基于语义的恶意代码检测算法研究①

Algorithm of Detection Malware Based on Semantics

王晓洁[1] 王海峰[1,2]

(1.临沂师范学院 信息学院 山东 临沂 276002；2.上海理工大学 管理学院 上海 200093)

摘 要： 代码迷惑是一种以增加理解难度为目的的代码变换技术，主要来保护软件免遭逆向分析。恶意代码的作者为了躲避检测经常采用代码迷惑技术对程序进行转换。但是商用反病毒软件采用基于特征码的模式匹配技术而忽略了恶意代码的语义，因此最容易受到代码迷惑或病毒变种的攻击。文章中提出一种基于语义匹配的检测算法，能准确的检测出经过代码迷惑处理的恶意代码。该方法应用数据流分析技术，以变量定义使用链为单元检测每个模板及程序节点。最后通过部分实验展示了原型系统的检测效果。

关键词： 代码迷惑 语义模板 定义使用链 恶意代码

1 引言

各类恶意代码的泛滥严重威胁到信息系统的安全，目前主要依靠反病毒软件和入侵检测系统等手段实施检测和防御。而商业反病毒软件采用成熟的特征码匹配技术，在特定格式的二进制文件中进行某种正则表达条件的匹配。该检测算法忽视了恶意代码的语义特征，必须频繁更新特征码数据库。而各类攻击或病毒变种的出现以及代码迷惑技术的应用，使得这些检测手段的作用越来越有限。代码迷惑是一种增加程序理解难度为目的的程序变换技术，比如插入垃圾指令、调整指令顺序、寄存器再分配等[1]。代码迷惑虽然改变了程序指令的字节级特征，但是并没有改变程序的语义。

恶意代码作者进行代码迷惑时保持了程序转换前后的语义信息，或者程序转换前后语义等价。恶意代码具备相似的行为，实现这些行为的代码语义等价。比如大多数具有二维变形病毒代码中都有自解密循环；很多蠕虫通过电子邮件进行传播的行为导致其代码中存在搜索邮件地址的代码。这些行为具有明显区别正常程序的语义信息，因此提取恶意代码中语义信息为检测特征是个有效方法。目前国内恶意代码语义模型的研究还很少，显然对于语义模型的方法重视度还不够，所以在此阐述恶意代码的语义分析理论，并希望以此引起对恶意代码语义研究的重视。

本文首先介绍程序代码的语义表示理论，精简为指令、运算量、函数等抽象形式，主要为反汇编代码转化为中间语言做理论支持；然后介绍基于语义检测的体系和算法思想；接着给出部分实验结果并简单分析；最后总结全文，分析下一步工作。

2 程序的语义理论

2.1 语义形式化

程序的语义以模块为一个单元，单元可建立对应语义模板。语义模板 T 由三元组构成，即 $T=<I_T, V_T, C_T>$。I_T 为指令序列。V_T 和 C_T 分别代表指令序列中的变量与符号常量集合，其中符号常量包括两种类型：n 元函数 F(n)及 n 元谓词命题 P(n)。n 元函数 F(n)是由 n 个自变量到一个因变量的映射，n 元谓词命题 P(n)由 n 个自变量到结果为“真”或“假”的映射[2]。

程序代码的运行直接改变内存的值，计算机内存可表示为函数 M，该函数是内存地址集合到值集的映射，M[a]表示地址 a 处的值。语义模板的执行上下文：

① 收稿时间:2008-12-10

对于给定模板 T，将符号常量集合 CT 赋予具体的值后就为该模板的执行上下文，记作 ECT。ECT 是自定义域为 CT 的函数，对于 CT 集合中的符号常量 c，其类型必须和 ECT(c)保持一致。

语义模板 T 的状态表示为 ST，该状态由三个元素组成：变量、指令地址指针、内存集合的值。因为代码执行到某一状态主要是对内存值的影响和改变，因此三元组(val_T, pc_T, mem_T)能准确表示出模板执行状态，其中 val_T 表示变量集合 V_T 的变量值，pc_T 为程序指针寄存器的值，memT 是对应内存单元中的值。具体到某一指令序列 I，也需要表示该指令序列的状态 I=(val, pc,mem)。此处的 val 是指寄存器的值；pc 是执行该指令序列 I 时程序计数器的值；mem 则是指令序列执行到某一时刻内存单元中的值。为了便于讨论 S_T 是语义模板的状态空间，而 s 则是指令序列的状态空间。

2.2 语义匹配

假设给定语义模板T一个具体的执行上下文EC_T，该语义模板的初始状态为 s_T，执行一条指令 I 后，状态转换为 $s1_T$,并且该指令的执行导致计算机系统产生了一个事件 e，事件 e 或许为一个操作系统服务调用或为一个内核陷阱门[3]。此过程形式化表示为 $s_T \xrightarrow{e} s1_T$。为了保持一致性，当指令 I 执行并未产生系统事件则定义为空事件，记作 e(i)=null。对于模板的初始状态 $S0_T$，在执行环境 ECT 中生成如下一个迹：$\sigma(T, EC_T, S0_T) = S0_T \xrightarrow{e_1^T} S1_T \xrightarrow{e_2^T} \cdots\cdots$.（i≥1, Si_T 表示模板在执行环境ECT中执行指令I后的状态；ei_T 是第指令 i-1 产生的事件)。对于具体代码也有一个执行的迹表示为:$\sigma(I,S_0)=S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} \cdots\cdots$。

定义 1：假设一个程序状态 S_0，执行环境 EC_T，语义模板状态 $S0_T$，如果 $mem(S0_T)=mem(S_0)$,并且模板的迹$\sigma(T, EC_T, S0_T)$和程序的迹$\sigma(I, S_0)$都是有限状态，因此，给定指令序列 I 必然表现出一定的程序行为，该行为的语义可以由语义模板描述。

条件 1：$\sigma(T, EC_T, S_{0T}) = S_{0T} \xrightarrow{e_1^T} S_{1T} \xrightarrow{e_2^T} \cdots\cdots \xrightarrow{e_k^T} . S_{kT}$；$\sigma(I, S_0) = S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} \cdots\cdots \xrightarrow{e_r} S_r$。令 $affected(\sigma(T, EC_T, S0_T))$ 为受影响的内存集合，状态变化后该集合中元素的内容随之改变。例如：$mem(S0T)[a] \neq mem(SkT)[a]$。$\forall a \in affected(\sigma(T,EC_T,S_{0T}))$, $em(S_{0T})[a]=mem(S_r)[a]$ 成立。条件 1 的本质是模板 T 的执行与具体指令序列 I 的执行对受影响的内存集合的影响相同。条件 2：如果忽略空事件，事件序列串 $<e_0^T,\cdots,e_k^T>?<e_0,\cdots,e_r>$。同时事件 e_1，e_2. 变量与返回值相同，则事件 $e_1=e_2$。条件 3：若 $pc(S_k^T)$? $affected(\sigma(T, EC_T, S_0^T))$,则 $pc(S_{rT}) \in affected(\sigma(T, EC_T,S_{0T}))$。该条件为程序地址指针指向受影响内存区域末端时，若执行了第 I 条指令后，PC 寄存器值仍在受影响内存区中。

定义 2：核心内存集合。由于受影响内存区域在某些场合下过于严格，比如某一行为模板使用特定内存去存放临时变量，该内存单元不能作为相等性的判定依据[4]。为此核心内存集合是受影响内存集合的子集。下面的算法讨论都局限在核心内存集合中。

定义 3：若程序 P 由指令序列 I 构成，而指令序列 I 具有特定行为，并且程序 P 满足语义模板 T。将判定程序是否满足行为模板 T 的问题称为模板匹配。

由核心内存集合的定义知，各类不同的程序代码对核心内存集合的影响效果相同，对于这类不同的程序代码可以提取语义相同的模板，根据定义 3 知这些程序代码与此语义模板相匹配，所以程序的行为语义完全可以形式化表示，并能提取为相应的语义特征模板。

3 语义检测

3.1 体系结构

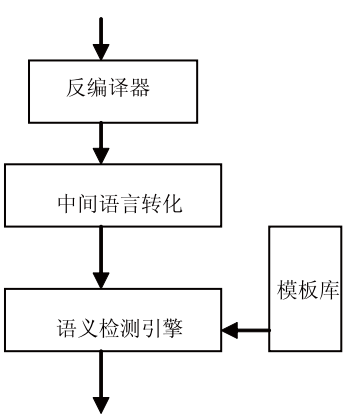

图 1　体系结构图

该语义检测的原型系统结构如图-1 所示。输入为二进制执行代码，经过反编译程序转换为汇编语言代码。原型中使用反编译器 IDApro5.0 的反编译结果，为了提高通用性，用 Perl 语言实现了中间代码转换模块，经过转换后变为与机器指令系统无关的中间代码。最后调用语义检测引擎搜索模板库，如果匹配则提示检测到恶意代码。

3.2 语义匹配算法

设 $T=\{T_1,T_2,\cdots,T_k\}$为语义模板集合，$P=\{P_1,P_2,\cdots,P_m\}$程序代码片段集合，Pi 是集合 P 的划分。设置阈值 θ 与 μ 分别用来判定正常程序与恶意代码，这两个阈值通过训练学习确定。其中语义模板的每个元素 T_i 都由经验确定权值 W_i。语义匹配算法描述如下：

输入：中间语言的程序段，以定义语义模板

输出：返回 true 表示检测到恶意代码，false 为正常程序，unknown 无法确定

```
Decision(P,T)
{
  for each Ti in T //遍历模板集合
  {
    for each Pi in P
    {
     if(match(Ti, Pi))
       W = W+Wi; //累计权值
    }
   }
if(W>= θ) //超过恶意代码阈值
  return true;
if(W<=μ) //低于正常程序阈值
  return false;
else
  return unknown;
}
```

算法关键部分为 match(T_i, P_i)，该函数实现了模板与某段程序代码片段的语义匹配。检测是否匹配的方法是基于模板或程序中的变量定义-使用链。程序或模板中的定义-使用链按照以下方法确定[2]：(1)在节点序列$<n_1,n_2,\cdots,n_k>$中，n_i 与 n_{i+1} 存在偏序关系；(2)在该序列中不存在中间节点对 $def(n_1)$ 变量集合中的变量重新定义；(3)节点 n_1 中定义的变量在节点 n_k 中使用，即 $def(n_1)\cap use(n_k)\neq\varnothing$。设 $T_i=\{n_1,n_2,\cdots,n_p\}$，$P_i=\{m_1,m_2,\cdots,m_q\}$即模板与程序都分别由若干个节点组成。match($T_i$, P_i)描述如下：

输入：模板节点序列，中间代码节点序列

输出：true 匹配，false 不匹配

```
match(Ti, Pi)
{
 for each ni in Ti //确定模板中的 DU 链
 { if(def(ni)∩use(nm) ≠)
     Tdu ∪Tdui;
     ni= nm+1;
     i++;
}
 for each mi in Pi//确定程序中的 DU 链
{ if(def(mi)∩use(mn) ≠
   Pdu ∪Pdui;
   mi= mn+1;
   i++;
}
 for each Tdui in Tdu//基于 DU 链检测
  if((match(Tdui, Pdui)==false)
   return false;
  else
   i++;
  return true;
}
```

4 实验结果

实验在 Windows2003 环境下进行，主要因为针对 Windows 平台的恶意代码比较多，容易构造实验过程。首先提取典型恶意代码中的语义模板，在 Bagle、Sober、Loicer、Foroux、Zaka、Cornad、Driller、求职信八种病毒及 32 个变种中提取 11 个语义模板，比如解密功能、暴力搜索电子邮件地址、应用 SMTP 发送简单邮件、搜索系统文件、花指令迷惑、代码段搬移等。统计这 11 种功能出现的频率，根据频率赋予相应的权值，权值的等级为 0.05。

然后在操作系统与应用程序中选择 400 个大小在 12KB 到 90KB 的程序，将初始的 μ 设置为 0.5，若模板库中的 11 个模板出现一次不匹配就减 0.05。统计 400 个正常程序的 μ 值取平均得 μ=0.07。再用这 32 个病毒匹配 11 个语义模板，求平均值得出 θ=0.525。

4.1 对代码模糊变换的检测

针对代码模糊变换采用两种形式进行实验，对于有源代码的病毒进行代码修改，比如插入垃圾指令、随机改变寄存器等技术。对于无源代码病毒采用加壳变换的方法，实验结果如表 1 所示。现有的反病毒软件均不能很好的检测出恶意代码的变种。比如瑞星能够检测出某种病毒，但是对该病毒简单加壳后就无法正确检测。

表 1　病毒实验结果分析

病毒名称	瑞星	NOD32	本算法
Sober	×	×	√
Netsky	×	×	√
Beagle	×	×	√
Win32.Loicer	×	×	√
Funlove	×	×	√
求职信	×	√	√
Win32.Foroux	×	×	√
Win32.Cornad	×	×	√
Win32.Driller	×	×	√
Win32.Zaka	×	×	√

注：√表示检测成功，×表示检测失败。(病毒均经模糊处理)

4.2 误报检测分析

对于恶意代码检测误报率十分重要，实验中选取了 2000 个正常的程序文件和 200 个经过代码变换或者加壳处理的正常程序，实验结果是未出现误报。但是将阈值 μ 降为 0 后，出现 24 例误报，由此可见 μ 的选择也很重要。

5　总结

本文提出的语义模板利用了数据流分析的手段检测恶意代码的模糊变形或病毒变种。该算法通过分析恶意代码行为建立语义模板库，通过实时判断程序代码段与语义模板的相似性来决定是否为恶意代码。该方法具有以下特点：(1)不需要大规模数据集进行训练就可以检测未知的恶意代码；(2)在正常程序未遭完整性破坏时误报率为 0。以后的研究方向是提高检测效率，同时增加中间语言的表达能力，增加其运算符的多样性。

参考文献

1 Ogiso T, Sakabe Y. Software obfuscation on a theoretical basis and its implementation. IEEE Transactions on Fundamentals, 2003,E86-A(1):176-183.

2 陈意云.编译原理.北京:高等教育出版社, 2003.

3 Mihai christodorescu, Somesh Jha. Static analysis of executables to detect malicious patterns. Proceedings of the 12th USENIX Security Symposium, USENIX Association, Berkeley, CA,USA, 2003:169-185.

4 Kruegel C, Robertson W, Vigna G. Detecting kernel-level rootkits through binary analysis. Proceedings of the 20th Annual Computer Security Applications Conference,Tucson,AZ,2004.