第29卷 第9期
2006年9月
计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS
Vol. 29 No. 9
Sept. 2006

基于可执行文件静态分析的入侵检测模型

苏璞睿 杨 轶

(中国科学院软件研究所信息安全国家重点实验室 北京 100039)

摘 要 基于进程行为的入侵检测技术是主机防范入侵和检测恶意代码的重要技术手段之一.该文提出了一种基于可执行文件静态分析的入侵检测模型,该模型通过对应用程序可执行文件的静态分析,建立应用程序所有可能执行的定长系统调用集合,通过实时监控进程执行的系统调用序列是否在该集合中实施检测.该模型不需要源文件、大规模训练数据,通用性和易用性好;在应用程序可执行文件完整的情况下,误报率为0,抵抗模仿攻击的能力更强,漏报率更低.

关键词 入侵检测;系统调用;静态分析

中图法分类号 TP309

Intrusion Detection Model Based on Executable Static Analysis

SU Pu-Rui YANG Yi

(*State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences, Beijing* 100039)

Abstract Intrusion Detection based on process' behaviors is one of the mainstream techniques for defend against intrusion and malicious code. In this paper, an intrusion detection model based on executable static analysis has been brought forward. The model statically analyzes the executable files of the application to construct the set of all the possible N-length system call sequences. When monitoring in real time, it splits the system call sequence the process triggered into N-length sequences by N-length slide window. If there is one in the N-length sequences not in the set, the process is marked as intrusive. The model needs not source code or large numbers of training data, and is much more universal and applicable. When the executable files of the application are complete, the rate of false positive is 0. It's much stronger for defending against mimicry attacks and its rate of false negative is much less.

Keywords intrusion detection; system call; static analysis

1 引 言

各类恶意代码的泛滥已经严重威胁到信息系统的安全,目前主要依赖于防病毒软件和入侵检测系统等手段实施检测和防范.目前的防病毒软件和入侵检测系统多通过特征匹配实施检测,而各类攻击或病毒变种的出现以及代码模糊变换(code obfuscation)等技术的应用,使得这些检测手段的作用越来越有限.基于行为的检测与防范技术已成为新的研究热点[1].

1996年,Forrest等人提出的基于系统调用的

收稿日期:2006-04-05;修改稿收到日期:2006-06-01.本课题得到国家"九七三"重点基础研究发展规划项目基金(G1999035802)、国家杰出青年基金项目(60025205)和国家自然科学基金(60273027)资助.苏璞睿,男,1976年生,博士,助理研究员,主要从事入侵检测、网络安全等方面的相关研究.E-mail:supurui@is.iscas.ac.cn.杨 轶,男,1982年生,硕士研究生,研究方向为网络安全.

入侵检测模型——N-gram 模型是基于进程行为的入侵检测技术的典型代表[2,3]. 由于系统调用在操作系统中的关键作用,进程系统调用相关属性是用于描述进程行为、实施入侵检测的重要数据源之一. 受 Forrest 等人工作的启发,后续出现了大量优秀成果,包括 Var-gram 模型[4]、FSA 模型[5]、Vt-Path 模型[6]等等. 但这些系统仍无法满足现实系统保护的需要,主要在进程行为建模、检测准确性、检测能力、系统实用性方面仍需进一步完善.

本文提出了一种基于可执行文件静态分析的进程行为建模方法,通过静态分析应用程序可执行代码,建立进程运行过程中可能的系统调用序列集合,以该集合为基础,对进程实施监控,进程执行过程中,出现任何不在该集合中的系统调用片断均认为发生入侵. 该模型主要有以下特点:(1)分析过程不依赖于任何源程序,可对各类应用程序实施监控;(2)不会出现任何误报;(3)具有较强的抵抗"模仿攻击"(mimicry attack)的能力.

本文将首先介绍该模型的基本思想、应用程序系统调用图的生成方法、应用程序所有可能执行的定长系统调用序列集合的生成方法和实时检测算法;然后,介绍部分实验结果,并对系统的检测能力、抗"模仿攻击能力"进行分析;最后将总结全文,并分析下一步工作.

2　检测模型

异常检测的前提是准确地定义正常行为,目前对于应用程序的正常行为定义生成主要分为两种思路:大量的数据训练生成和静态分析方法. 采用大规模数据训练的方法主要问题在于搜集完备的训练数据是困难的,这就造成在实施检测时,为了降低误报率,不能采取严格的检测措施. 采用静态分析的方法主要是通过静态分析应用程序相关数据建立应用程序行为定义,这一类型的主要代表是 Call Graph 模型和 Abstract Stack 模型,这两个模型主要是通过分析应用程序源代码建立有限自动机来定义应用程序行为. 其主要问题表现在以下方面:(1)需要应用程序源程序,通用性不强;(2)依赖于指令地址定位状态,未能解决消息处理、动态链接库等问题;(3)采用非确定性有限自动机效率低.

本文提出了基于可执行文件静态分析的检测模型,其基本思想是通过分析应用程序的可执行文件(包括相关的动态链接库文件),建立应用程序所有可能执行的定长 L 的系统调用序列集合,在实时监控中,当进程执行的系统调用序列不在该集合中时,我们认为系统受到入侵. 其总体结构如图 1 所示.

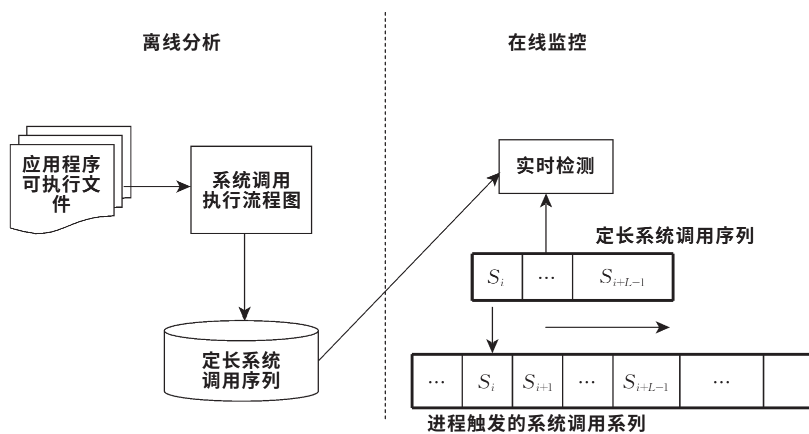


图 1　模型总体结构

2.1　系统调用图生成

在生成一个应用程序的系统调用图之前,我们首先需要分析应用程序的流程图. 一个应用程序的操作是由一序列的函数来完成的,分析一个应用程序的流程图是通过分析一序列的函数流程图完成的.

函数由几种典型的代码结构组成:函数调用(CALL)、IF-THEN-ELSE 结构、LOOP 结构、GOTO 结构等等. 函数调用涉及程序执行流程的指令包括 CALL 和 RET,而 IF-THEN-ELSE,LOOP,GOTO 等结构最终均转换为 JMP,JNZ 等跳转指令. 因此,通过对应用程序中 CALL,RET,JMP,JNZ 等指令的跟踪和分析,可建立应用程序的系统调用图.

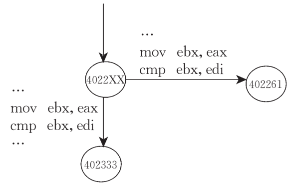
定义 1.　引用点(Referenced Point). 当应用程序 A 中指令 I 的地址 $addr_I$ 是程序中任意一个跳转指令跳转的目标地址时,我们称 $addr_I$ 是应用程序 A 的一个引用点.

JMP 跳转指令的目标地址是主要的引用点. 另外,子函数执行完成之后,从子函数代码返回执行函数调用后的下一条指令,在应用程序分析过程中,函数调用 CALL 指令的下一条指令我们也看作一个引用点.

分析函数 F 的流程图时,首先为函数 F 建立所有的状态空间,包括初始状态 S_0,退出状态 S_E 和为每一个引用点 r 建立一个状态 S_r,即函数 F 的所有状态集合 $S=\{S_0,S_E\}\cup\{S_r \mid r$ 是函数 F 的一个引用点$\}$.

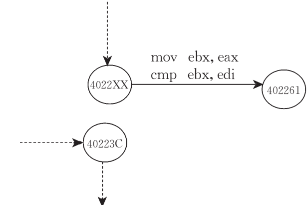
从函数的入口地址顺序扫描所有可执行代码,根据 CALL、JMP、JXX(指除 JMP 以外的跳转指令)、RET 四种类型的指令建立函数的执行流程图. 以下是四种跳转过程的跳转图构建方法示例(示例代码均来自于 Windows Internet Explorer 6.0,更新版本 SP2),参见图 2,详细算法参见附录.

```
.text:00402236    mov              ebx, eax
.text:00402238    cmp              ebx, edi
.text:0040223A    jnz              short loc_402261
…
.text:00402333    loca_402333 mov  ebx,eax
…
```



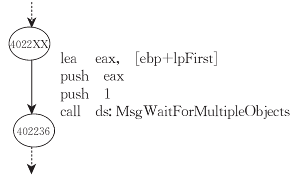
(a) 指令JXX跳转图构建

```
.text:00402236    mov              ebx, eax
.text:00402238    cmp              ebx, edi
.text:0040223A    jmp              short loc_402261
.text:0040223C    loca_40223C      mov ebx,eax
…
```



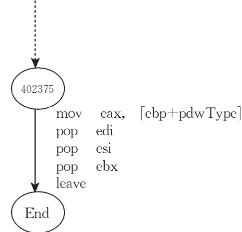
(b) 指令JMP跳转图构建

```
.text:0040222A    lea              eax, [ebp+lpFirst]
.text:0040222D    push             eax
.text:0040222E    push             1
.text:00402230    call             ds:MsgWaitForMultipleObjects
.text:00402236    mov              ebx, eax
.text:00402238    cmp              ebx, edi
```



(c) 指令CALL跳转图构建

```
.text:00402375    loca_402375:
.text:00402375
.text:00402375    mov              eax, [ebp+pdwType]
.text:00402378    pop              edi
.text:00402379    pop              esi
.text:0040237A    pop              ebx
.text:0040237B    leave
.text:0040237C    retn             10h
```



(d) 指令RET跳转图构建

图 2　四种跳转指令分析方法图

JMP 指令强制跳转,由其所在状态强制跳转到目标地址,其它跳转指令 JXX 有可能跳转到目标地址也有可能继续执行. 程序在任意位置执行 RET 指令,均将跳转到函数结束状态 End.

理解应用程序的语义是复杂的,分析函数的执行流程主要是为了分析系统调用执行情况,因此,流程图中的转换中如果不包含任何函数调用,则将其视为空操作 ε. 这样,可将函数 F 的流程图转换为只包含空操作 ε 和函数调用作为转换的函数调用图.

一般情况下,函数调用图中会包含大量的空操作 ε,为了便于后续分析,需要消除图中的空操作. 对于任意状态 S_r,如果其到状态 S_{r+1} 存在一个空操作转换 ε_0:

(1) 当且仅当状态 S_r 仅有一条输出 ε_0,删除状态 S_r,并将 S_r 的所有输入边增加到 S_{r+1};

(2) 当 S_r 到状态 S_{r+1} 存在多个空操作,将其合并为 1 个空操作转换;

(3) 当状态 $S_r=S_{r+1}$ 时,去掉边 ε_0.

为了分析应用程序的系统调用执行流程,需要以应用程序各个函数的系统调用图为基础构建应用程序的流程图,主要是对各个函数中的函数调用操作进行分解和替换,方法如图 3 所示.

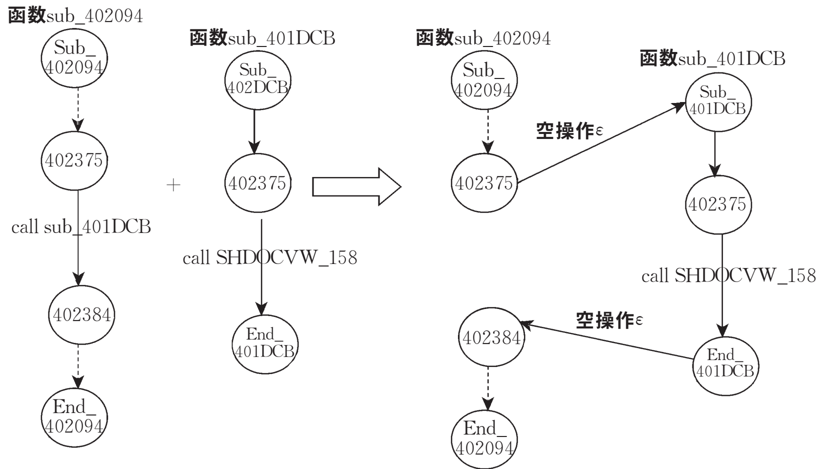


图 3 函数系统调用图合并过程

这里的转换过程主要是针对应用程序自身定义的函数和部分库函数，对于系统调用不作转换. 合并完成之后，再次对获得的图做消除空操作处理，最后得到以主函数初始状态为入口、主函数结束状态为结束状态的应用程序系统调用流程图. 在合并过程中，可能因为递归函数引入一些不可能序列（或不可能路径），但这并不直接影响检测的准确性，关于这一问题的详细分析请参见 3.2 节.

在 Windows 应用程序分析过程中，由于 Windows 系统采用消息处理机制，需要特殊处理. 应用程序中，消息处理部分先是注册一批消息处理函数，然后由操作系统负责调用相应的函数处理对应的消息. 通过分析应用程序可执行代码可以获得所有注册的消息处理函数和消息的分发点（DispatchMessageA）. 在分析消息过程中不考虑消息之间的依赖关系，其处理过程类似于 SWITCH 结构处理.

2.2 生成系统调用序列集

在获得应用程序系统调用图之后，可以以此图为基础分析该应用程序所有可能执行的定长为 L 的系统调用序列. 假设应用程序 A 所对应的系统调用图为 $Graph$，则应用程序 A 所有可能的定长为 L 的系统调用序列

$$SequencesOfGraph(Graph, L) = \bigcup_{state \in Graph} SequencesOfState(Graph, state, L),$$

其中 $SequencesOfState(S_r, Graph, L)$ 是以状态 $state$ 为起点，所有可能的长度 L 的系统调用序列. 利用以下算法可以获得 $SequencesOfGraph(Graph, L)$.

算法 1. 系统调用图的定长系统调用集合生成 $SequenceOfGraph(Graph, L)$.

输入：系统调用图 $Graph$，系统调用序列长度 L

输出：定长系统调用序列集合 S

1. $S := \varnothing$
2. for all $state$ in $Graph$ do
3. $\quad S := S \cup SequencesOfState(Graph, state, L)$
4. endfor
5. Output S

算法 2. 以固定状态为起点的定长系统调用集合生成算法 $SequenceOfState(Graph, state, L)$.

输入：系统调用图 $Graph$，状态 $state$，系统调用长度 L

输出：以状态 $state$ 为起点的定长 L 的系统调用片断的集合 S

1. $S := \varnothing$
2. if $L = 0$ then
3. $\quad$ Output S
4. endif
5. for each $e \in \{e \mid e$ is one of the outgoing edge of $state\}$ do
6. $\quad$ if $e.operation = \varepsilon$ then
7. $\quad\quad S_{tmp} := SequencesOfState(Graph, e.target, L)$;
8. $\quad\quad S := S \cup S_{tmp}$;
9. $\quad$ else
10. $\quad\quad S_{tmp} := SequencesOfState(Graph, e.target, L-1)$;
11. $\quad\quad$ If $S_{tmp} = \varnothing$ then
12. $\quad\quad\quad S = S \cup \{e.operation\}$
13. $\quad\quad$ else
14. $\quad\quad\quad$ for each s in S_{tmp} do
15. $\quad\quad\quad\quad s := e.operation + s$
16. $\quad\quad\quad\quad S := S \cup \{s\}$
17. $\quad\quad\quad$ endfor
18. $\quad\quad$ endif
19. $\quad$ endif
20. endfor
21. Output S

2.3 实时检测

实时检测时，监控应用程序进程 A 执行的系统

调用，并将进程执行的系统调用以步长为1，长度为L的滑动窗口截成定长系统调用序列S_i. 如果S_i不在集合$SequencesOfGraph(Graph, L)$中，则认为受到入侵. 假设应用程序实际所有可能执行的系统调用序列为$RealSeqneuces_A$，由于在分析应用程序可能执行的序列过程中未对变量的值等情况进行分析，实际获得的集合$SequencesOfApplication(Graph, L)$中可能包含一些实际运行中不可能出现的序列. 但应用程序实际可能执行的序列均包含在集合$SequencesOfApplication(Graph, L)$中，即

$$SequencesOfGraph(Graph, L) \supset RealSeqneuces_A.$$

因此，该模型在分析过程完整的情况下，检测误报率为0. 检测的漏报率我们会在实验结果部分进一步详细分析.

3 实验结果

本实验在Windows XP环境下进行，这主要是基于目前针对Windows平台的攻击方法和恶意代码比较多，容易构造实验过程. 但Windows操作系统的系统调用层次不如Linux清晰，系统调用数量很庞大，部分系统调用仍不公开，因此，本实验选择了一个典型的Windows程序Ping.exe作为实验对象. 主要实现了对Ping.exe中所执行的系统调用的截获.

3.1 经代码模糊变换的病毒检测

由于病毒是恶意代码的典型代表，从本模型的检测原理来看，检测病毒执行过程和缓冲区溢出、进程代码远程注入的方法是一致的，而病毒攻击过程更容易构造. 因此，可通过对病毒的检测来分析本模型的检测能力.

实验先对Ping.exe进行分析，建立应用程序Ping的正常行为定义，系统调用序列长度L为15. 然后让Ping.exe程序感染事先搜集的几种病毒：Funlove、求职信(W32.Elkern.C)等.

代码变换采取两种思路：对于有源代码的病毒样本，采用文献[7]中的方法；对于没有源代码的病毒样本，采用病毒加壳代码转换的方法[8]. 实验结果如表1所示.

现有的防病毒软件均不能很好地检测病毒变种，文献[7]中的方法由于无法解析经过变形转换的代码，也不能有效地实施检测经过加壳代码转换的病毒. 本模型对这些恶意代码均能有效检测.

表1 几种病毒检测情况分析

病毒名称	瑞星	诺顿	文献[7]中的方法	本模型
Win32. Loicer(Obscuated)	×	×	×	√
Win32. Foroux(Obscuated)	×	×	×	√
Win32. Zaka(Obscuated)	×	×	×	√
Win32. Cornad(Obscuated)	×	×	×	√
Win32. Driller(Obscuated)	×	×	×	√
Funlove(Obscuated)	×	×	√	√
求职信(W32.Elkern.C)(Obscuated)	×	×	√	√

注：√表示检测成功，×表示未检测到.

3.2 抗"模仿攻击"能力

针对基于系统调用序列的异常检测方法，Wagner等人提出了"模仿攻击"方法[9,10]. 该方法实际上是检测模型的漏报问题. "模仿攻击"假设攻击者在完全清楚异常检测模型的方法、参数设置的情况下，通过构造检测系统认为正常的系统调用序列隐藏攻击过程，绕过入侵检测系统. 现已有一些防范"模仿攻击"的方法，但均未能很好解决，具有明显的局限性[11,12]. 本实验按照文献[9]中的方法未能构造出"模仿攻击"，在这里主要分析在同等条件下，对抵抗模仿攻击能力的差异.

对于一个攻击序列$Attack_s = S_0 S_1 \cdots S_{n-1}$，集合$M$是入侵检测系统可接受的系统调用序列. "模仿攻击"是以M中的系统调用序列构造一个新的序列

$$Attack_N = N_0 N_1 \cdots N_{m-1} (m \ge n),$$

使得存在一个集合

$$\{j_0, j_1, \cdots, j_{n-1}\},\ 0 \le j_0 \le j_1 \le \cdots \le j_{n-1} < m,$$
$$N_{j_0} = S_{j_0},\ N_{j_1} = S_{j_1}, \cdots, N_{j_{n-1}} = S_{j_{n-1}}.$$

并且，对于任意$0 \le j < m - L + 1$，$N_j N_{j+1} N_{j+2} \cdots N_{j+L-1} \in M$.

在攻击序列一定、应用程序一定的情况下，M中包含的不可能系统调用序列①越多，对构造攻击序列帮助越大. 在本模型中，由于没有考虑实际应用环境以及程序的语义，可能引入一些不可能序列，比如循环的次数、递归问题等等. 但与其它模型相比，本模型引入的序列要少很多. 一般异常检测模型由于不能有效避免误报，采用近似匹配方法，通过分析系统调用之间的距离进行检测，即在一定程度上允许实时的系统调用序列和正常行为定义存在差别. 如果监控的系统调用类型总数为C，对于长度N的系统调用序列S，假设检测系统允许1位的差异，这样由该序列而引入的不可能系统调用序列数量为$N \times C - S'$，其中S'是应用程序可能的系统调用序列中与S只有1位不匹配的序列. 对于所有的系统

① 不可能系统调用序列是指应用程序本身不可能触发的系统调用序列.

调用序列，因为1位的差异而引入的其它序列数量是很庞大的. 因此，本模型虽然不能保证不受到“模仿攻击”，但和N-gram等模型相比，其抗“模仿攻击”能力更强.

3.3　递归函数问题

在合并递归函数的函数调用图过程中，不仅会因为不控制递归层次而引入不可能系统调用序列，而且合并方法也会引入不可能系统调用序列. 但实验发现因为后者引入的不可能系统调用序列并不能让攻击过程变得更容易，即它不会增加新的漏报.

以图4的函数为例. 假设在某一函数sub_C中调用函数sub_A()，sub_C执行的系统调用序列如下：

$$C \rightarrow efAgh,$$

其中A表示函数A执行的系统调用序列，e，f，g，h分别是系统调用.

以图4示例则有可能出现：

$$C_1 \rightarrow efacacdbgh.$$

但这种序列在实际运行过程中是不会发生的. 以上面的调用过程为例，应该是

$$C_2 \rightarrow efacacdbdbgh.$$

C_1是由于递归函数系统调用图合并造成的不可能序列，C_2是应用程序的正常系统调用序列. 但C_1增加到行为定义中并不会增加误报. 假设攻击$Attack_A$可以通过C_1构造一个攻击序列，$Attack_A = S_0 S_1 \cdots C_1 \cdots S_{n-1}$，利用“模仿攻击”思想，可以插入两个无操作系统调用d，b，将C_1转化为C_2，即得到$Attack_A$仍可以通过$Attack'_A = S_0 S_1 \cdots C_2 \cdots S_{n-1}$完成攻击. 即包含这些不可能系统调用序列的集合和不包含这些序列的集合，在两种情况下构造攻击过程的难度是一样的，并不会因为引入这些不可能路径而增加漏报.

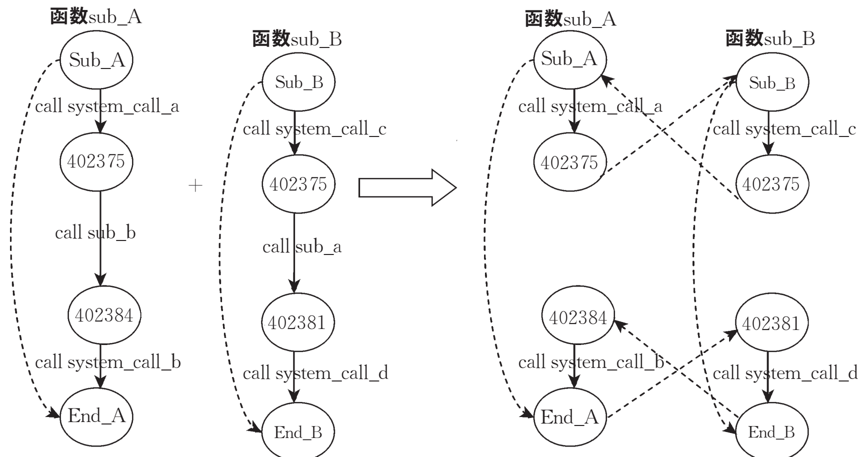


图4　递归函数合并

4　总　结

本文提出了一种基于可执行文件静态分析的入侵检测模型，该模型通过对应用程序可执行文件的静态分析，建立应用程序所有可能执行的定长系统调用集合，通过实时监控进程执行的系统调用序列是否在该集合中实施检测. 该模型具有以下特点：

（1）该模型不需要源文件、大规模训练数据，通用性和易用性好；

（2）在应用程序可执行文件完整的情况下，可达到误报率为0；

（3）抵抗模仿攻击的能力更强，漏报率更低.

参　考　文　献

1　David Geer. Behavior-based security become the main-stream of network security. Computer, 2006, 39(3): 14～17

2　Forrest S., Hofmeyr S. A., Somayaji A., Longstaff T. A.. A sense of self for unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 1996, 120～128

3　Hofmeyr S. A., Forrest S., Somayaji A.. Intrusion detection using sequences of system calls. Journal of Computer Security, 1998, 6(3): 151～180

4　Wepsi A., Dacier M., Debar H.. Intrusion detection using variable-length audit trail patterns. In: Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection, Springer-Verlag, London, UK, 2000, 110～129

5　Sekar R., Bendre M., Dhurjati D., Bollineni P.. A fast automation-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy. IEEE Computer Society, Washington, DC, USA, 2001, 144～149

6　Feng H., Kolesnikov O., Fogla P., Lee W., Gong W.. Anomaly detection using call stack information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2003, 62

7　Mihai Christodorescu, Somesh Jha. Static analysis of executa-

bles to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2003, 169～186

8 Wroblewski G.. A general method of program code obfuscation [Ph. D. dissertation]. Wroclaw University, Poland, 2002

9 Wagner David, Soto Paolo. Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, USA, 2002, 255～264

10 Christopher Kruegel, Engin Kirda. Automating mimicry attacks using static binary analysis. In: Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, 2005, 161～176

11 Sufatrio, Roland H. C. Yap. Improving host-based IDS with argument abstraction to prevent mimicry attacks. In: Proceedings of RAID 2005, Springer, Germany, 2006, 146～164

12 Chung S. P., Mok A. K.. On random-inspection-based intrusion detection. In: Proceedings of the RAID 2005, Springer, Germany, 2006, 165～184

13 Wang Yi-Min, Roussev R.. GateKeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In: Proceedings of the 18th USENIX Conference on System Administration, USENIX Association, Berkeley, CA, USA, 2004, 33～46

14 Sung A. H., Xu J., Chavez P., Mukkamala S.. Static analyzer of vicious executables(SAVE). In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04), IEEE Computer Society, Washington, DC, USA, 2004, 326～34

附录.

算法. 生成函数流程图 *FunctionGraphGeneration*(*F*) **算法.**

输入:函数可执行代码 *F*

输出:函数流程图 *Graph*

```
1. InitialGraph(Graph); //生成初始状态和结束状态
2. FindAllReferencedPoints(); //确定所有引用点
3. GenerateStatesOfAllReferencedPoints();
                        //为每一个引用点建立一个状态 state
4. state := Graph.start
5. addr := F.start
6. While addr != F.end do
7.    command=ReadOneCommond(addr)
                                        //读取一条指令
8.    if commond.type==CALL then  //指令是 CALL
9.    newstat=GetState(Graph,comond.next)
                        //取 CALL 指令的下一条指令状态
10.   transation=AddTrans(Graph,state,newstate,
                commond.func)//图中增加一个状态转换
11.     state=commond.next
12.   else if(commond.type==JMP) then  //JMP 指令
13.      newstate=GetState(Graph,commond.target)
                              //取跳转指令目标地址状态
14.     transation=AddTrans(graph,state,
                newstate,NULL)
15.     addr=addr.next;
16.     while addr != F.end do
17.       if IsReferencedPoint(addr) then
18.          state=GetState(graph,newaddr); break;
19.       endif
20.     endwhile
21.     if addr=F.end then break;
22.   else if IsJXXseries(commond.type) and
          command.type != JMP then //JXX 系列指令
23.     newstate=GetState(graph.target);
24.     transation=AddTrans(graph.state,newstate,ε);
25.   else if command.type=RET then //RET 指令
26.     transation=AddTrans(graph,state,end, ε);
27.   else if IsReferencedPoint(addr) then
                              //引用点,更新当前状态
28.     newstate=GetState(graph,addr);
29.     transation=
          AddTrans(graph,state,newstate,NULL);
30.     state=newstate;
31.   endif
32.   addr := Command.nextaddr;
33. end while
34. Output Graph
```

SU Pu-Rui, born in 1976, Ph. D.. His research interests include intrusion detection and network security.

YANG Yi, born in 1982, M. S. candidate. His research interests focus on network security.

Background

The research is supported by the National Natural Science Foundation of China under Grant Nos. 60025205, 60273027, and National Key Basic Research Program of China under Grant No. G1999035802.

This paper is part of work of COMUS(COmputer iMmune System) system, a host-based intrusion detection system developed by us. The system detects intrusions by monitoring the system calls triggered by key processes and introduces the immune mechanism, partly inspired by nature immune system. The paper introduces the key algorithm of the anomaly detection, which detects the abnormalities based on the profile from the static analysis of the executables.