ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.1, January 2009, pp.138−151
doi: 10.3724/SP.J.1001.2009.03165

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

基于混杂模型的上下文相关主机入侵检测系统*

李 闻+, 戴英侠, 连一峰, 冯萍慧

(中国科学院 研究生院 信息安全国家重点实验室,北京 100049)

Context Sensitive Host-Based IDS Using Hybrid Automaton

LI Wen+, DAI Ying-Xia, LIAN Yi-Feng, FENG Ping-Hui

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

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

Li W, Dai YX, Lian YF, Feng PH. Context sensitive host-based IDS using hybrid automaton. *Journal of Software*, 2009,20(1):138−152. http://www.jos.org.cn/1000-9825/3165.htm

Abstract: A key function of a host-based intrusion detection system is to monitor program execution. Models constructed based on static analysis have the advantage of not producing false alarms; still, they can not be put into practice due to imprecision or inefficiency of the model. The prior work has shown a trade-off between efficiency and precision. In particular, models based upon non-deterministic finite state automaton (DFA) are efficient but lack precision. More accurate models based upon pushdown automaton (PDA) are very inefficient to operate due to non-determinism in stack activity. DYCK model, VPStatic model and IMA use some subtle approaches to achieve more determinism by extracting information about stack activity of the program or inserting code to expose program state or just inline the local automaton but still can not solve the problem of indirect call/JMP. This paper presents a new training-free model (hybrid finite automaton, HFA) to gain more determinism and resolves indirect call/JMP through static-dynamic hybrid approach. The results show that in run-time, these models slowed the execution of the test programs by 5% to 10%. This paper also formally compares HFA with some typical models and proves that HFA is more accurate than others and it is more suitable for dynamic linked applications. Some technical details of the protocol type system on Linux and experimental results are also presented in the paper.

Key words: intrusion detection; hybrid automaton; training-free; call context sensitive; Linux

摘 要: 主机入侵检测的关键是监测进程的运行是否正常.现有的基于静态分析建模的方法具有零虚警的优良特性,但是,由于缺乏精确性或者效率的问题仍然不能实际使用,先前的工作试图在这两者之间寻找平衡点.基于NFA(nondeterministic finite automaton)的方法高效但是不够精确,基于 PDA(push down automaton)的方法比较精确但却由于无限的资源消耗而不能应用.其他模型,例如 Dyck 模型、VPStatic 模型和 IMA 模型使用一些巧妙的方法提高了精确性又不过分降低可用性,但是都回避了静态分析中遇到的间接函数调用/跳转问题.提出一种静态分析-动态绑定的混杂模型(hybrid finite automaton,简称 HFA)可以获得更好的精确性并且解决了这一问题.形式化地与典型的上下文相关模型作比较并且证明 HFA 更为精确,而且 HFA 更适合应用于动态链接的程序.还给出了基于 Linux 的原型系统的一些实现细节和实验结果.

* Supported by the National Natural Science Foundation of China under Grant Nos.60403006, 60503046 (国家自然科学基金)
Received 2007-02-27; Accepted 2007-08-03

关键词: 入侵检测;混杂自动机;免训练;调用上下文相关;Linux
中图法分类号: TP393 文献标识码: A

入侵检测系统一般可以分为两大类:基于特征的入侵检测系统(misuse-based)和基于异常的入侵检测系统(anomaly-based).前者定义一系列针对已知入侵事件的特征(signature),如果当前事件和特征匹配,则认为是发生了入侵.后者定义一个正常系统的行为模型(profile),如果当前系统的行为和这个模型有背离,则认为有入侵发生.前者具有较低的虚警概率,但是不能检测未知的攻击事件;而后者没有这个弱点.由于对进程的行为建模相对容易,所以后者经常应用于主机入侵检测系统中.

对于基于异常的入侵检测系统,模型的好坏主要有两种度量:精确性和资源消耗.精确性是指模型和实际系统的贴合程度,模型越精确,攻击者躲避检测的可能性越小,从而漏警/虚警概率越小.资源消耗越小的模型其效率越高,从而更适用于实际系统.

主流的异常入侵检测系统建模方式可以分为两类:通过训练建模和通过静态分析建模.

Denning[1]等人构建了一种基于用户登录次数和资源访问行为构建模型的主机入侵检测系统,由于用户的行为经常突然改变而且难以模型化,所以研究热点由用对户行为建模转移到了对进程的行为的建模.Forrest等人[2,3]提出枚举所有可能的长度为k的系统调用序列(下文简称“序列”)来构建模型的方法,这一方法被后人继承和发展,Helman[4],Lee[5],Warrender[6]等人提出了一系列的改进,提取序列的频率分布特性或者引入数据挖掘及隐式马尔可夫链等理论和工具.Sekar[7]等人通过把PC指针和系统调用相关联,利用有穷自动机(finite state automaton,简称FSA)学习序列,把进程的上下文信息引入到模型中.Feng[8]等人提出的VtPath模型抽取函数调用栈信息,可以显著地减少收敛时间,提高模型精确性,减少虚警概率.

基于训练的方法建模方法本质上都需要解决经典的机器学习问题,训练不充分会使模型的虚警概率较高,而且存在构造训练集有时过于困难(例如,对 SendMail)、训练结果无可移植性、模型收敛时间过长等缺陷,所以很难应用于实际环境中.近年来出现了通过对源代码或者二进制代码静态分析获得序列特性的建模方法,这类方法把进程看作一个自动机,检查序列是否是自动机的生成语言,由于具有零虚警概率的良好特性,这类方法逐渐成为研究的热点.

Wagner[9]等人提出了通过静态分析源代码建立模型的方法,讨论了基于不确定有穷自动机(nondeterministic finite automaton,简称NFA)的callgraph,digraph和基于下推自动机(push down automaton,简称PDA)的abstract stack三种模型.NFA模型虽然高效但却存在不可能路径(impossible paths),攻击者有可能利用这些路径躲避检测.上下文相关的PDA模型包含了对进程的函数调用栈的抽象但是运行代价过高,Wagner不得不提出一种digraph模型作为折衷的方案.为了克服对源代码的依赖性,Giffin[10]等人提出了对二进制文件作静态分析,建立NFA和PDA模型的方法,还给出了通过插入null-call减少PDA模型不确定性的方案.在其后期的研究中[11]提出了上下文相关Dyck模型,通过插入pre-call和post-call进一步提高了效率.Feng[12]随后提出的VPStatic模型是VtPath模型的免训练版本,他分析了前人的研究,指出模型的精确性本质上是和自动机的确定性相关的,还指出Dyck模型和VPStatic模型本质上是栈确定下推自动机(sDPDA),形式化地证明了这些模型比单纯的PDA具有更高的精确性.Gopalakrishna[13]等人提出的IMA(inline model of automaton)模型在由函数局部自动机构造全局自动机时使用了直接嵌入的思想,给出了提高PDA精确性的新思路.

单纯静态分析不可能获得间接函数调用和跳转的目的地址,这种不确定性只有在进程运行时才能消除.先前的研究工作或者回避这个问题(静态编译且不考虑库函数),或者通过在自动机中添加ε边来解决.ε边会为模型引入更多的不确定性从而降低精确性,而且ε边会使自动机变得复杂而低效.

本文的思想是把静态分析中无法解决的问题推迟,在进程运行中利用对函数局部自动机的动态链接和运行时修正消除这种不确定性.本文提出的 HFA(hybrid finite automaton)模型是一种动态-静态混杂的模型,比先前的同类自动机模型更接近确定下推自动机(deterministic push down automaton,简称 DPDA),从而可以获得更高的精确性和效率.而且 HFA 由于先天的模块化特性,更适用于动态链接的应用环境中.

本文第 1 节描述 HFA 的建立过程.第 2 节描述使用 HFA 的入侵检测过程并对入侵检测能力加以分析.第 3 节形式化地分析和比较同类的上下文相关自动机模型.第 4 节描述实现基于 Linux 的原型系统的一些技术细节.第 5 节给出实验结果.最后是对现有工作的总结和对未来研究工作的展望.

1 HFA 模型的建立

HFA 模型的建立分为两步,首先,我们通过对二进制代码的静态分析建立每一个函数的不确定有穷自动机(局部 NFA),并且转换为有穷状态自动机(局部 DFA).然后,在进程的运行中,运用动态绑定和运行时修正建立完整的 HFA 模型,流程如图 1 所示.

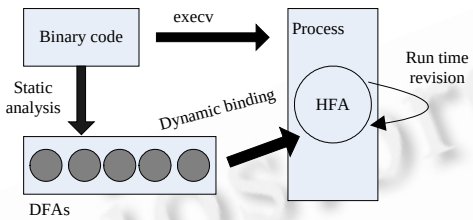

Fig.1 Construction of HFA model
图1 HFA模型的建立过程

1.1 从二进制代码生成单函数控制流程图

通过对源代码或者二进制代码静态分析可以获得每个函数的控制流程图(control flow graph,简称 CFG),静态分析源代码本质上是一种预编译的过程,可以避免对间接函数调用/转移的分析,但是程序的源代码不总是可用,静态分析二进制代码的方法与编程语言无关,只与硬件平台相关,所以我们采取静态分析二进制代码的方法.

从二进制代码生成控制流程图的技术已经很成熟,文献[10–13]对这种方法有比较详细的描述,我们使用修改后的EEL(executable editing library)[10,11]库来完成这一工作.EEL是一种用C++实现的SPARC平台修改和操作二进制可执行文件的工具库,我们保留它的框架,并且把它移植到了IA32 平台上.EEL把一个函数划分为很多个基础块(base block),每一个基础块是一组线性执行的指令序列,这些基础块通过控制流跳转关系联接起来生成CFG,间接跳转指令(如jne [eax])无法在静态分析时获知其目的地址,我们暂时不添加相应的边,并且对这一基础块做标记.间接跳转主要存在于库函数,特别是动态链接库中,如果编译时打开了优化选项,(gcc –O2)也大量存在于程序主体中.图 2 是示例C代码以及函数num2char对应的单函数CFG,这一函数包含间接调转和间接函数调用指令.

通过静态分析可以获知所有函数调用指令的源地址(call 指令所在地址)和所有的直接函数调用的目的地址,我们把它们分别保存于两张二叉查找表 CSAT(call site source address table)和 CDAT(call site destination address table)中.类似地,对调转指令也有两张二叉查找表 JSAT(JMP site source address table)和 JDAT(JMP site destination address table).

图 2 中两个基础块之间的虚线表示通过静态分析不能获知的间接调转,在进程执行时通过运行时修正操作将添加这些边,为了清楚起见,也把它们包含在图 2 中.

1.2 从单函数控制流程图生成局部确定有穷自动机

首先从单函数 CFG 生成局部 NFA,然后通过ε边缩减进一步生成局部 DFA.

从 CFG 生成局部 NFA 的过程很直观,每一个基础块对应一个状态,如果基础块之间的转移是由于函数调用引起,则在两个基础块之间添加一条标号为“目的函数名@源地址”的有向边,否则添加ε边,如图 2 所示.与文献[10,11]不同,我们把用户函数调用也看作自动机的符号.符号中包含源地址是为了消除同名但却不同位置的函数调用造成的不可能路径以及符号表的二义性.

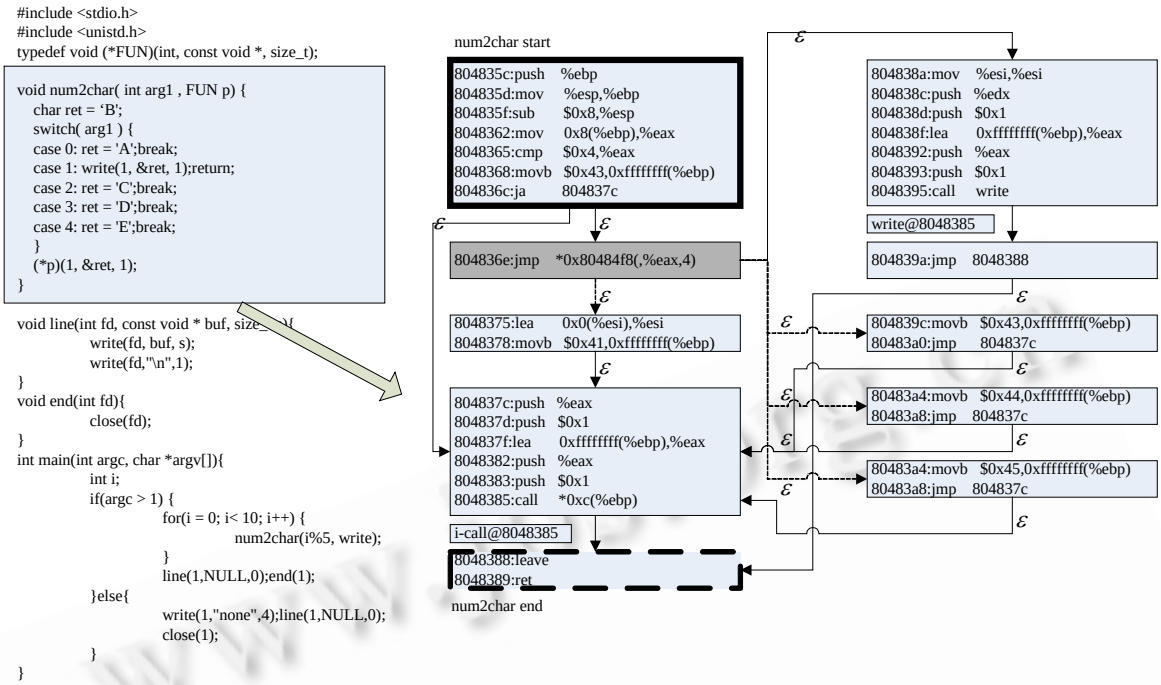

Fig.2 From binary code to CFG of individual functions
图 2 从二进制代码生成单函数 CFG

函数调用有两种方式:直接函数调用(如 call 0x80403734)和间接函数调用(如 call [eax]),直接函数调用在静态分析时可以获得目的函数地址,而间接函数调用则不能.对于间接调用,用一个函数范围内唯一的符号标记有向边,如图 2 中的“i-call@8048385”.文献[9−13]最终把所有的局部 NFA 按照函数调用关系通过ε边连接起来,生成跨函数的全局 NFA 或者 PDA,对直接函数调用可以这样做,但是对于间接函数调用,由于无法获知被调用函数,所以不可行,HFA 本质上把链接推迟到进程运行时进行,从而避免了这一问题,我们将在第 1.3 节加以讨论.

局部NFA通过ε边缩减可以减少状态数,文献[10,11]提出了线性时间缩减算法,但是不能消除所有的ε边,我们使用时间复杂度为$O(n\log n)$[14]的算法,n为NFA的状态数,经过缩减后得到了不含ε边的局部DFA.先前的文献并没有生成局部DFA,这是由于在从局部NFA到全局NFA或者全局PDA的过程中还需要重新引入ε边,动态绑定方案不需要引入ε边,所以这一转换是有意义的.

图 3 给出了图 2 示例 C 代码中各函数对应的局部 DFA.

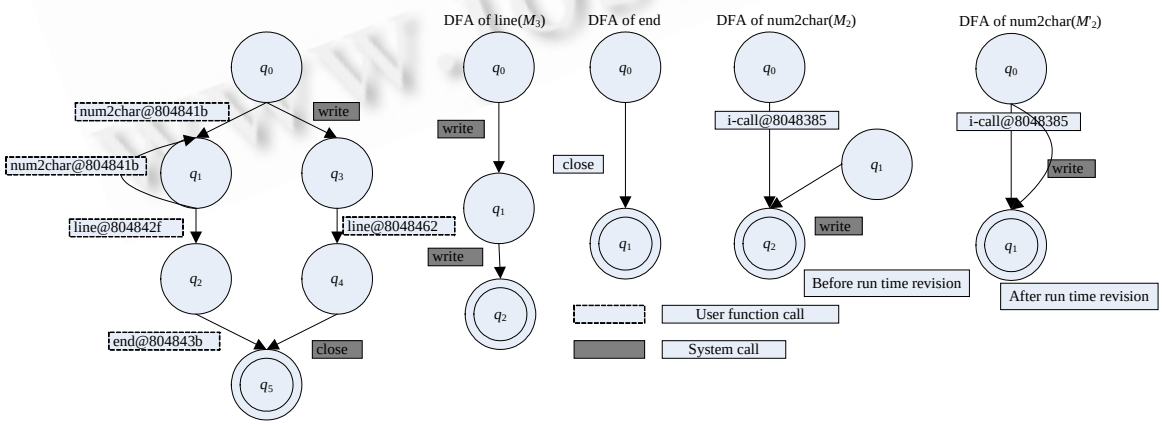

Fig.3 DFAs of individual function
图3 各个函数对应的局部DFA

某些 C 库函数,如 write,对应于单一的同名系统调用,但是某些 C 库函数,如 malloc,可能产生多个系统调用,在实际的原型系统中,我们为每一个库函数预先建立 DFA.为了方便讨论,这里,我们忽略标准 C 库函数 write 和 close 的对应 DFA,并且认为这些库函数等同于对应的系统调用.

1.3 局部确定有穷自动机的动态绑定和运行时修正

通过第 1.1 节和第 1.2 节的处理,我们得到了一系列局部自动机,为了建立一个完整的进程行为模型,需要把这些自动机连接起来,之前的文献都采用静态的方法构造一个全局的自动机.图 4(a)、图 4(b)给出了一些典型模型的处理过程,上下文无关的NFA模型[10]的做法是:假设局部NFA M_1在非ε边g调用局部NFA M_2,则从g的出射状态到M_2的初始状态引一条ε边并且从M_2的所有终止状态到g的入射状态引ε边,然后删除M_1中g的出射状态到入射状态的边.这样会为模型引入不可能路径(impossible paths)从而降低精确性,不可能路径是指进程执行中不可能出现的路径,即函数返回到位置不同的另一个同名调用点.PDA模型[10,11]在每一个新添加的ε边上保存目的状态作为栈符号,把函数调用的上下文关系引入模型中消除了不可能路径,但是,PDA模型操作代价过高,特别是有可能出现资源无限消耗的问题.

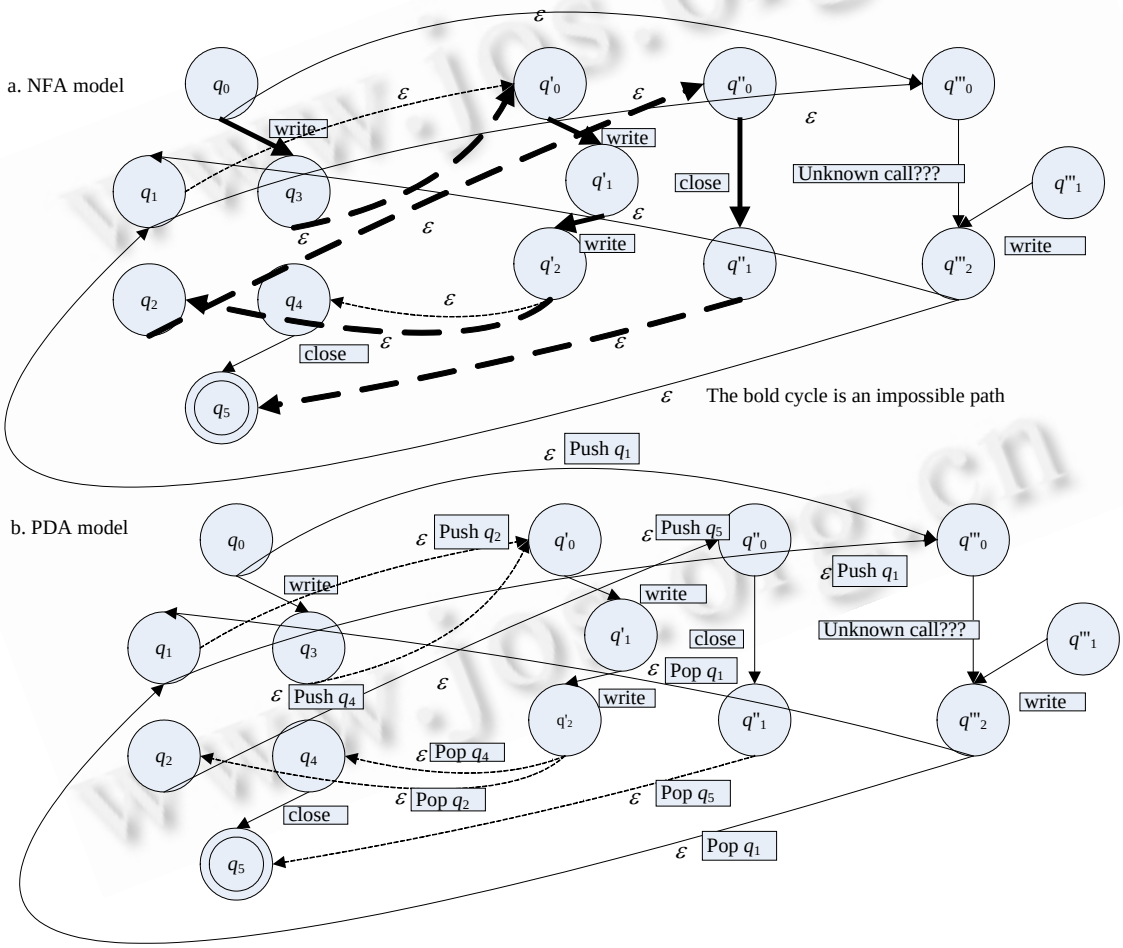

Fig.4 Static binding process of NFA model and PDA model

图4 NFA模型和PDA模型的静态绑定过程

Dyck模型[11]通过在每次函数调用前、后插入虚拟的pre-call和post-call并且使用null-call squelching技术消除了大量的ε边,从而使每次自动机操作的栈信息确定,比PDA模型具有更好的精确性.IMA模型[13]在每个调用点把局部NFA M_2直接嵌入(inline)到NFA M_1中,如果M_1反复调用M_2,将会导致大量的重复嵌入,极端情况下,如

果M_1通过M_2间接递归调用自身会导致无限资源消耗.VPStatic[12]本质上也是静态绑定模型,Feng证明了其精确性与Dyck模型相当,由于这些模型构造流程比较复杂,在此不再赘述.

上述静态绑定局部自动机的方案除了可能需要引入额外的ε边增加不确定性外,还存在固有的缺陷:如果某个函数包含间接跳转指令,则对应的 NFA 可能包含不可达状态(例如 q_1''')不能反映真实的函数行为.如果在局部 NFA 中存在间接调用,则仅仅通过静态分析不能确定函数的调用关系而无法把局部自动机联系起来(例如从 q_0''' 到 q_2''' ,尽管根据源代码可知是对函数 line 的调用).为了克服这些缺陷,我们通过动态-静态混杂的方法建立模型,即通过静态分析得到模型的各个局部DFA,整个HFA模型直到进程运行时才建立起来.我们的做法可以归纳为:1) 操作和动态绑定局部 DFA.2) 根据间接跳转的目的地址运行时修正局部 DFA.3) 验证局部 DFA 之间绑定操作和局部 DFA 内部修正操作的合法性.

1.3.1 局部 DFA 的操作和动态绑定

一个DFA可以表示为 5 元组 $\{Q,\Sigma,\delta,q_0,F\}$,其中,Q为状态集,Σ为字母表,δ为状态转移函数,q_0为开始状态,$F\subset Q$ 为接受状态集.如图 5 所示.

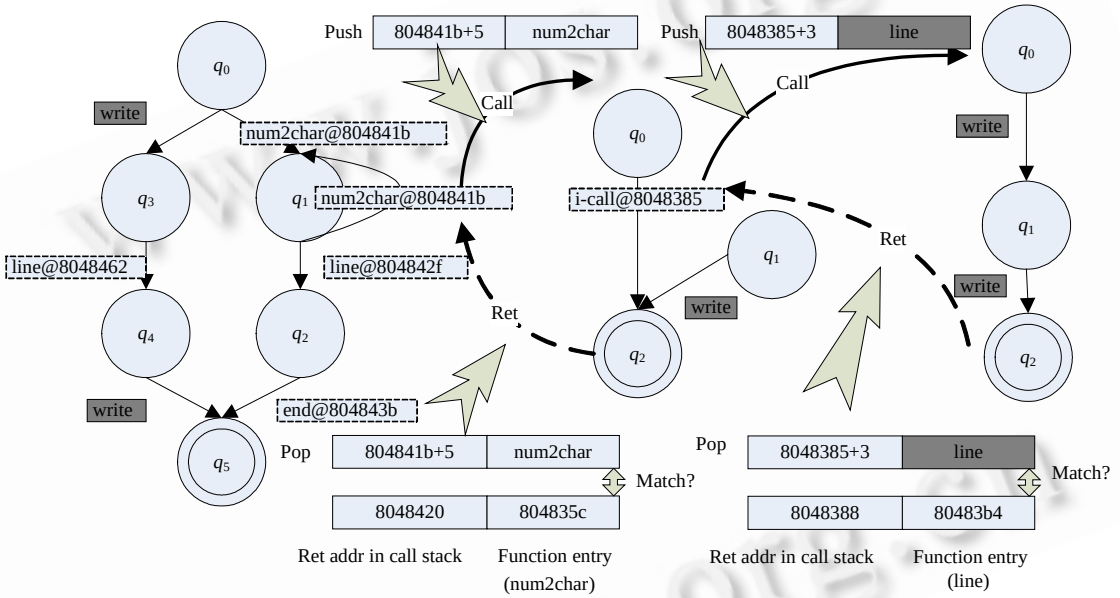

Fig.5 Dynamic binding process of HFA model

图5 HFA模型的动态绑定过程

在图 5 中,函数*main*对应的DFA M_1可以表示为

$$Q^{M_1}=\left\{q_0^{M_1},q_1^{M_1},...,q_5^{M_1}\right\},\ F^{M_1}=\left\{q_5^{M_1}\right\},$$

$$\Sigma^{M_1}=\left\{num2char@804841B,line@8048462,line@804842F,end@804843B,write,close\right\}$$

$$\delta^{M_1}(q_0^{M_1},num2char@804841B)=q_1^{M_1},\ \delta^{M_1}(q_0^{M_1},write)=q_3^{M_1},$$

$$\delta^{M_1}(q_1^{M_1},num2char@804841B)=q_1^{M_1},\ \delta^{M_1}(q_1^{M_1},line@804842F)=q_2^{M_1},$$

$$\delta^{M_1}(q_2^{M_1},end@804843B)=q_5^{M_1},\ \delta^{M_1}(q_3^{M_1},line@8048462)=q_4^{M_1},$$

$$\delta^{M_1}(q_4^{M_1},close)=q_5^{M_1}.$$

函数*num2char*对应的DFA M_2可以表示为

$$Q^{M_2}=\left\{q_0^{M_2},q_1^{M_2},q_2^{M_2}\right\},\ F^{M_2}=\left\{q_2^{M_2}\right\},$$

$$\Sigma^{M_2}=\left\{i\text{-}call@8048385,write\right\},$$

$\delta^{M_2}(q_0^{M_2}, i\text{-}call@8048385) = q_2^{M_2}$, $\delta^{M_2}(q_1^{M_2}, write) = q_2^{M_2}$.

函数*line*对应的DFA M_3可以表示为

$Q^{M_3} = \{q_0^{M_3}, q_1^{M_3}, q_2^{M_3}\}$, $F^{M_3} = \{q_2^{M_3}\}$,

$\Sigma^{M_3} = \{write\}$, $\delta^{M_3}(q_0^{M_3}, write) = q_1^{M_3}$, $\delta^{M_3}(q_1^{M_3}, write) = q_2^{M_3}$.

我们为每一个 DFA 在内核中维护一个运行时状态栈.假设在程序运行中,某一个时刻 *main* 即将调用*num2char*,即M_1处于状态 $q_1^{M_1}$ 即将接受符号*num2char*@804841*b*.在*num2char*被调用时,我们可以确定*call*指令的源地址A_{src}=804841*b*+*call*指令长度、目的地址A_{dst}=*num2char*(由静态分析确定),如果A_{src}和A_{dst}都合法就把 2 元组 (804841*b*+*call* 指令长度,*num2char*) 压入 M_1 的状态栈中转而操作 M_2. 在 M_2 操作过程中我们发现,"*i-call*@8048385"的源地址A_{src}=8048385+*call*指令长度,目的地址A_{dst}=*line*(直到运行时才能确定),我们把 2 元组(8048385+*call*指令长度,*line*)压入M_2的状态栈转而操作M_3,依此类推.

M_2操作有两种结果:或者M_2遇到了不被接受的某一个符号,或者达到了接受状态.前一种情况可能由于攻击者插入了额外的系统调用序列.后一种情况表示*num2char*正常返回,此时,我们从M_1的状态栈中弹出 2 元组(A_{src},A_{dst})并检查该 2 元组是否等于(*num2char*的用户栈返回地址,*num2char*),如果两者不相等,表示攻击者试图操纵函数返回到另一个运行点发动不可能路径攻击,否则,我们继续操作M_1,依此类推.

由于局部DFA直到进程运行时才根据A_{dst}联系起来,特别地,对于间接函数调用A_{dst}直到运行时才能被确定,所以我们称这种把局部DFA链接起来的方法为动态绑定.

与先前的基于静态绑定的自动机模型相比,动态绑定有如下特点:

1) 在任一时刻,我们操作一个局部 DFA 而不是把整个进程看作一个大的自动机模型来对待.每一个局部 DFA 可以有自己的状态集、字母表等要素,这些集合的秩比把进程看作一个大的自动机模型要小很多,而且每一步操作只和局部 DFA 的规模相关,所以 HFA 的每一步操作对进程性能的影响几乎与进程代码规模(函数数量)无关.
2) 局部 DFA 之间相互独立,所以我们可以修正每一个单独的局部 DFA 而对整个模型不会产生影响.对此,我们将在第 1.3.2 节进一步加以讨论.
3) 局部 DFA 之间的转移操作、局部 DFA 内部的操作都是确定的,所以每一步操作所需的资源和计算量较小.对于 NFA 模型或者纯粹的 PDA 模型,每一步操作都有可能派生出多个分支,基于 sDPDA 模型,如 Dyck 和 VPStatic 局部 NFA 之间的栈操作是确定的,但是,局部自动机内部的操作则不是.对此,我们将在第 3 节进一步加以讨论.

1.3.2 运行时修正局部 DFA

回顾第 1.2 节,在静态分析二进制代码生成 CFG 时,某些基础块中可能包含间接跳转指令,我们不能确定它的目的地址,为了便于后继处理,我们做了标记.除了某些非标准的控制跳转,如调用 longjmp,siglongjmp 函数以外,间接跳转指令的目的地址都在指令所在函数的内部,另外,一个非函数内部的跳转一般由于攻击者改写了目的地址导致对非标准的控制跳转需要作特殊的处理,见后面第 4.2 节的讨论.

在进程的运行过程中,根据已经被确定的接跳转指令的目的地址添加适当的ε边,对得到的 NFA 运用ε边缩减算法,并且用新得到的局部 DFA 替换原有的局部 DFA.因为这两个 DFA 有相同的字母表(字母表仅由函数包含的 call 指令和系统调用点决定),所以原有 DFA 的运行时状态栈保持不变,我们也可以根据当前的 PC 指针确定在新 DFA 中所处的状态,也因此修正局部 DFA 而对整个模型没有影响.

注意,动态绑定操作由输入符号触发,而运行时修正操作由基础块中的标记触发,一旦触发就清除这个标记,表示运行时修正已经完成.

如图 3 所示,函数*num2char*对应的DFA经过修正后消除了不可达状态 $q_1^{M_2}$,这个修正在函数*num2char*执行过程中(操作M_2过程中)进行,此时,M_2处于状态 $q_0^{M_2}$.修正后,我们用新的DFA M'_2取代原有的DFA M_2,根据PC指针确定当前M'_2中的状态 $q_0^{M'_2}$,然后继续操作M'_2.修正后的DFA M'_2可以表示为

$Q^{M_2'}=\{q_0^{M_2'},q_1^{M_2'}\}$, $F^{M_2'}=\{q_1^{M_2'}\}$,

$\Sigma^{M_2'}=\{i\text{-}call@8048385,write\}$,

$\delta^{M_2'}(q_0^{M_2'},i\text{-}call@8048385)=q_1^{M_2'}$, $\delta^{M_2'}(q_0^{M_2'},write)=q_1^{M_2'}$.

只有包含间接跳转指令的局部 DFA 需要运行时修正,这种修正主要发生在动态链接库中,我们可以采取一些缓存技术保存修正后的结果,所以对进程的运行时性能影响不大.

1.3.3 动态绑定和运行时修正的合法性校验

在操作 HFA 模型的过程中需要确保局部 DFA 之间的动态绑定操作和运行时修正操作不会被攻击者控制,所以需要对它们的合法性加以验证.

首先,需要确保函数调用不会被攻击者控制.攻击者无法修改硬编码的直接函数调用目的地址,所以只需验证 call 指令的源地址在 CSAT 表中.对于间接函数调用,除了要验证 call 指令源地址在 CSAT 表中以外,还必须验证 call 指令目的地址在 CDAT 表中,或者 call 指令目的地址在指令所在模块的代码段或者 PLT(procedure linkage table)表中.虽然攻击者也可能修改间接函数调用的目的地址到代码段的任意一点,但这样会导致目的地址所在 DFA 的异常从而被检测出来.回顾第 1.1 节 CDAT 表包含了所有直接函数调用的目的地址即所有函数入口地址的一个子集,如果源代码编译时没有优化,则可根据函数的 prologue(push %ebp;mov %esp,%ebp)获得所有函数的入口地址,这样可以直接验证目的地址是否为函数入口地址,但是我们不能依赖于这种假设.

其次,需要确保局部跳转指令不会被攻击者控制.攻击者无法修改硬编码的直接跳转指令的目的地址,所以只需验证间接跳转指令的目的地址在函数内部,其操作需要查找 JSAT 和 JDAT 表.虽然这样可以检测到大多数入侵行为(大多数入侵行为将导致跳转目的地址位于数据段内或栈内),但却不排除攻击者操纵目的地址从而改变子函数调用顺序的行为,对此种异常的检测机制尚在研究中.

2 基于 HFA 模型的入侵检测和分析

综上所述,在操作 HFA 监测进程运行时可能会发生如下几种异常:

1) 某一个局部 DFA 不接受当前的符号(系统调用或者函数调用);
2) 某两个局部 DFA 试图进行动态绑定操作,但是 call 指令源地址不在 CSAT 表中;
3) 某两个局部 DFA 试图进行动态绑定操作,但是 call 指令目的地址非法;
4) 从某个局部DFA M_1返回前一个DFA M_2,但是M_1保存的Asrc和M_2的返回地址不一致;
5) 某一个局部 DFA 的运行时修正操作非法.

如果任意一种异常发生,说明进程的行为与模型发生了背离.不同的异常可能代表了不同类型的入侵事件,虽然在实际的系统中,也许不同的异常分别处理会更适当,但是,目前在原型系统中,为了简便起见,采取了一律杀死进程并报警的做法.基于静态分析的模型理论上可以获得零虚警概率,异常就代表了入侵的发生,我们的 HFA 模型继承了这一特性,所以直接杀死进程的做法并不会导致可用性的降低.

下面简要讨论常见的几种攻击类型和异常类型的关系.

2.1 代码注入攻击(code injection attacks)

代码注入攻击是指攻击者本地或者远程向进程的线性地址空间注入一段可执行的二进制代码,然后通过某种手段修改进程的正常控制流程,使进程执行这段代码,从而达到预定目的的攻击行为.早期文献中所提到的“缓冲区溢出攻击(buffer overflow)”严格意义上是这一类攻击的一个子类,由于攻击者注入代码的位置通常在缓冲区,而且修改进程的正常流程往往需要利用对缓冲区缺乏边界检查的编程错误,所以用“缓冲区溢出攻击”来指代全体.其实,某些代码注入攻击从实现手段上,例如格式字符串误用、double-free 等等与“溢出”都不直接相关,所以,在近来的一些文献中都采用了“代码注入攻击(code injection)”这一比较严格的称谓,本文也采用这个称谓取代“缓冲区溢出攻击”.

代码注入攻击在实现过程上可以分为两步,首先,攻击者利用进程的某些缺陷(如对缓冲区缺乏检查、对格

式字符串处理不当、存在悬疑指针等等)向进程的线性地址空间注入一段代码(shellcode).其次,设法利用函数调用等控制流转移指令转而执行这段代码.为了达到攻击者预想的功能,在 shellcode 中必须包含一些系统调用点或者控制转移指令.

这一类型的攻击包含有传统的“缓冲区溢出攻击”,据统计,仅去年基于代码注入的攻击就占使 CERT/CC 提出建议的所有重大安全性错误的 50%以上.此种攻击会触发 HFA 全部类型的异常,具体地讲,基于栈溢出攻击一般会触发类型为 1,2,4 的异常,基于堆的溢出攻击一般会触发类型为 1,2,3,5 的异常,格式字符串攻击一般触发类型为 1,4,5 的异常,由于 shellcode 所包含的代码不同,所以触发何种异常是不确定的.

2.2 不可能路径攻击(**impossible paths**)

不可能路径[9]是指模型中存在而进程执行中不可能出现的路径,即函数返回到不同的另一个同名调用点,为了实现此种攻击,攻击者需要构造调用栈内保存的函数返回地址,可能通过注入代码来实施操作.

HFA 模型是上下文相关模型,所以可以对此种攻击加以检测,攻击发生时,主要触发类型为 4 的异常.由于攻击者需要注入代码以间接达到目的,所以,可能会附带触发其他类型的异常.

2.3 拟态攻击(**mimicry attacks**)

拟态攻击[15]是指攻击者利用合法的系统调用序列,但是可以通过巧妙地修改调用参数以达到攻击的目的.例如,有如下系统调用序列:

fd=open(/tmp/file.tmp,…);…read(fd,…);…write(fd,…);…close(fd);

攻击者可能修改 open 调用的参数并且不改变序列顺序获得 passwd 文件的内容:

fd=open(/etc/passwd,…);…read(fd,…);…write(1,…);…close(fd);

当前的HFA模型是一种纯粹的基于控制流的模型,在分析序列时忽略调用参数,所以不能对该种攻击进行检测.近年来的一些研究提出了通过参数发现[11]或者是数据流分析[16]检测此种攻击的方法,我们希望在下一步的研究中引入这些机制来克服这个缺点.

2.4 **DoS/DDoS**攻击

DoS/DDoS 攻击按作用机制分为两种类型:资源耗尽式 DoS 攻击和基于异常的 DoS 攻击.前者是指攻击者利用大量的输入耗尽系统资源的攻击方式.例如,建立大量的网络连接,迫使进程处理大文件等等,一般情况下广泛运用于分布式DoS(DDoS)攻击中,由于此种攻击不会触发异常,所以不能被HFA检测.后一种攻击依赖于进程特定的缺陷.例如可能由于整数溢出而导致死循环等等.后一种攻击方式一般需要改变进程的控制流来达到目的,此种类型的 DoS 攻击有可能触发类型为 3 和 5 的异常,从而被 HFA 检测到.

3 确定性和效率的形式化分析比较

我们选取比较有代表性的两种上下文相关模型:PDA模型[9]和基于sDPDA的Dyck模型[11],分别对它们加以形式化分析,并且与HFA模型加以比较.首先给出PDA,sDPDA和DPDA的定义.

3.1 **PDA**,**sDPDA**和**DPDA**的形式定义

一个 PDA 可以用 7 元组 $\{Q,\Sigma,\Gamma,\delta,q_0,z_0F\}$ 来表示,其中:Q 为状态集,Σ为字母表, $q_0\in Q$ 为初始状态, $F\subseteq Q$ 为接受状态集.Γ为栈符号表, $z_0\in\Gamma$ 为初始栈符号,δ为从集合 $Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma$ 到集合 $Q\times(\Gamma\cup\{\varepsilon\})^*$ 的状态转移函数,一个 DPA 有下列 3 种操作:

1) 消耗输入符号: $(p,z)\in\delta(q,a,z),a\in\Sigma\cup\{\varepsilon\}$.如果 $a=\varepsilon$,表示从状态 p 转移到 q 并且不消耗输入符号.如果 $\varepsilon\neq a\in\Sigma$ 表示消耗符号 a 并且从状态 p 转移到 q.
2) Push 操作: $(p,zz')\in\delta(q,a,z),a\in\Sigma\cup\{\varepsilon\}$.与 1)的解释类似,符号 z' 被压入了符号栈.
3) Pop 操作: $(p,\varepsilon)\in\delta(q,a,z),a\in\Sigma\cup\{\varepsilon\}$.与 1)的解释类似,符号 z 从符号栈弹出.

如果一个 PDA 满足如下两个条件,则称为栈确定 PDA(sDPDA):

(sDPDA 条件 1):在ε转移中没有栈操作.即对于所有 $a=\varepsilon$的 $\delta(q,a,z)$ 没有 Push 和 Pop 操作.

(sDPDA条件 2):栈操作仅仅依赖于输入符号和栈顶符号.即 $\forall a\in\Sigma,\forall z\in\Gamma$, 不存在两个状态$q_1,q_1$使得: $(p_1,w_1)\in\delta(q_1,a,z)$, $(p_2,w_2)\in\delta(q_2,a,z)$ 且 $w_1\neq w_2$.

一个 PDA 是确定 PDA(DPDA)当且仅当下列两个条件成立:

(DPDA 条件 1): $\forall q\in Q$, $\forall z\in\Gamma$, $\forall a\in\Sigma\cup\{\varepsilon\}$,如果 $\delta(q,a,z)$ 至多含有一个元素.

(DPDA 条件 2):如果 $\delta(q,a,z)$ 对于 $a\in\Sigma$ 非空,则 $\delta(q,\varepsilon,z)$ 必须为空集.

由上述定义可知,一个 DPDA 必然是一个 sDPDA,一个 sDPDA 必然是 PDA.

3.2 使用PDA,sDPDA和DPDA的异常检测过程分析

假设一个 PDA $M=(Q,\Sigma,\Gamma,\delta,q_0,z_0,F)$,我们定义它的一个配置 c 为 2 元组(q,r),其中 r 是当前栈上的符号串.以 $c\xrightarrow{a}c'$ 表示 M 在输入了符号 a 之后由配置 c 变为了配置 c' .

PDA 的异常检测过程可以描述为:假设 M 是一个 PDA,模型当前的状态是配置 c 的一个集合(由于 PDA 可能不确定,输入一个符号后可能有多个分支),模型的初始状态为 $\{(q_0,z_0)\}$,假设集合C是模型处理了序列 $w\in\Sigma^*$ 后的状态,并且下一个输入符号为a,模型预测到下一个状态C_a为集合 $\{c'\mid\exists c\in C,c\xrightarrow{a}c'\}$,如果C_a为空集,则发生了一个异常,否则,模型切换到状态C_a并且继续处理下一个输入符号.在接受到一个输入符号a后,模型的状态可能是无限集合:假设当前状态C= $\{(p,z)\}$,并且存在转移函数: $\delta(p,\varepsilon,z)=\{(p,zz)\}$, $\delta(p,a,z)=\{(q,\varepsilon)\}$,易见 C_a= $\{(q,z^i),i\geq 0\}$ 是无限集合,所以PDA有可能在一步操作中就耗尽所有的系统资源.

如果一个 PDA 是 sDPDA,根据前面的两个 sDPDA 条件,模型某一个状态的所有元素必然有相同的栈字符串,且 $C\in Q\times\Gamma^*$,所以,C的秩最多不会超过Q的秩.可见sDPDA每一步操作的时间复杂度(从C_a中查找特定配置c)和空间复杂度(C_a的元素个数)都是 $O(|Q|)$.进一步,如果一个PDA是DPDA,则所有的状态都只含有一个元素,每一步操作的时间复杂度和空间复杂度都是 $O(1)$.

文献[12]证明,Dyck模型和 VPStatic模型本质上都是sDPDA模型,下面证明HFA模型本质上是一个DPDA.

3.3 HFA的形式定义和证明

假设我们由CFG获得了一系列的DFA: $M_0,M_1,\ldots,M_n$, $M_i=\{Q^{M_i},\Sigma^{M_i},\delta^{M_i},q_0^{M_i},F^{M_i}\}$,并且我们假设进程运行时从M_0开始操作,结束于某几个M_i, $i\in I_F$ 的接受状态(假设 1).并且记集合 $C^{M_i}\subseteq\Sigma^{M_i}$ 是每一个DFA中的函数调用符号(见第 1.2 节).由第 1.3.1 节的规定可知,任意两个DFA之间都是独立的,所以有: $Q^{M_i}\cap Q^{M_j}=\varnothing$, $C^{M_i}\cap C^{M_j}=\varnothing$ 对所有 $i\neq j$ 都成立.

我们按照下述规则构造 PDA $M=\{Q,\Sigma,\Gamma,\delta,q_0,z_0,F\}$:

$Q=\bigcup_i Q^{M_i}$ (PDA 状态集合是所有 DFA 状态的并集).

$\Sigma=\bigcup_i \Sigma^{M_i}$ (PDA 输入符号是所有 DFA 输入字母表的并集).

$\Gamma=\bigcup_i C^{M_i}\bigcup\{\varepsilon\}$ (PDA 栈符号是所有 DFA 函数调用符号的并集).

$q_0=q_0^{M_0}$, $z_0=\varepsilon$, $F=\bigcup_{I_F}F^{M_i}$ (按照假设 1).

下列定义假设对所有的 $i\in[0,n+1]$,有:

$\delta(p_i,a_i,z)=(q_i,z)$, $p_i,q_i\in Q^{M_i}$, $a_i\in\Sigma^{M_i}-C^{M_i}$ 且 $\delta^{M_i}(p_i,a_i)=q_i$ (消耗输入符号),

$\delta(p_i,c_i,z)=(q_i,zc_i)$, $p_i,q_i\in Q^{M_i}$, $c_i\in C^{M_i}$ 且 $\delta^{M_i}(p_i,c_i)=q_i$ (动态绑定开始),

$\delta(p_i,c_i,c_i)=(q_i,\varepsilon)$, $p_i,q_i\in Q^{M_i}$, $c_i\in C^{M_i}$ 且 $\delta^{M_i}(p_i,c_i)=q_i$ (动态绑定结束).

定理 **1**. HFA 是 DPDA.

证明:易见“动态绑定开始”形式上对应于 Push 操作,“动态绑定结束”对应于 Pop 操作,所以 HFA 是一个 PDA.由于对所有的 δ^{M_i} , $\forall i\in[0,n+1]$ 都是 DFA 的转移函数,且由 DFA 的独立性有 $Q^{M_i}\cap Q^{M_j}=\varnothing,i\neq j$,可知 $\forall q\in Q$, $\forall z\in\Gamma$, $\forall a\in\Sigma\cup\{\varepsilon\}$, $\delta(q,a,z)$ 至多含有一个元素,所以 DPDA 条件 1 成立.又由于不存在形如 $\delta(q,\varepsilon,z)$ 的转移函数,所以 DPDA 条件 2 也成立.由此可知 HFA 是 DPDA. □

定理 1 表明,HFA 本质上是一个 DPDA,每一步操作的时间复杂度和空间复杂度都是 $O(1)$.但在实际中,由于存在运行时修正操作,所以,如果某一个 DFA 存在间接跳转,操作该 DFA 可能会花费更多的资源.由于对 DFA 的运行时修正操作只是一种对于特殊情况的补救,而且 DFA 修正后还是一个 DFA 不影响整体的 HFA 结构,所以我们不把这种操作包含在形式模型中.

4 基于 Linux 的原型系统

4.1 原型系统的体系结构

如图 6 所示,原型系统由 3 个模块构成.

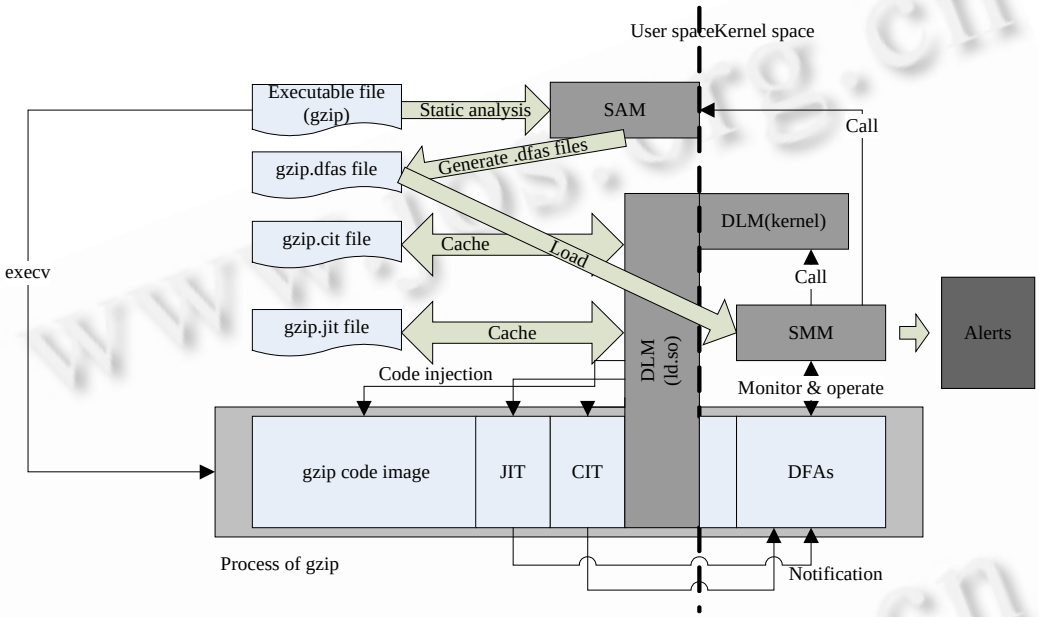

Fig.6 Architecture of prototype

图 6 原型系统的体系结构

静态分析器 SAM(static analyze module).SAM 是一个运行于用户态的工具程序,它读取可执行模块(可执行文件或者动态链接库)p 并且生成对应的 DFA 定义文件 p.dfas.在一个*.dfas 文件中包含有对该可执行文件中每一个函数的 DFA 描述和 CSAT,CDAT,JSAT,JDAT 表.当执行一个程序时,如果没有对应可执行模块的*.dfas 文件,则内核自动调用该工具生成.为了避免程序启动时间过长,管理员也可以手动地为所有可执行模块应用该工具.

动态链接模块 DLM(dynamic linking module).DLM 由两部分组成,一部分位于内核中,一部分位于经过修改的 ld.so(ELF 文件链接/解释器)文件中.该模块的功能是在可执行模块装入内存后在进程用户态地址空间内分配并参考 CSAT,CDAT 和 JSAT,JDAT 配置 CIT(call indirection table)和 JIT(JMP indirection table).DLM 分别把可执行模块中的每条 call 指令和每条间接跳转指令替换为跳转指令指向 CIT 中或者 JIT 中的某条记录.CIT 的每条记录保存有原来的 call 指令和若干代码通过保留系统调用号和内核交互,在记录结尾返回原有 call 指令所在地址,JIT 与此类似.通过 CIT 表和 JIT 表,系统可以追踪所有的函数调用行为以及间接跳转行为.我们使用 Linux 废弃的保留系统调用号作为系统监测动态绑定操作和运行时修正的手段.与 SAM 类似,系统采用了缓存机制使用文件*.cit 和*.jit 保存每一个可执行模块的 CIT 表和 JIT 表.

系统调用监测模块 SMM(system-call monitor module).SMM 位于内核中,在 DLM 模块的替换工作结束、进程开始执行前载入对应的*.dfas 文件,在进程的内核态线性地址空间内初始化各个局部 DFA,并且随着进程执行监测系统调用(保留系统调用除外)和操作 DFA.SMM 通过 CIT 表和 JIT 表中的保留系统调用获知进程的 DFA

动态绑定操作,并在适当时机调用 DLM 进行 DFA 的运行时修正,如果发生异常,则报警并作出适当的响应动作(杀死进程).所有的局部 DFA 放置于内核内可以防止攻击者恶意操作自动机.

4.2 对非标准控制流跳转的处理

非标准控制流跳转包含对 POSIX 标准的库函数 siglongjmp,longjmp 等调用所导致的非局部跳转和由于接收到信号(signal)而导致的控制流暂时跳转.前者发生时机在静态分析时即可确定,当发生跨函数的非局部跳转时,除了转换对应的局部 DFA 以外,还根据当前函数调用栈的内容替换相应的 DFA 运行时状态栈.进程接收到信号的时机是随机的,我们的处理方法是,在进程接收到信号时假设在当前运行位置调用对应的信号处理函数,转换 DFA,并且使原 DFA 忽略对应的信号处理函数的调用符号(DFA 状态栈不包含信号处理函数的函数调用符号).

4.3 对动态链接库的支持

HFA 模型先天具有的模块化特性可以使我们方便地为每个动态链接库定义自己的一组 DFA,然后随着动态链接库被载入系统和内存中,已有的 DFA 作动态绑定操作.操作这些 DFA 与操作主执行文件中的 DFA 没有区别.据我们所知,其他模型由于使用“整体”式的结构,不能对动态链接库提供很好的支持.

为了提高效率,管理员可以预先手工调用 SAM 为系统中的动态链接库生成 DFA 描述文件,这些文件只和硬件平台和操作系统版本相关,具有良好的可移植性.

4.4 对进程/线程的支持

在 Linux 平台上使用系统调用 fork 生成子进程执行分支任务,使用系统调用 exec 执行新的可执行文件,线程(thread)使用轻量级的进程(LWP)实现.我们的原型系统提供了对多进程的支持,在 fork 调用结束后,父子进程共享当前的一组 DFA,并且 SMM 把父进程当前 DFA 的运行时状态栈复制给子进程,此后,父子进程进行各自的 DFA 操作.如果子进程在 fork 调用后又执行了 exec 调用,则子进程变成了一个全新的进程,DLM 和 SMM 重新建立子进程的运行环境并操作全新的一组 DFA.

5 实 验

5.1 模型的精确性和效率

为了检验HFA的精确性,我们采用了Wagner和Dean[9]提出、并被后继研究者广泛使用的一种度量方法:平均分支因子(average branching factor).平均分支因子是一种表示攻击者可能向序列中插入危险系统调用的机会的度量值.平均分支因子等于一个模型在监测序列过程中记录下所有的系统调用数目除以模型的操作次数,平均分支因子越小说明攻击者插入危险系统调用的机会越少.为了检验HFA的效率,我们简单地比较一个程序在不使用入侵检测系统时的运行时间和加载了入侵监测系统后的运行时间.

我们比较了 NFA 模型、PDA 模型、Dyck 模型和 HFA 模型的精确性和效率,为了公平起见所有测试用程序都采用了静态链接,对 HFA 模型还给出了动态链接的测试结果.表 1 是实验中使用的目标程序的一些特性统计,图 7 和图 8 是平均分支因子和运行时对进程运行时间的影响.

Table 1 Statistical properties of objective applications (*static linked)

表 **1** 目标程序的特性统计(*数值均为静态链接测试结果)

Application	Payload	Functions*	Instructions*	System calls*	Functions calls*
gzip	Compress a text file (13MB)	894	56 740	96	2 476
ps	Report status of all process	963	59 845	96	3 330
ls	List all files of Linux kernel source	766	49 937	57 828	175 828
cat	display the content of linux kernel source (total 145MB)	660	47 563	58 422	163 812

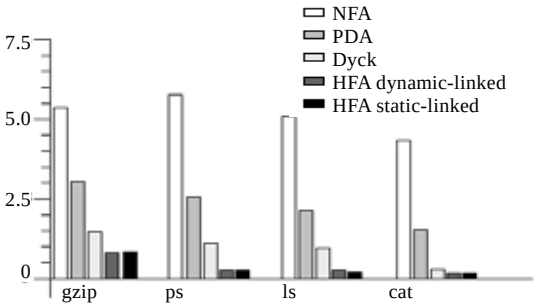

Fig.7 Result: Average branching factor

图 7 平均分支因子实验结果

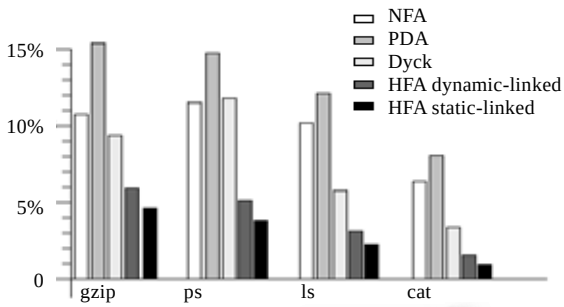

Fig.8 Result: Run time

图 8 运行时间实验结果

从实验结果可以看出,HFA 模型的精确性和效率比其他 3 种模型要好,其中,平均分支因子小于 0.5,在目标程序动态链接的情况下,使程序的运行时间减慢大约 7%,在目标程序静态链接的情况下,由于省去了很多 DFA 运行时修正的操作,所以影响还会更小.

5.2 模型的有效性

有效性是指模型的入侵检测能力,对有效性没有一个统一的度量.我们选择 Unix/Linux 平台上广泛使用的一种 FTP 服务器——wu-ftpd-2.60 作为测试对象.wu-ftpd-2.60 存在多个漏洞,我们选择了一些具有代表性的漏洞,并使用现有的攻击程序加以攻击,检测结果见表 2.

Table 2 Typical vulnerabilities of wu-ftpd-2.60 and test results (*exploit lead to various exceptions)

表 **2** wu-ftpd-2.60 的一些典型漏洞和攻击检测结果(*不同的攻击程序导致不同的异常)

Vulnerability CVE	Vulnerability type	Description	Exception type
CVE-2000-0573	Format string overflow	SITE EXEC format string remote overflow	1
CVE-2001-0187	Format string overflow	Debug mode format string overflow	1,4*
CVE-2001-0550	Heap overflow	Heap vorruption in file glob	1,2*
CAN-2003-0466	Integer overflow	fb_realpath remote off-by-one	4
(BID8668)	Stack overflow	SockPrintf remote stack over run	1
CVE-2004-0148	Logic error	Restricted gid gain access	Can not be detected
CVE-2004-0185	Stack overflow	S/key remote stack based buffer over run	1

实验结果表明,原型系统对基于栈溢出、基于堆溢出以及基于格式字符串滥用的攻击能够检测.对于编号为 CVE-2004-0148 的漏洞进行的攻击不会改变调用序列,类似于拟态攻击,未能加以检测.我们无法测试所有类型的攻击,但是表 2 的实验结果部分地证明了系统的入侵检测能力.

前述第 2 节中,3 和 5 两种异常情况在理论上是存在的,但是如果要产生这两种异常,需要构造十分复杂的攻击负载,同时,被攻击程序或软件必须具备相应的可以利用的漏洞(对负载长度也有限制).在现实中我们还没有发现有软件存在类似的可以利用的漏洞.

6 总结和展望

本文提出的 HFA 模型是一种动态-静态混杂的模型,比先前的同类上下文相关自动机模型更接近确定下推自动机(DPDA),从而可以获得更高的精确性和效率.而且 HFA 由于先天的模块化特性更适用于动态链接的应用环境中.HFA 模型解决了纯静态分析模型不能解决的间接函数调用和跳转问题.虽然如此,还有一些难题在当前阶段未能解决,主要有:

(1) 运行时 DFA 修正的合法性验证中,攻击者有可能操纵间接跳转的目的地址从而改变子函数调用顺序,虽然在实验中没有发现此种现象,但是不能排除这种攻击一定不会发生.一个设想是通过数据流分析预测目的地址,引入更严格的校验.

(2) 由于目前的 HFA 模型没有考虑系统调用的参数,所以不能有效地检测拟态攻击,在今后的研究工作中,我们将借鉴数据流分析、参数发现等技术来弥补这一缺陷.

References:

[1] Denning D. An intrusion detection model. IEEE Trans. on Software Engineering, 1987,13(2):222−232.

[2] Forrest S. A sense of self for UNIX processes. In: Proc. of the IEEE Symp. on Security and Privacy. Oakland: IEEE Press, 1996. 120−128. http://www.cs.unm.edu/~forrest/publications/ieee-sp-96-unix.pdf

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

[4] Helman P, Bhangoo J. A statistically based system for prioritizing information exploration under uncertainty. IEEE Trans. on Systems, Man and Cybernetics, Part A: Systems and Humans, 1997,27(4):449−466.

[5] Lee W, Stolfo SJ. Data mining approaches for intrusion detection. In: Proc. of the 7th USENIX Security Symp. San Antonio, 1998. 26−40. http://www.usenix.org/publications/library/proceedings/sec98/full_papers/lee/lee.pdf

[6] Lee W, Stolfo SJ, Chan PK. Learning patterns from UNIX process execution traces for intrusion detection. In: AAAI Workshop on AI Approaches to Fraud Detection and Risk Management. AAAI Press, 1997. 50−56. http://www.cc.gatech.edu/~wenke/papers/osid_paper.ps

[7] Sekar R, Bendre M, Bollineni P, Dhurjati D. A fast Automaton-Based method for detecting anomalous program behaviors. In: IEEE Symp. on Security and Privacy. Oakland: IEEE Press, 2001. 144−155. http://www.cc.gatech.edu/~wenke/ids-readings/automaton.pdf

[8] Feng HH, Kolesnikov OM, Fogla P, Lee W, Gong W. Anomaly detection using call stack information. In: Proc. of the 2003 IEEE Symp. on Security and Privacy. Oakland: IEEE Press, 2003. 62−75. http://www-unix.ecs.umass.edu/~gong/papers/ok_idpc.pdf

[9] Wagner D, Dean D. Intrusion detection via static analysis. In: Proc. of the IEEE Symp. on Security and Privacy. Oakland: IEEE Press, 2001. 156−168. http://www.csl.sri.com/users/ddean/papers/oakland01.pdf

[10] Giffin J, Jha S, Miller B. Detecting manipulated remote call streams. In: Proc. of the 11th USENIX Security Symp. San Francisco: 2002. 61−79. http://www.cs.wisc.edu/wisa/papers/security02/gjm02.pdf

[11] Giffin J, Jha S, Miller B. Efficient Context-Sensitive intrusion detection. In: Proc. of the 11th Network and Distributed System Security Symp. San Diego, 2004. http://www.cs.wisc.edu/wisa/papers/ndss04/dyck.ps

[12] Feng HH, Giffin JT, Huang Y, Jha S, Lee W, Miller BP. Formalizing sensitivity in static analysis for intrusion detection. In: IEEE Symp. on Security and Privacy. IEEE Press, 2004. 194−208. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.930&rep=rep1&type=pdf

[13] Gopalakrishna R, Spafford EH, Vitek J. Efficient intrusion detection using automaton inlining. In: 2005 IEEE Symp. on Security and Privacy. IEEE Press, 2005. 18−31. http://www.cs.purdue.edu/homes/jv/pubs/sp05.pdf

[14] Hopcroft J. An *n*log*n* algorithm for minimizing states in a finite automaton. Theory of Machines and Computations. New York: Academic Press, 1971. 189−196.

[15] Wagner D, Soto P. Mimicry attacks on host-based intrusion detection systems. In: Proc. of the 9th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2002. 255−264. http://www.xcf.berkeley.edu/~paolo/ids-res/mimicry.pdf

[16] Sandeep B, Abhishek C, Sekar R. Dataflow anomaly detection. In: Proc. of the 2006 IEEE Symp. on Security and Privacy. Washington: IEEE Press, 2006. 48−62. http://www.seclab.cs.sunysb.edu/seclab/pubs/papers/ieee06.pdf

李闻(1980－),男,安徽蚌埠人,博士,主要研究领域为安全信息系统,入侵检测系统.

连一峰(1974－),男,博士,副研究员,博士生导师,主要研究领域为网络安全.

戴英侠(1942－),女,教授,博士生导师,主要研究领域为信息安全.

冯萍慧(1979－),女,博士生,主要研究领域为脆弱性分析.