正确地提取出代码的执行流程的优点.对代码进行动态分析的典型工具有 OllyDbg,SofICE 以及 WinDbg 等.由于恶意代码可以检测到调试器的存在而隐藏其真实行为,后来提出了 TTAnalyze[8],TEMU[9],Panorama[10]等具有良好透明性、能够进行指令级分析的行为分析平台.Bailey 等人[11]提出了使用系统状态改变描述恶意代码行为,并提取其中的特征行为序列比较其相似性的方法.Tony 和 Mody 等人[12]使用加权的字符串编辑距离方法,计算两个代码的行为差异.文献[11,12]中的方法提高了比较的准确性,但是由于计算文字距离针对于恶意代码行为之间的顺序,因此对于混淆代码的分析能力不强.Bayer 等人[3]通过动态污点传播提取行为关联数据进行相似性比较.该方法的局限性在于无法识别恶意代码中加入的垃圾调用,不适合于判断含有垃圾调用的恶意代码的相似性.

由于静态结构相似性比较方法的局限,恶意代码的相似性比较逐渐以动态行为比较为主.恶意代码行为之间联系的相似性比较方法是动态行为相似性比较的主要手段之一.针对当前的恶意代码行为相似性比较存在的描述方法针对行为顺序、抗干扰能力不强、算法复杂度高、需要人工干预等局限性,我们提出了基于依赖特征的恶意行为相似性比较方法.该方法通过动态污点传播分析恶意代码,提取其中的控制依赖关系和数据依赖关系,并对两种依赖关系进行预处理,消除恶意代码反制手段带来的干扰,提取行为轮廓和逻辑结构信息,在比较过程中使用前期比较结果作为指导引导后面的比较,提高比较准确性的同时也降低了复杂度.

## 2 问题描述

在本节中,我们以僵尸网络程序 SDBot 为例说明我们要研究的问题.图 1 显示了 SDBot 用于完成远程文件下载的一段代码,其中,图 1(a)所示为原始的代码,图 1(b)所示为经过混淆之后的代码.

为了便于说明,我们在不改变原有功能的情况下手工地对代码进行了如下混淆:

(1) 修改了缓冲区的大小,新的接收缓冲区只有原始代码的一半,相应的循环执行序列变为原来的两倍,导致行为序列长度增加,行为比较差异扩大;

(2) 在程序初始位置加入垃圾调用 CreateFile,在程序的循环中加入垃圾调用 ReadFile,通过加入垃圾调用改变代码的行为序列,增加序列比较时的差异调用个数,增大序列比较的误差;

(3) 调换 CreateFile 和 InternetOpenUrl 顺序的同时,使用 MapViewOfFile 函数序列等价替换 WriteFile 函

数,在不改变语义的情况下改变行为序列中的节点,通过替换函数,使针对函数名称的行为序列比较难以获得正确结果.

```
1:  char fbuff[512];
2:  ...
3:  fh = InternetOpenUrl(internetHandle, dl.url, NULL, 0, 0, 0);
4:  if (fh != NULL){
5:    f = CreateFile(dl.dest, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, 0);
6:    if (f < (HANDLE)1){
7:      return 0;
8:    }
9:    total = 1;
10:   start = GetTickCount();
11:   do{
12:     memset(fbuff, 0, sizeof(fbuff));
13:     InternetReadFile(fh, fbuff, sizeof(fbuff), &r);
14:     WriteFile(f, fbuff, r, &d, NULL);
15:     total = total + r;
16:     if (dl.update != 1)
17:       sprintf(threadDescriptions[dl.threadnum],
              "file download (%s - %dkb transferred)", dl.url, total / 1024);
18:     else
19:       sprintf(threadDescriptions[dl.threadnum],
              "update (%s - %dkb transferred)", dl.url, total / 1024);
20:   } while (r > 0);
21:   speed = total / (((GetTickCount() - start) / 1000) + 1);
22:   CloseHandle(f);
23:)
```

(a)

```
1:  char fbuff[128];
2:  ...
3:  f = CreateFile(dl.dest, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, 0);
4:  hFileMapping=CreateFileMapping(f,NULL,PAGE_READWRITE,0,0,NULL);
5:  pCurrentPointer=MapViewOfFile(hFileMapping,FILE_MAP_WRITE,0,0,0);
6:  temp=CreateFile("C:\\test.txt",GENERIC_READ,0,
                    OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);
7:  if (f < (HANDLE)1){
8:    return 0;
9:  }else{
10: ifh = InternetOpenUrl(internetHandle, dl.url, NULL, 0, 0, 0);
11: if (fh == NULL){
12:   return 0;
13: }
14: total = 1;
15: start = GetTickCount();
16: do{
17:   ReadFile(fh,fbuff,sizeof(fbuff),&r);
18:   memset(fbuff, 0, sizeof(fbuff));
19:   InternetReadFile(fh, fbuff, sizeof(fbuff), &r);
20:   memcpy(pCurrentPointer,fbuff,r);
21:   pCurrentPointer=(void*)((DWORD)pCurrentPointer+r);
22:   total = total + r;
23:   if (dl.update == 1)
24:     sprintf(threadDescriptions[dl.threadnum],
            "update (%s - %dkb transferred)", dl.url, total / 1024);
25:   else
26:     sprintf(threadDescriptions[dl.threadnum],
            "file download (%s - %dkb transferred)", dl.url, total / 1024);
27:   }while(r>0);
28:   speed = total / (((GetTickCount() - start) / 1000) + 1);
29:   CloseHandle(f);
30:)
```

(b)

Fig.1 SDBot code

图 1 SDBot 代码片段

由上面的例子我们可以看出,恶意代码常使用的混淆和等价替换等反制手段,容易实现并且具有明显的干扰分析效果.

以上方法使代码文字特征和行为序列特征发生变化,但代码的内部结构、行为逻辑和行为之间的依赖关系仍然保持相对的稳定.使用依赖关系进行相似性比较是目前的主要思路之一.图 2 所示为混淆前后代码产生的系统调用序列和依赖关系.为了更清楚地表示其中的变化,我们在两图中同时略去一些相同的依赖关系和编译器内联展开的 memset 函数.但是对于依赖关系的比较,同样存在被干扰和混淆的问题.在图 2 所示的代码依赖图中,由于混淆和等价替换等反制手段的使用,在不改变语义的条件下,代码的控制依赖范围、函数个数与名称发生改变,从而改变了图的结构.由于传统依赖图无法表示此类顺序可调换的系统调用,基于传统依赖图的相似性比较方法也不具备等价行为序列识别的能力,因此容易受到干扰而无法获得正确结果,需要相关手段消除依赖图中的混淆.此外,由于图相似性比较是一种 NP 完全问题[1],为了利用依赖图实现准确和高效的相似性比较,我们还需要降低比较的复杂度.

针对以上问题,本文的研究目标可以描述为:

(1) 需要设计一套描述方法,解决传统依赖图无法描述顺序可调系统调用的问题,使之在描述恶意代码行为之间依赖关系的同时,不受由于无关行为顺序改变而引起的依赖图结构变化影响;

(2) 需要提出数据处理算法去除依赖图中的混淆,需要去除由于记录循环操作和恶意代码使用垃圾调用、等价调用替换等反制手段对相似性比较过程产生的干扰;

(3) 需要提出一种恶意代码相似性比较算法,使用扩展依赖图比较恶意代码的相似性,在提高准确度的同时提高效率.

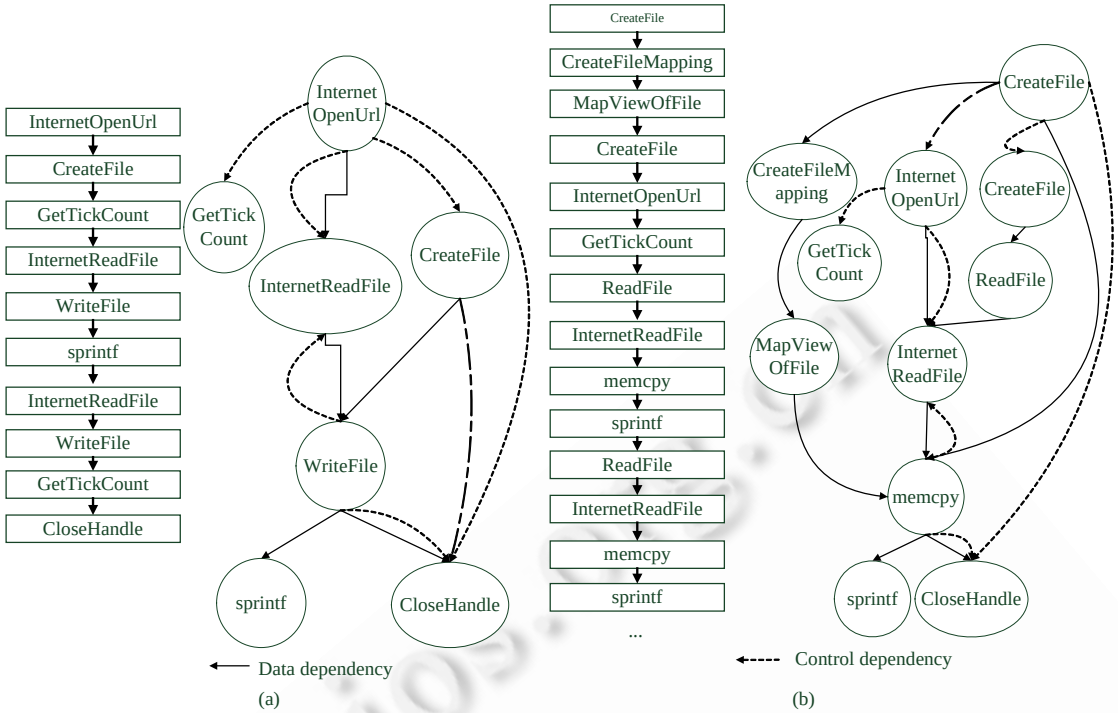



Fig.2 SDBot system call sequence and dependence graph

图 2 SDBot 代码的系统调用序列和依赖依赖图

## 3 分析方法

本节详细论述具体分析方法.我们的恶意代码相似性比较方法主要包括如下 3 个步骤:首先,使用动态污点传播分析构建带有虚拟节点的控制依赖图集合和扩展数据依赖图集合;然后,对生成的控制依赖图和数据依赖图进行预处理,消除混淆,缩减循环,构造行为轮廓等;最后,使用基于扩展依赖图的相似性比较算法,其输出结果是相同控制依赖和数据依赖中行为的个数与全部控制依赖和数据依赖中行为个数的百分比,以此作为恶意代码相似性判断的依据.

### 3.1 依赖图构建

依赖图构建是相似性比较的基础.我们扩展了传统依赖图,提出带有虚拟节点的控制依赖图和扩展数据依赖图.我们定义带有虚拟节点的控制依赖图如下:

带虚拟节点的控制依赖图 $G_{cd}=\{N_{cd},E_{cd},VN_{cd},Entry_{cd},Exit_{cd},Type_{cd},Count_{type}\}$,其中:$N_{cd}$ 为实节点,表示系统调用;$E_{cd}$ 为控制依赖边,表示控制依赖关系;$VN_{cd}$ 为虚拟节点,表示顺序无关的控制依赖关系;$Entry_{cd}$ 表示控制依赖图的入口点;$Exit_{cd}$ 表示控制依赖图的出口;$Type_{cd}$ 表示该依赖图中所有节点的系统调用类型统计信息;$Count_{type}$ 表示该图中各类型系统调用的个数.

本文中引入的带虚拟节点的控制依赖图与传统依赖图相比,增加了虚拟节点 $VN_{cd}$、统计信息记录 $Type_{cd}$ 和 $Count_{type}$.其中,增加虚拟节点主要是考虑到恶意代码中常常出现同一行为依赖于多个条件的情况.这种情况中,行为的触发与这些条件被满足的先后顺序无关,恶意代码作者常常调换这些条件的顺序来混淆代码.若使用传统控制依赖图描述这种情况,多个条件的控制范围按照代码执行的先后顺序层次包含,无法体现不同条件相互之间的顺序无关特性.因此,我们引入了虚拟节点 $VN_{cd}$ 表示这种顺序无关的控制依赖关系,去除条件之间的相互依赖,使之直接与触发的行为关联,在进行相似性比较时首先查找虚拟节点 $VN_{cd}$,对与虚拟节点相连的依赖条件进行无序比较,消除了传统依赖图在调换条件顺序后依赖边发生改变的干扰;为了快速产生变种并提高模块

的复用度,恶意代码模块的功能划分越来越清晰,模块之间耦合度不断减小,不同模块中各种行为的操作类型和频度往往出现很大的差别,为此,我们使用依赖图中各种系统调用的类型统计信息和各类调用的数量信息来构造依赖图的行为轮廓,通过首先比较依赖图的行为轮廓来快速地初步判定两个依赖图的相似性.为此,我们引入了行为统计信息记录 $Type_{cd}$ 和 $Count_{type}$.

类似地,定义扩展数据依赖图如下:

扩展数据依赖图 $G_{dd}=\{N_{dd},E_{dd},VN_{dd},Entry_{dd},Exit_{dd},Type_{dd},Count_{type}\}$.其中:$N_{dd}$ 为实节点,表示系统调用;$E_{dd}$ 为数据依赖边,表示数据依赖关系;$Entry_{dd}$ 表示依赖图的入口点;$Exit_{dd}$ 表示依赖图的出口;$Type_{dd}$ 表示该依赖图中所有节点的系统调用类型统计;$Count_{type}$ 表示该图中各类型系统调用的个数.与扩展控制依赖图相同,我们扩展了传统数据依赖图,加入了行为统计信息记录 $Type_{cd}$ 和 $Count_{type}$.与控制依赖图不同,数据依赖图由于节点之间必然存在引用关系,其次序不可调换,因而没有加入虚拟节点.

扩展依赖图集合的构建,通过可回溯的动态污点传播完成.可回溯污点传播方法将感兴趣的数据标记为污点,制定污点传播规则,根据指令计算污点状态的改变,在相关联的污点记录中建立联系以便回溯查找.污点状态记录以字节为单位进行描述,分为记录污点数据地址与范围的内存污点和记录污点寄存器状态的寄存器污点.污点传播是一种动态分析方法,传统动态分析方法不记录执行过的历史状态,但在构造依赖图的过程中需要判断当前节点与其前面节点的依赖关系,因而在污点传播过程中我们引入了回溯机制,记录传播路径并允许按照污点传播路径回溯恶意代码行为.

带有虚拟节点的控制依赖图构造过程包括初始节点的产生、实节点的添加、虚拟节点的添加、依赖边的添加和构造结束的判断.在初始状态下,我们在控制依赖图集合 $C$ 中创建带有虚拟节点的控制依赖图 $G_1=\varnothing$ 并将当前控制依赖图置为 $G_1$.在恶意代码执行过程中,我们标记感兴趣的内容为污点,将产生污点的系统调用作为初始节点 *Entry* 加入图 $G_1$,依赖图构造过程开始.

实节点的添加和控制依赖边的产生,通过判断当前系统调用所在的控制依赖范围来实现.由于控制流转移方向由标志寄存器决定,我们标记标志寄存器为污点.当污点影响控制流转移时,计算当前指令后必经节点获取控制依赖范围,将控制依赖范围内的调用作为实节点加入当前控制依赖图 $G_n$ 中(初始状态下,$n=1$).同时,以该控制流转移指令为起点回溯污点传播过程,在刚加入的实节点和回溯中遇到的节点之间添加控制依赖边,直到回溯抵达污点源.新添加的实节点存在两种情况:一种是系统调用,另一种是恶意代码的子函数.对于系统调用,只需添加实节点记录调用地址和参数;对于子函数,我们记录入口地址并标记节点为 $N_{G(n+1)}$,同时在控制依赖图集合 $C$ 中添加新控制依赖图 $G_{n+1}$,将当前控制依赖图置为 $G_{n+1}$,根据前面讲述的递归运算方法在 $G_{n+1}$ 产生新的节点并添加控制依赖边,直到该子函数返回.此时,恢复当前控制依赖图为 $G_n$,继续污点传播运算添加实节点和控制依赖边.

控制依赖图构建结束的判断通过判定污点传播终止实现,其结束条件包括 3 种情况:第 1 种是函数返回,函数内部控制依赖关系构造结束;第 2 种是函数内的污点漂白,依赖于该污点的控制依赖结束;第 3 种是程序执行结束.由于木马等程序常使用死循环获取和执行指令而难以正常退出,我们定义超时时间 $T$,当执行时间超过超时时间 $T$ 时,则终止控制依赖图的构建.

扩展数据依赖图的构造与带有虚拟节点的控制依赖图类似,其不同之处在于:首先,构造流程不同.由于污点传播表示数据之间的赋值和计算关系,因此构造数据依赖图时不需要判断数据依赖的范围,只需在添加新的节点后回溯污点传播过程并标记数据依赖边即可;其次,数据依赖图可跨越函数边界,因此其构造的结束条件仅有污点漂白和程序退出两种;最后,为了在控制依赖图添加虚拟节点需要记录数据依赖节点之间的数据状态,如果污点从节点 $A$ 流到节点 $B$ 的过程中未被修改,则标记 $AB$ 之间的依赖边为 *Stable*,否则为 *Changed*.

依赖图构建的最后一步是生成控制依赖图中的虚拟节点,虚拟节点表示控制依赖关系之间的顺序无关特性.节点的控制依赖关系即是当前节点的出边,控制集合包括当前节点所有出边连接的节点.我们定义实节点 $n$ 的控制集合为 $Control_n$.对于图 $G_{cd}$,从 *Entry* 开始遍历控制依赖图,计算路径上每个节点 $n\in N$ 的控制集合.每当新节点(设为 $n'$)的控制集合计算完成,则将该节点的控制集合和前面已遍历节点的控制集合相比较.如果

$n'\in Control_n$,并且有 $Control_n\backslash Control_{n'}=\{n\}$,可知节点 $n$ 与 $n'$的控制依赖范围重叠.此时,根据调用地址,在数据依赖图中查找 $n$ 和 $n'$之间的数据依赖.如果二者之间不存在数据依赖,或者存在数据依赖但该数据依赖边的状态为 *Stable*,则认为 $n$ 与 $n'$是顺序无关的控制依赖关系.在图中添加一个虚拟节点,并将与两个节点相连的控制依赖边都转移到新添加的虚拟节点上.同时,在两个节点和虚拟节点之间也构建一个控制依赖边.当节点遍历至 $Exit_{cd}$ 时,虚拟节点的构建结束.

依赖图集合构造完成的条件是代码执行结束或执行时间超过设定的超时时间 $T'$.在依赖图集合构建完成后,我们将恶意代码行为及其之间的关系表示为扩展依赖图.为了消除混淆手段带来的干扰并降低相似性比较的复杂度,在进行相似性比较前还需要对依赖图进行预处理.

### 3.2 依赖图预处理

为了消除混淆手段带来的干扰并降低相似性比较的复杂度,我们对依赖图进行正常化处理并构造依赖图的执行轮廓信息,其主要处理过程包括垃圾调用删除、循环缩减、等价行为序列识别、主分派函数识别和行为轮廓构建,过程如图 3 所示.

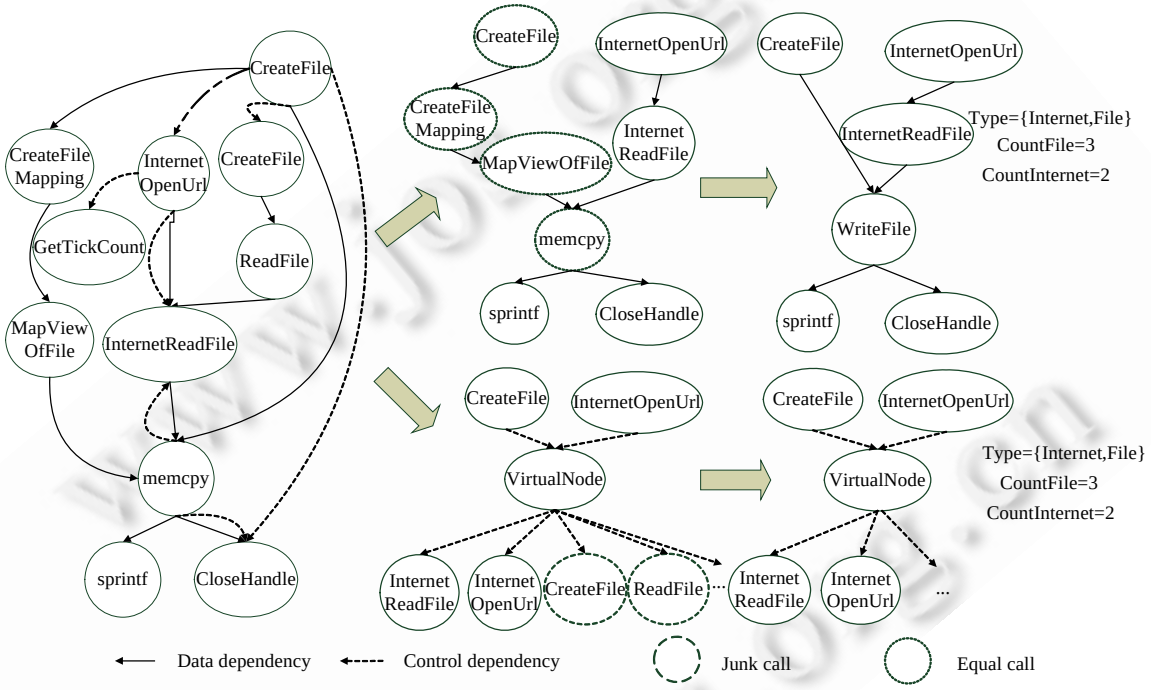



Fig.3 Pre-Processing procedure

图 3 预处理过程

首先,我们去掉依赖图中的垃圾调用.垃圾调用即从代码中移除后不改变整个代码功能的调用,其在污点传播中表现为产生了污点,但没有进行传播,或进行了传播但在传播过程中没有引起系统状态改变.与传统的尝试移除每一个调用后判定流程是否正确以识别垃圾调用的方法不同,我们根据恶意代码行为的逻辑关系进行判定.定义产生污点 $t$ 的指令为 $l_t$,漂白该污点的指令为 $l_b$,指令 $l$ 引用的数据集合为 $use[l]$,指令 $l$ 定义的数据集合为 $def[l]$,改变系统状态的函数集合 *OutputFunction*.如果$\neg\exists l'\in L$,使得 $use[l']\in Taint$ 和 $def[l']\in OutputFunction$ 同时满足,则路径 $L$ 上所有仅依赖于污点源 $t$ 的节点都是垃圾调用,可从图中删除.垃圾调用可能是单个系统调用也可能是一个子图.单个系统调用可直接从图中删掉其节点和边,如果该节点同时连接了虚拟节点,则计算删除该节点后与虚拟节点相连的剩余节点数;如果剩余节点数小于 2,则将虚拟节点一同删除,将原有指向虚拟节点的边指向剩余节点.如果是子图,则直接将该图从控制依赖图和数据依赖图中删除.在删除和合并节点时,我们

需要将控制依赖图和数据依赖图保持一致,即当在控制依赖图中删除节点时,需要同时根据函数地址删除数据依赖图中的对应节点.

其次,为了消除循环对依赖图构造和比较过程的干扰,我们进行循环缩减.具体地,我们通过创建执行指令的历史记录来缩减循环中的连续污点操作.采用文献[13]中的基于 DJ 图的分析方法识别循环,能否进行缩减则取决于使用的污点内存是否连续以及对内存进行的操作是否相同,我们通过比较传播路径上的操作行为和污点边界来加以判别.对每一条指令记录了污点操作范围信息 $TaintEdge=\{TaintAddress,TaintLength\}$,并假设该指令前一个节点为 $Pri(l_n)$.其归并算法为:

如果循环 $LoopA\subseteq L$,$\exists l_n\in LoopA$ 并且 $use[l_n]\in Taint$,同时由

$$Pri(l_n).TaintAddress+Pri(l_n).TaintLength=l_n.TaintAddress$$

可知,对于该污点的操作在循环中顺序进行.由于每次循环进行的操作相同,我们将污点操作节点合并为一个,合并污点操作范围如下:

$$(TaintAddress_{first},\sum_{n=1}^{l_n\in LoopA}l_n.TaintLength).$$

然后,识别等价行为序列.等价行为序列即为在代码中调用次序不同、函数名称不同而功能相同的行为序列.如图 1 中使用 CreateFileMapping 和 MapViewOfFile 替换 ReadFile.为了消除等价行为序列带来的干扰,我们手工构建了等价行为序列集合,该集合中分类保存相互等价的行为序列,并选择其中某一序列作为该类序列的代表.在分析中使用确定性有限状态自动机(DFA)匹配行为序列.当存在连续的污点传播过程使自动机到达完成状态时,则根据自动机表示的序列在等价序列集合中寻找统一表示的序列将其替换.

之后,识别主分派函数.主分派函数识别是为了判别恶意代码的行为结构.由于目前 IRC 协议仍然是僵尸和木马程序的主流控制协议[14],在僵尸或木马程序中往往存在着分析控制协议并执行相应操作的分派函数,该函数决定了代码的行为逻辑,在代码的变形和升级过程中保持稳定,因此,识别分派函数对于恶意代码的相似性比较和行为分析都具有重要意义.由于分派函数往往是针对网络污点数据使用大量的字符串处理类系统调用,并且在整个执行过程中不改变网络数据污点的内容,我们定义字符串处理类系统调用集合 $DealString=\{strcmp,strncmp,strcpy,strtok,\ldots\}$以及阈值 $y$,如果对于网络污点 $T_{net}$ 有节点路径 $L$,其中存在数目大于 $y$ 的污点传播行为序列 $l_{n_1},l_{n_2},l_{n_3},l_{n_4},\ldots,l_{n_i}$ 并且 $l_n\in DealString$,则认为该部分的控制依赖图为分派函数的控制依赖图.在该依赖图上设置标记,在控制依赖相似性比较和数据依赖相似性比较中优先进行比较.阈值 $y$ 可根据经验,也可使用其他方法,如使用训练集合的方法[15]等,进行设置,在我们的分析中,阈值 $y$ 设置为 5.

最后,建立行为轮廓.行为轮廓信息即系统调用的种类和每种行为的频度.我们将系统函数分为文件、注册表、网络、进程这 4 类操作.初始状态下,依赖图记录中的 $Type$ 和 $Count_{type}$ 都为空.从控制依赖和数据依赖的入口点开始遍历,分析访问到的节点,如果是虚拟节点,则继续遍历;如果是实节点,则判断节点类型,并将该函数所在的类别标志加入 $Type$ 记录中,同时增加表示该类函数调用个数的 $Count_{type}$ 计数.一直到所有节点都遍历完成为止.

经过依赖图预处理过程,我们去掉了图中可能存在的干扰,缩减了图的规模并构建了图的行为轮廓.下一步,根据处理过的控制依赖图和数据依赖图以及行为轮廓信息开展基于依赖关系的相似性比较.

### 3.3 相似性比较

我们的相似性比较过程基于扩展依赖图来进行,由于扩展控制依赖图带有虚拟节点,因此无法在其上直接使用传统的图相似性比较算法.在提出新的依赖图比较方法的过程中,注意到,传统的图相似性比较算法忽略了行为轮廓的差异和行为之间的联系,使用穷举式比较,复杂度较高.

针对以上问题,本文提出了基于扩展依赖图的相似性比较方法.其核心思想是,利用代码执行轮廓和结构信息引导扩展依赖图的比较过程,在提高准确度的同时降低复杂度.该比较过程针对于同类依赖图,具体分为两步:首先,根据分派函数相关信息和行为轮廓差异大小确定该类依赖图比较的先后次序;然后,在比较过程中寻找相同的系统调用作为比较的起点进行递归比较.

设两个样本产生的扩展依赖图集合分别为$\{C_1,D_1\}$和$\{C_2,D_2\}$,我们首先比较控制依赖图集合 $C_1$ 和 $C_2$,然后比较数据依赖图集合$D_1$和$D_2$.控制依赖图集合的比较分为两步:确定依赖图的比较顺序和比较集合中依赖图的相似性.依赖图比较顺序的确定是通过判定依赖图是否表示分派函数以及相互之间行为轮廓差异的大小来完成的,分为有分派函数标记和无分派函数标记两种情况.如果依赖图有分派函数标记,则最先进行比较.对于无分派函数标记的依赖图,我们定义符号 $ABS(Type[G']-Type[G''])$,表示两个图中行为类型的差值,$ABS(Count_{type}[G']-Count_{type}[G''])$表示两个图中同类行为频度的差值.计算所有 $G'\in C_1$ & $G''\in C_2$ 的 $ABS(Type[G']-Type[G''])$行为差值,并根据行为差值从小到大排列比较顺序.当两个图的 $ABS(Type[G']-Type[G''])$相同时,则根据 $ABS(Count_{type}[G']-Count_{type}[G''])$的值从小到大排序.比较顺序确定过程保证算法始终选取函数调用类型和个数差距最小的两个图进行比较.接下来,我们按次序比较控制依赖图.依赖图的比较以入口点 *Entry* 为起点.由于相同污点的传播过程构建了控制依赖图,可知两个图的 *Entry* 节点应当是相同的系统调用.在比较入口点相同后,以入口点为起点递归地遍历相连节点比较相似性.节点的比较分为实节点和虚拟节点两种情况.当 $N_{G_1}\in G_1, N_{G_2}\in G_2$ 并且 $N_{G_1}=N_{G_2}$ 时,如果存在 $N'\in G_1, N''\in G_2, N'$和 $N''$都是实节点且表示相同的系统调用,同时,连接 $N_{G_1}$ 和 $N'$以及连接 $N_{G_2}$ 和 $N''$的边同是出边或同是入边,则 $N'$和 $N''$等价.若其中有一个节点是虚拟节点,假设为 $N'$,若 $N''$也是虚拟节点,则比较与两个虚拟节点相连的实节点.若 $N''$是实节点,则返回到 $N''$上一级节点 $N_{G_2}$,遍历 $N_{G_2}$ 的边,查找与之相连的虚拟节点.若存在,将其与 $N'$比较;若不存在,则将与 $N'$相连的所有节点按照实节点比较方法与 $N''$进行比较.在比较节点等价后,我们将两个节点都标记为 *visited*,以当前的等价节点为起点递归地遍历并和与其相连的非 *visited* 的节点进行比较.如果与某个节点相连的所有节点都为 *visited*,则该节点的递归计算完成,返回上一级节点.我们设置等价节点计数器 $Count_{equ}$,初始状态下计数为 0,每当有新的等价节点时,计数器就加 1.控制依赖图相似性比较算法的停止条件是其中一个图的所有节点都被标记为 *visited*.控制依赖图集合 $C$ 比较结束的条件是其中一个集合未比较依赖图的个数为 0.

数据依赖图的比较以代码的分派函数信息、控制依赖图相似性信息和行为轮廓信息为指导进行.与控制依赖图比较类似,该过程分为确定比较顺序和图的相似性比较两步.数据依赖图的比较分为 3 个优先级:首先比较的是与分派函数相关的数据依赖图,其次是与已匹配控制依赖图相关的数据依赖图,最后比较剩余的数据依赖图.通过比较依赖图中的函数地址,完成与分派函数相关和已匹配控制依赖图相关数据依赖图的识别.我们在依赖图的构建过程中记录了依赖图中的系统调用地址,根据控制依赖图中的系统调用地址在数据依赖图中查找相应的节点,认为包含相同地址系统调用的控制依赖图与数据依赖图相对应.若控制依赖图中有分派函数标记,则对应的数据依赖图具有最高的比较优先级;若无标记,则具有次高的优先级.剩余的数据依赖图则根据依赖图之间的行为类型差值和行为频度差值,从小到大进行比较.数据依赖图的相似性比较算法同样首先比较入口点 *Entry* 相同,而后设置 *Entry* 为当前节点,遍历当前节点连接的节点并递归地进行比较.数据依赖图节点等价判定方法为:当 $N_{G_1}\in G_1, N_{G_2}\in G_2$ 并且 $N_{G_1}=N_{G_2}$ 时,如果存在 $N'\in G_1, N''\in G_2, N'$和 $N''$表示相同的系统调用,同时,连接 $N_{G_1}$ 和 $N'$以及连接 $N_{G_2}$ 和 $N''$的边同是出边或同是入边,则 $N'$和 $N''$等价.可将两个节点都标记为 *visited*,并以这两个节点为起点,递归地比较其所有相连的非 *visited* 节点.我们同样设置了数据依赖图集合的等价计数器 $Count_{equ}$.若两个图中有一个图的所有节点都被访问到,或在其后的数据依赖图与当前数据依赖图的差距大于数据差异比例,则数据依赖图的比较结束.数据依赖集合比较完成的条件是其中一个集合未比较依赖图个数为 0.依赖图的相似性比较算法流程如下:

Require: Cacluate All Match Nodes and Edges

Ensure: $C\neq\varnothing$ && $D\neq\varnothing$

**While** ($LeftGraphCount(C_1)\neq\varnothing$) && ($LeftGraphCount(C_2)\neq\varnothing$) **do**

  **If** $GetDispatchFunction()$==true **then**

    $C_{source}\leftarrow Dispatch_{Source}$; //Compare dispatch Function

    $C_{dest}\leftarrow Dispatch_{dest}$;

    $CompareWithVirtualNode(C_{source},C_{dest})$;

```
            MarkCompared(C_source,C_dest);
        end if
        for Min(ABS(|C_source|−|C_dest|)) to Max(ABS(|C_source|−|C_dest|)) do
            if HasVirtualNode()==true && !Compared(C_source) && !Compared(C_dest) then
                CompareWithVirtualNode(C_source,C_dest);          //Compare Left Graphs
                    MarkCompared(C_source,C_dest);
            else
                CompareFromEntry(C_source,C_dest);
                    MarkCompared(C_source,C_dest)
            end if
        end for
    end while
    while (LeftGraphCount(D_1)≠∅) && (LeftGraphCount(D_2)≠∅) do
        if GetDispatchFunction()==true then
            D_source←DispatchDataDependencyGraph_source;          //Compare Dispatch Function
            D_dest←DispatchDataDependencyGraph_dest;
            CompareFromDispatch(D_source,D_dest);
            MarkCompared(D_source,D_dest);
        else if HasRelativeControlDependencyGraph()==true then
            GetRelativeDataDependencyGraph(D_source,D_dest);
            CompareFromEntry(D_source,D_dest);
            MarkCompared(D_source,D_dest);
        else
            for Min(ABS(|D_source|−|D_dest|)) to Max(ABS(|D_source|−|D_dest|)) do
                CompareFromEntry(D_source,D_dest);
                MarkCompared(D_source,D_dest);
            end for
        end if
    end while
```

在相似性比较算法的描述中,我们将相似性比较的顺序选择条件范围简化表示为 Min($ABS|C_{source}|-|C_{dest}|$)和 Max($ABS(|C_{source}|-|C_{dest}|)$).相似性比较输出的结果是两个样本的行为中存在的相同控制依赖关系和数据依赖关系与其依赖关系总和的比例.分析人员可以通过比较结果快速识别并构造代码的行为轮廓和数据操作逻辑,对于族类代码的特征提取和防御研究具有重要意义.同时,在相似性比较中识别出来的恶意代码分派函数和行为触发逻辑流程,可以作为基础信息协助分析木马和僵尸网络控制协议.

## 4 实验评估

### 4.1 系统概述

我们在恶意代码动态分析平台 Wookon 的基础上设计并实现了原型系统.Wookon 是一套基于硬件模拟器 QEMU[16]的恶意代码分析平台,可在其上运行 XP 操作系统并获取进程代码执行的各种信息.原型系统由基于硬件虚拟化的虚拟执行环境、指令与 API 分析模块、污点传播引擎和相似性比较引擎等多个模块组成,其整体架构如图 4 所示.我们扩展了虚拟 CPU,增加了用于识别当前进程的指令,提取 CPU 寄存器的内容.我们扩展了虚拟内存,加入了影子内存用于监视内存数据的读/写.分析控制模块按照监控配置实时分析 CPU 状态,为污点

传播引擎采集数据.污点传播引擎根据指令执行信息计算污点传播过程,回溯构建扩展依赖图.相似性比较引擎通过比较依赖图集合,获得样本之间的差异并输出结果.

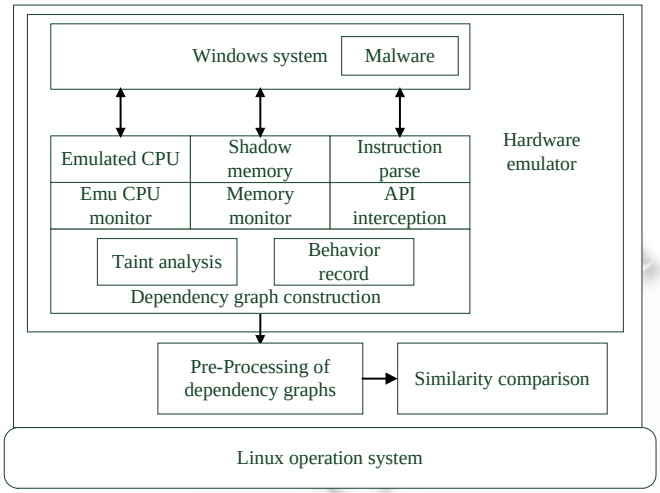



Fig.4 Structure of prototype system

图 4 原型系统结构

为了达到分析目标,我们对 Wookon 系统进行了改造,实现了进程识别、系统调用拦截、单步指令分析和污点传播引擎.

由于 CR3 寄存器是进程的页表物理地址寄存器,对所有进程具有唯一性,我们使用进程的 CR3 寄存器实现进程的识别和标记.

通过判断系统函数的入口地址实现系统调用拦截.Wookon 系统在进程加载时分析进程上下文,查找加载的动态库导出表,将函数地址记录在函数列表中.在代码执行中比较虚拟 CPU 的当前 EIP 和函数列表的入口地址判断是否为系统调用,若是,则读取堆栈中的参数.该方法不对系统和程序进行任何修改,具有良好的隐蔽性和透明性.

为了提高透明性并减少单步执行对整体效率的影响,我们在 Wookon 系统中添加了虚拟单步标志,该标志可被硬件模拟器识别,但并不影响虚拟 CPU 的单步标记.

为了进行污点传播,我们为进程内存创建了影子内存.该影子内存记录内存状态信息,所映射的区域包括:进程映射进内存后的文件、进程在堆中分配的内存、进程栈等.影子内存中记录污点内存的相关信息包括:内存所在的位置、以字节计算的长度、内存的类型、内存当前的状态、是否为寄存器、寄存器状态等.

我们在系统中整合了反汇编引擎 udis86[17]、反汇编的每一条执行指令,并记录了该指令执行时的 CPU 各项内容.通过分析反汇编后的代码进行污点传播运算.

### 4.2 实例分析

本文的实验在 DELL Optplex360 3.2G Duo core CPU,3G 内存,160G 硬盘的 Linux 主机上进行.恶意代码样本在虚拟环境的 Windows XP 操作系统中运行,我们使用桥接模式将虚拟系统接入互联网,隐蔽底层主机,在分析过程中拦截了字符串、文件、注册表和网络相关的函数,手工编写了污点传播规则.

我们以 SDBot 和 Bagle 恶意代码样本来进行实验.SDBot 是一个典型的基于 IRC 协议的僵尸程序,有数量繁多的各种变种.Bagle 是一个具有大量变种的蠕虫程序,由于其能够快速地产生变种并广泛加以传播,安全公司 McAfee 将其变种的危害等级评估为最高.我们对 SDBot 源码进行手工变形处理来验证算法对于加壳和代码混淆的比较效果.Bagle 蠕虫样本则取自网络恶意代码样本库 VXHeavens[18],该样本库是国内外恶意代码研究

(如文献[19,20])的重要样本来源.由该网站获取样本以测试我们的算法在真实环境中对于真实恶意代码样本的分析效果.由于恶意软件的特殊性,没有可供参照的协议规范和说明文档,通过人工分析,我们来确认结果的准确性.

我们分析了 SDBot 僵尸程序的 5 个变种样本.其中:SDBot.b 直接由源码编译得来,作为比较的基准;其余样本通过添加垃圾调用、调换行为顺序、替换等价行为序列,并使用加壳软件对混淆后的代码进行处理.加壳软件分别为 UPX,EXECryptor,ASProtect.三者分别是压缩壳、加密壳和综合混淆壳的代表.其中,ASProtect 具有极高的加壳强度,在没有预备知识的情况下很难手工分析.对混淆和加壳后的代码进行分析,对于验证本文提出的相似性比较方法的准确度和有效性具有较大的意义.

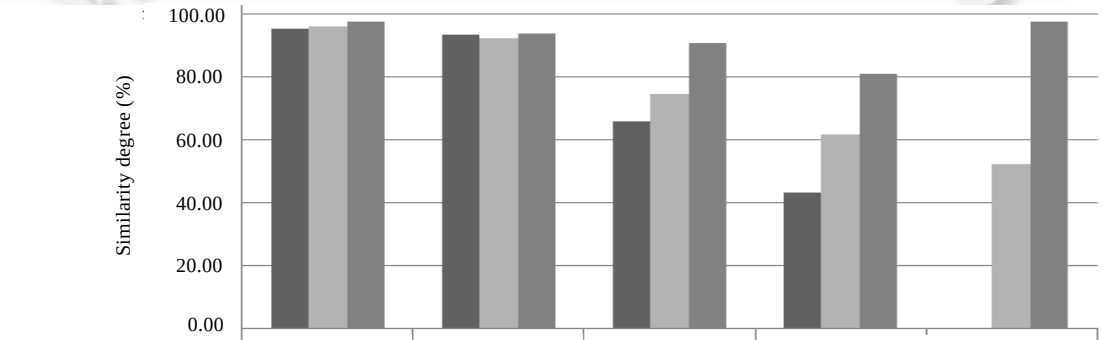
从表 1 可以看出,使用加壳软件处理后的代码产生了大量的新基本块,分析过程时间复杂度和空间复杂度都远远大于未加壳代码.其中,以 Asprotect 最为典型,带有该壳的代码具有最高的时间复杂度和空间复杂度,生成 634MB 的执行记录,是正常执行记录的 90 倍;19 分钟 35 秒的分析时间,是基准样本分析时间的两倍.从表中依赖边数的变化我们可以看到,加壳对于控制依赖的影响大于对数据依赖的影响,如使用 Asprotect 加壳后,代码控制依赖图增加到 317 个.经手工分析后确认,这是由于在加壳软件中包含了反制调试和监控的代码,这些代码产生了大量控制依赖,但是在垃圾调用删除阶段被消除,对于最终结果并未产生影响.

**Table 1** Anaylsis result of SDBot

**表 1** SDBot 样本分析记录

| Name | Pack tool | Record size (MB) | Basic block count | CDG count | DDG count | Removable graph count | Analyzing time |
|---|---|---|---|---|---|---|---|
| SDBot.b | No | 7 | 807 | 286 | 36 | 3 | 10m50s |
| SDBot.UPX | UPX | 116 | 858 | 303 | 36 | 20 | 11m21s |
| SDBot.EXECryptor | EXECryptor | 310 | 1 215 | 312 | 38 | 27 | 13m34s |
| SDBot.Asprotect | Asprotect | 634 | 940 | 317 | 37 | 32 | 19m32s |
| SDBot.Manual | Manual | 9 | 850 | 287 | 36 | 4 | 14m6s |

为了与其他方法进行比较,我们同时实现了文献[7]中的基于基本块的比较以及文献[2]中的基于系统调用的比较方法.图 5 是采用 3 种方法对 SDBot 样本的相似性比较结果.从图中可以看到,我们的方法对于经加壳处理的样本具有较好的结果.这是因为代码被混淆和加壳后,其代码特征和行为序列特征发生较大的变化,但由于其基本功能和执行逻辑未发生改变,因此依赖关系特性仍然具有较高的相似度.




| | SDBot.b(*) | SDBot.UPX | SDBot.EXECryptor | SDBot.Asprotect | SDBot.Manual |
|---|---|---|---|---|---|
| Structual similarity | 95.63% | 93.70% | 66.25% | 43.63% | 0.11% |
| Behavioral similarity | 96.40% | 92.30% | 74.63% | 62.01% | 52.32% |
| Dependency similarity | 97.65% | 93.79% | 90.81% | 80.98% | 97.79% |

Fig.5 Result of similarity comparison on SDBot samples

图 5 SDBot 样本相似性比较结果

为了进一步评估我们的方法的有效性和准确性,还采用同样的方法对 Bagle 样本进行分析.与 SDBot 不同,Bagle 样本由网络恶意代码样本数据库[18]直接获取,没有进行任何人工处理.为了清晰地表示实验结果,我们

使用外壳识别工具 FileInfo[21]对恶意代码是否加壳以及是何种外壳进行了分析.由于与 SDBot 相比,Bagle 功能较为简单,因此产生的控制依赖和数据依赖规模相对于 SDBot 要小很多.

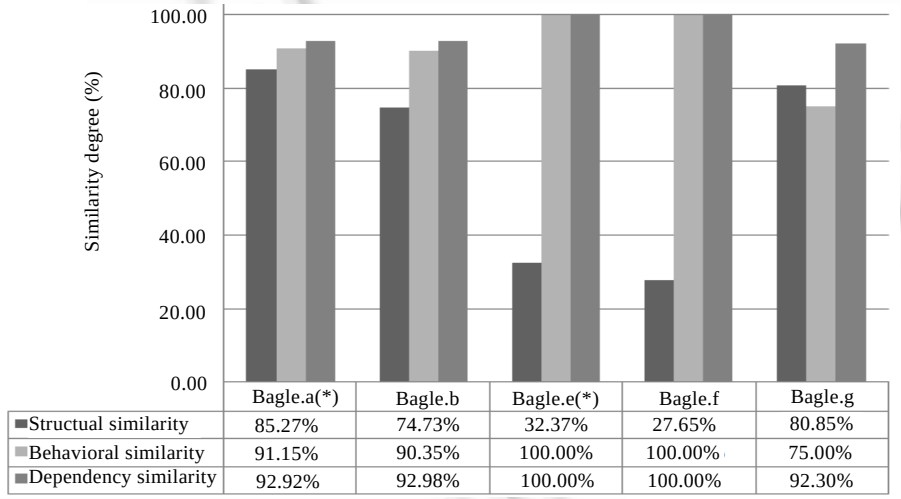
由表 2 可以发现,Bagle 样本获得的系统调用数量和依赖关系规模远远小于 SDBot 样本,通过手工分析 Bagle 代码可以发现,Bagle 的作者在样本中手工实现了部分系统函数的功能.它将文件映射进内存,通过手工编写的字符比较代码,而非使用系统函数进行处理.从样本分析数据可以看出,此类自行编写代码替换操作系统调用的做法对我们的方法没有太大的影响;但是,如果 Bagle 作者大量手工实现系统函数以减小代码的外部行为,可能会使我们的比较方法产生较大的误差.因此,研究基于系统调用依赖关系和代码语义的比较方法,将是我们未来工作的方向.

**Table 2** Analysis result of Bagle

**表 2** Bagle 样本分析记录

| Name | Pack tool | Record size (MB) | Basic block count | CDG count | DDG count | Removable graph count | Analyzing time |
|---|---|---|---|---|---|---|---|
| Bagle.a | 无 | 87 | 141 | 28 | 29 | 0 | 7m31s |
| Bagle.b | UPX | 99 | 186 | 51 | 29 | 21 | 6m20s |
| Bagle.e | PeX | 130 | 141 | 16 | 2 | 4 | 7m11s |
| Bagle.f | PeX | 160 | 141 | 16 | 2 | 4 | 6m22s |
| Bagle.g | PeX | 137 | 141 | 13 | 2 | 1 | 5m49s |

与 SDBot 样本不同,我们分析的 Bagle 代码样本并非两两之间都具有较高的相似度.其中,Bagle.a 与 Bagle.b 相互之间相似度较高,后缀为 e,f,g 的样本两两相似度较高.我们将其分为两类,在两类之间进行比较时相似度很低,具体比较结果如图 6 所示,其中,(*)表示该类别中作为参照的样本.如图 6 所示,其中一个比较反常的情况是,同属一类的 Bagle.e 和 Bagle.f 的代码结构相似性程度较低.通过手工分析发现,代码在变种过程中大量基本块的大小和指令发生改变,验证了该数据的正确性.




| | Bagle.a(*) | Bagle.b | Bagle.e(*) | Bagle.f | Bagle.g |
|---|---|---|---|---|---|
| Structual similarity | 85.27% | 74.73% | 32.37% | 27.65% | 80.85% |
| Behavioral similarity | 91.15% | 90.35% | 100.00% | 100.00% | 75.00% |
| Dependency similarity | 92.92% | 92.98% | 100.00% | 100.00% | 92.30% |

Fig.6 Result of similarity comparison on Bagle samples

图 6 Bagle 样本相似性比较结果

在验证比较准确性的同时,还需要验证算法对于不同种类代码的误报率的大小,我们将 SDBot 样本的依赖图集合与 Bagle 样本的依赖图集合进行了对比,其相似度为 0.该结果说明,我们的算法对于不同种类的代码误报率很低(如图 7 所示).

在分析实验中,Bagle 样本在真实机器上的运行时间在 2 秒以内.而我们的分析平台上进程测试和分析的平均执行时间为 6 分 39 秒,是真实执行时间的 200 倍.与其他方法的分析时间相比,我们的方法时间复杂度略高,具体数据如图 8 所示.基于系统调用的相似性比较算法和基于代码结构的相似性比较算法,其时间复杂度约为

我们算法的 20%~50%.本文所提算法仍有进一步优化的空间,该方法是一种事后的离线分析方法,对实时性要求不高,现有的性能不影响该算法的实际应用.目前,分析过程的大部分时间损耗在虚拟执行上,当前的原型系统实现并没有对此进行优化,将来可以通过升级硬件和优化虚拟机执行来提高分析效率.并且,我们的方法尽管相对于单个代码的执行时间具有较高的时间复杂度,但是对于高度混淆并使用 ASProtect 加壳的样本分析也在 20 分钟以内,不需要人工干预并且具有较高的准确度.因此,对于恶意代码分析和研究而言,该方法是一种具有实用价值的相似性比较方法.

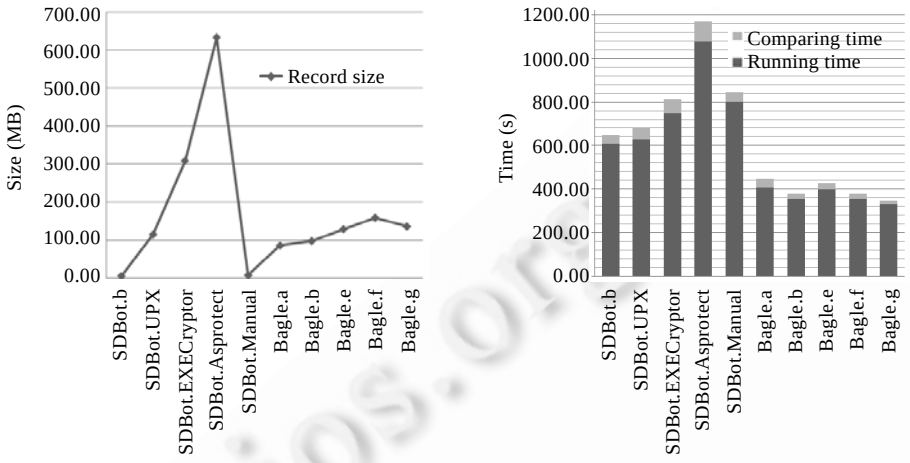



Fig.7 Time and space complexity

图 7 比较过程中的空间和时间复杂度

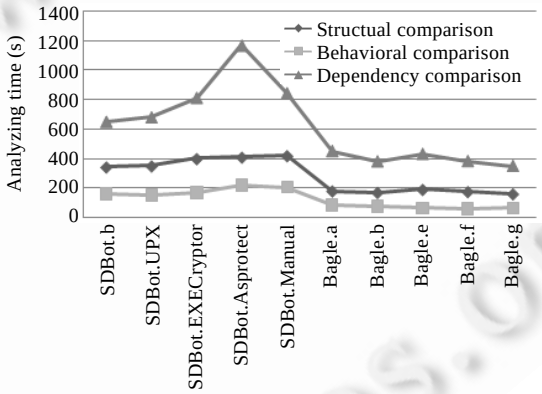



Fig.8 Time complexity compare

图 8 与其他相似性算法的时间复杂度比较

综上所述,分析实验结果表明,使用基于依赖关系的相似性比较算法对恶意代码相似性进行比较,尤其是对经过混淆和加密后代码的比较,具有较高的准确性.虽然该方法由于虚拟单步执行造成了时间效率的降低,但是目前的分析时间仍然在可接受的范围内,并且该问题会随着硬件的升级和虚拟机的进一步优化得到解决.

## 5 总 结

本文提出了一种基于行为依赖特征的恶意代码相似性比较方法,该方法通过构造带有虚拟节点的控制依赖图和扩展数据依赖图进行相似性比较.使用了预处理算法消除混淆并构建依赖图的行为轮廓,以图的行为轮

廓和结构信息引导依赖图的比较.实验结果表明,该方法可以有效去除由于混淆、加壳等反制手段引起的干扰,具有较高的准确性.

我们下一步工作将专注于提取代码的语义,与系统调用信息结合进行相似性比较.同时,本文提出的方法受到动态分析每次只能分析一条路径的限制.在接下来的研究中,我们将尝试使用符号执行方法判断路径可达性,实现路径遍历以更好地进行相似性比较.另外,由于恶意代码变种的不断出现,大量未知恶意代码对系统安全提出了挑战,未知恶意代码检测成为了研究热点和难点.在后续工作中,我们将尝试根据恶意代码的相似性提取其依赖特征,以此为基础检测恶意代码变种和未知恶意代码,并尝试使用相似性比较的方式进行未知恶意代码的检测实时预警.

**References**:

[1] Microsoft security intelligence report. 2007. http://www.microsoft.com/downloads/details.aspx?FamilyID=4EDE2572-1D39-46EA-94C6-4851750A2CB0

[2] Wang Z, Pierce K, McFarling S. BMAT—A binary matching tool for stale profile propagation. The Journal of Instruction-Level Parallelism, 2000,2:23–43.

[3] Bayer U, Comparetti PM, Hlauscheck C, Kruegel C, Kirda E. Scalable, behavior-based malware clustering. In: Proc. of the Network and Distributed System Security Symp. (NDSS). San Diego, 2009. http://www.isoc.org/isoc/conferences/ndss/09/proceedings.shtml

[4] Flake H. Structural comparison of executable objects. In: Proc. of the Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2004). 2004. 83–97.

[5] Dullien T, Rolles R. Graph-Based comparison of executable objects (English version). In: Proc. of the SSTIC 2005. 2005. http://www.sstic.org/2005/programme/

[6] Rabek JC, Khazan RI, Lewandowski SM, Cunningham RK. Detection of injected, dynamically generated, and obfuscated malicious code. In: Staniford S, Savage S, eds. Proc. of the 2003 ACM Workshop on Rapid Malcode. New York: Association for Computing Machinery, 2003. 76–82. [doi: 10.1145/948187.948201]

[7] Gao DB, Reiter MK, Song D. Binhunt: Automatically finding semantic differences in binary programs. In: Proc. of the Int'l Conf. on Information and Communications Security. Berlin, Heidelberg: Springer-Verlag, 2008. 238–255. [doi: 10.1007/978-3-540-88625-9]

[8] Bayer U, Moser A, Kruegel C, Kirda E. Dynamic analysis of malicious code. Journal in Computer Virology, 2006,2(1):67–77. [doi: 10.1007/s11416-006-0012-2]

[9] TEMU. http://bitblaze.cs.berkeley.edu/temu.html. 2008.

[10] Yin H, Song D, Egele M, Kruegel C, Kirda E. Panorama: Capturing system-wide information flow for malware detection and analysis. In: Ning P, ed. Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: Association for Computing Machinery, 2007. 116–127. [doi: 10.1145/1315245.1315261]

[11] Bailey M, Oberheide J, Andersen J, Mao ZM, Jahanian F, Nazario J. Automated classification and analysis of internet malware. In: Kruegel C, Lippmann R, Clark A, eds. Proc. of the 10th Int'l Conf. on Recent Advances in Intrusion Detection. Berlin, Heidelberg: Springer-Verlag, 2007. 178–197. [doi: 10.1007/978-3-540-74320-0_10]

[12] Lee T, Mody JJ. Behavioral classification. 2006. http://www.microsoft.com/downloads/details.aspx?FamilyID=7b5d8cc8-b336-4091-abb5-2cc500a6c41a\&displaylang=en

[13] 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]

[14] Zhuge JW, Han XH, Zhou YL, Ye ZY, Zou W. Research and development of botnets. Journal of Software, 2008,19(3):702−715 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/702.htm [doi: 10.3724/SP.J.1001.2008.00702]

[15] Lee W, Stolfo SJ. Data mining approaches for intrusion detection. In: Rubin A, ed. Proc. of the 7th Conf. on USENIX Security Symp., Vol.7. Berkeley: USENIX Association, 1998. 6.

[16] Bellard F. QEMU, a fast and portable dynamic translator. In: Pai V, ed. Proc. of the USENIX Annual Technical Conf. Berkeley: USENIX Association, 2005. 41−46.

[17] Udis86. http://udis86.sourceforge.net/

[18] VXHeavens. http://www.netlux.org

[19] Kolter JZ, Maloof MA. Learning to detect malicious executables in the wild. In: Kim W, Kohavi R, eds. Proc. of the 10th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. New York: Association for Computing Machinery, 2004. 470−478. [doi: 10.1145/1014052.1014105]

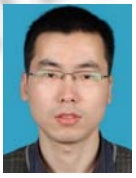
[20] Zhang QH, Reeves DS. Meta aware: Identifying metamorphic malware. In: Proc. of the Annual Computer Security Applications Conf. (ACSAC). 2007. 411−420.

[21] FileInfo. http://www.pediy.com

**附中文参考文献:**

[14] 诸葛建伟,韩心慧,周勇林,叶志远,邹维.僵尸网络研究.软件学报,2008,19(3):702−715. http://www.jos.org.cn/1000-9825/19/702.htm [doi: 10.3724/SP.J.1001.2008.00702]

**杨轶**(1982－),男,河南鹤壁人,博士生,主要研究领域为恶意代码分析与防范.



**应凌云**(1982－),男,博士生,主要研究领域为僵尸网络分析.

**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.

**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.