第 34 卷 第 5 期 计 算 机 工 程 2008 年 3 月
Vol.34 No.5 Computer Engineering March 2008
·安全技术· 文章编号：1000—3428(2008)05—0169—03 文献标识码：A 中图分类号：TP309

基于归一化的变形恶意代码检测

金 然，魏 强，王清贤
(信息工程大学信息工程学院，郑州 450002)

摘 要：许多未知恶意代码是由已知恶意代码变形而来。该文针对恶意代码常用的变形技术，包括等价指令替换、插入垃圾代码和指令重排，提出完整的归一化方案，以典型的变形病毒 Win32.Evol 对原型系统进行测试，是采用归一化思想检测变形恶意代码方面的有益尝试。
关键词：变形恶意代码；归一化；恶意代码检测

Metamorphic Malware Detection Based on Normalization

JIN Ran, WEI Qiang, WANG Qing-xian
(Information Engineering Institute, Information Engineering University, Zhengzhou 450002)

【Abstract】Much of unknown malware comes from transformed known malware. This paper proposes a complete normalization scheme to resolve the common transforming methods, including identical instructions substitution, garbage code insertion and code reordering. It implements a prototype system and a test to the system is conducted using Win32.Evol, a typical metamorphic virus. It makes a useful attempt to adopt normalization to detect metamorphic malware.
【Key words】metamorphic malware; normalization; malware detection

1 概述

据统计，新出现的恶意代码大部分是在原有恶意代码的基础上修改转换而来[1]，其中，一种较先进的转换技术称为变形[2](metamorphism)技术，即在保持语义等价的前提下，通过程序变换改变代码形态。近些年出现的变形病毒(如 Win32.Evol, Win32.Metaphor等)能自动完成这一过程，变换前后的代码不论是在静态文件中，还是动态执行在内存里，都不呈现相同特征码，这给基于特征码的检测方法带来了极大挑战。

采用归一化方法，结合传统的基于特征码检测技术是应对变形恶意代码的一种方案，其主要思想是将待检测代码进行归一化处理，再根据特征码匹配进行检测。本文针对恶意代码常用的变形技术构造了一个完整的归一化模型。文献[3]使用词重写系统 TRS 对代码进行归一化处理。该方法的最大缺点是须为每种变形引擎构造相应 TRS，而且该构造依赖于对特定引擎的分析，对于未知变形引擎，无法构造相应 TRS，从而无法检测经该引擎变形而来的恶意代码。本文介绍的归一化模型并不针对特定变形引擎，具有较好的普遍适用性。

2 恶意代码常用的变形技术

指令重排、等价指令替换和插入垃圾代码这 3 种变形方法简单有效，易用自动化引擎实现，许多变形恶意代码，如 Win95.Zperm, Win32.Evol, Win32.Zmist 等采用的都是这些方法，因此，研究针对这 3 种变形技术的归一化有重要意义。

(1)等价指令替换。将指令替换为在语义上等价的其他指令(序列)。图 1(1)是 Win32.Evol 的代码片段，它的变形引擎会将其中的 movsd 替换为一指令序列得到图 1(2)。容易看出变换前后的代码在语义上等价。

(2)插入垃圾代码。增加冗余操作指令，但不改变原有语义。例如在图 1(2)中 call Ln 前增加一条给 eax 的赋值指令得到图 1(3)。由于 call 返回后会给 eax 重新赋值，因此增加的指令并不会改变原有程序的语义。

(3)指令重排。改变代码中指令位置，同时插入 jmp 指令保持相对执行顺序。将图 1(3)中各指令位置打乱，并插入相应的 jmp 指令后得到图 1(4)，从指令执行顺序看，变换前后是一样的。

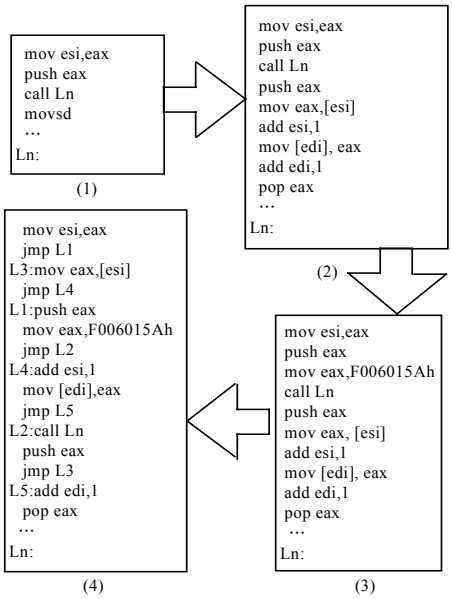

图 1 恶意代码常用的变形技术

3 基于归一化的检测系统框架

图 2 显示了基于归一化的恶意代码检测系统框架。已知 *M* 为恶意代码，将其送入归一器进行归一化，然后提取代码

作者简介：金 然(1979－)，男，博士研究生，主研方向：网络安全；魏 强，博士研究生；王清贤，教授、博士生导师
收稿日期：2007-04-06 **E-mail：**jinran17@163.com

特征；对于待检测的代码 $C1,C2,\cdots, Cn$，同样先送入归一器进行归一化，然后提取输出结果特征与 M 的特征进行匹配，以此来判断待检测代码是否为 M 的变种。

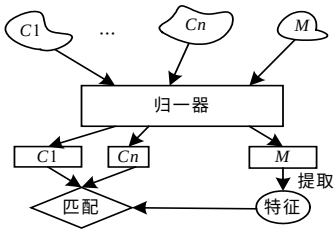

图 2　基于归一化的检测系统

4　归一器的构造

归一器以二进制代码作为输入，以规范的二进制代码作为输出，为提高处理效率，先对输入反汇编，再针对各变形技术采取相应的归一化措施。

4.1　反汇编

基于以下优点，二进制代码在输入归一器后应先反汇编，然后进行后续处理。

(1)利于控制转移指令的处理。在归一化处理中，势必会有指令增加、删除及位移，这将影响控制转移指令的跳转。为保持语义，必须修正这些控制转移指令。如果直接处理二进制代码，应不断重新分析计算转移地址，并修改相应控制转移指令。而对于汇编码，处理过程则较简单，由于使用标号，因此不需频繁修改控制转移指令。

(2)可提前消除部分等价指令替换产生的差异。例如相对无条件短跳转指令(机器码为 eb xx)，其跳转偏移范围是[-128, 127]，恶意代码变形时可将其替换为相对无条件长跳转(机器码为 e9 xx xx xx xx)。但经反汇编后，这些指令将用统一的助记符“ jmp Label ”表示。

4.2　等价指令替换产生差异的消除

4.2.1　规范代码指令

x86 汇编指令可分为 2 类：(1)进行基本操作的简单指令，如 mov eax, ebx 和 add edx, ecx 等，这种指令如果存在等价替换，只能被替换为其他单条指令。例如 xor eax, eax 可被替换为 mov eax, 0，也可替换为 sub eax, eax。(2)隐含了多个基本操作的复杂指令，如 movsd 和 call label 等，这种指令可被等价替换为一个指令序列，前面已经举例说明了 movsd。

如果将代码中的复杂指令全部替换为等价简单指令序列，同时对简单指令进行规范化，即对表述相同语义的等价简单指令，从中选择一个作为标准指令，并将代码中非标准简单指令都替换为标准的，例如对寄存器 eax 的清零操作，选择 mov eax, 0 作为标准简单指令，而将代码中的 xor eax, eax 和 sub eax, eax 都替换为 mov eax, 0。这样将在很大程度上消除等价指令替换产生的差异。

为进行上述规范操作，需要建立规范规则集 R，其中每条规则的左部是非标准简单指令或复杂指令，而右部是与左部等价的标准简单指令(序列)。首先给出最小完备汇编指令集的定义。

定义 设 S 是包含 x86 所有汇编指令的集合。对于集合 $B\subset S$，如果 $\forall b\in B$，b 不等价于 B 中其他任何指令(序列)，并且 $\forall s\in S\backslash B$，B 中存在指令(序列)等价于 s，则 B 称为 x86 最小完备汇编指令集。

这里，对于汇编指令操作数中的立即数、标号和变量，不区分其具体值，分别用 imm, label 和 var 表示。另外操作符相同但操作数不同的指令是不相同的，例如 mov eax, imm 和 mov ebx, imm 是 2 条不同汇编指令。

从 B 的定义可看出，它只包含简单指令，而且对各基本操作语义，B 中有且只有一条指令能够表述。因此，可将 B 作为标准简单指令集，并在此基础上构造规范规则集。

构造 B 和 R 的算法描述如下：

(1)构造B。初始设$B= S$，然后从B中随机选择一条指令b，如果$B\backslash\{b\}$中存在一指令(序列)等价于b，则$B=B\backslash\{b\}$，重复此过程，直到不能再从B中删除任何指令。

(2)构造R。初始设$T=S\backslash B$, $R=\varnothing$，从T中随机选择一条指令t, $T=T\backslash\{t\}$，从B中构造与其等价的指令(序列)$<b_1b_2\cdots b_n>$，$R=R\cup\{t\rightarrow <b_1b_2\cdots b_n>\}$，重复此过程，直到$T=\varnothing$。

从 R 的构造算法可看出，规范规则都是一到一或一到多的形式，据此规范代码指令方法为：顺序扫描代码各指令，将其与 R 中各规则左部匹配，如果匹配成功，将扫描到的指令替换为规则右部。

4.2.2　规范指令顺序

规范代码指令在很大程度上消除了等价指令替换产生的差异，但还不能完全保证一致性，这种不一致体现在指令顺序上。例如恶意代码在变形时可能将 movsd 替换为图 3(1)所示指令序列，对其代码指令进行规范不会改变它们的相对顺序，而对初始恶意代码指令进行规范后，movsd 将被扩展为图 3(3)所示指令序列，这 2 个序列虽然有相同的指令，但在顺序上有差异。因此，规范完代码指令后须对指令顺序进行规范。

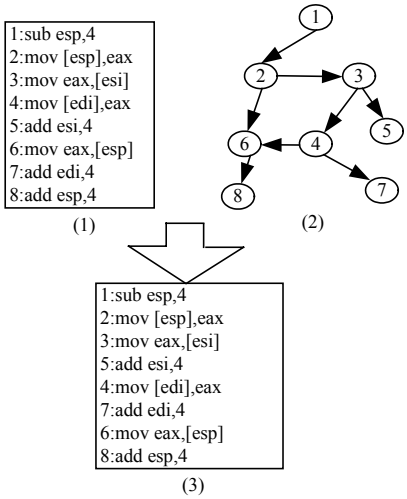

图 3　规范指令顺序示例

由于这种不一致局限于同一基本块中，因此只需在各基本块内进行规范。指令间的数据依赖关系决定了指令相对执行顺序，例如图 3(1)中 mov eax, [esi]必须位于 add esi,4 前，因为它们之间存在反向数据依赖，改变它们之间的顺序将改变原有语义，而 mov [edi], eax 和 add esi,4 则无数据依赖，可交换位置。指令顺序规范算法如下：

(1)分析基本块内数据依赖关系并构造有向图$G=<V,E>$, V中各节点代表块内各指令，E中各有向边$v_i\rightarrow v_j$代表指令i与指令j有数据依赖，且i必须在j前；分配指令缓冲池P。

(2)选取 G 中无入边节点，将对应指令放入 P。

(3)按优先级规则从P中输出指令i，将v_i从G中删除，并删除其所有出边。

(4)查看 G 和 P 是否空，如果空则结束，否则转(2)。

运用上述算法对图 3(1)进行规范后得到图 3(3)，图 3(2)

是图 3(1)中各指令间数据依赖关系图。

上述算法执行时，*P* 中可能同时有多条指令，它们无依赖关系，都可被输出，因此，须定义优先级规则来选择输出的指令，以此保证规范的一致性：

(1)定义操作符优先关系，例如 add 优先于 mov。如果 *P* 中操作符优先级最高的指令只有一条，则选择该指令输出。

(2)如果 *P* 中操作符优先级最高的指令有多条，则分别计算这些指令与其所依赖指令间的距离，选择距离最大的输出。距离以当前输出位置到所依赖的指令操作数之间包含的操作数目计算，例如对图 3，当指令 2 可被输出时，它到指令 1 的距离为 2。如果指令有多个依赖，则距离取平均值。

4.3 垃圾代码的消除

恶意代码变形时插入垃圾代码会使其尺寸变大，执行速度降低，消除这些垃圾代码类似于对其进行优化处理，因此，可借鉴编译技术中的相关优化方法来删除垃圾代码。这些优化方法主要包括：删除无用赋值，常量传播，合并已知量，复写传播，删除无用运算等，下面以 Win32.Evol 的一段含有垃圾指令的变形码为例来说明这些方法。

图 4(1)中 L1 处对 eax 的赋值是无用的，因为 L2 处函数调用返回后将对 eax 重新定值，所以可把 L1 处指令删除。另外经过对 eax 的常量传播，L4 处指令将变为 push 4，同时由于 L5 处的函数调用在返回后将对 eax 重新定值，因此 L3 处指令也将成为无用赋值被删除。L6 至 L9 处的约减稍微有点复杂，首先，经过复写传播，L8 处变为 mov edi, eax，此时由于 L9 处要对 ebx 重新定值，因此 L7 处指令可以先删除，随后，由于在 L6 到 L9 之间再也没有对 ebx 的使用，因此 L6 和 L9 处的指令都可以被删除。删除垃圾代码后的结果如图 4(2)所示。

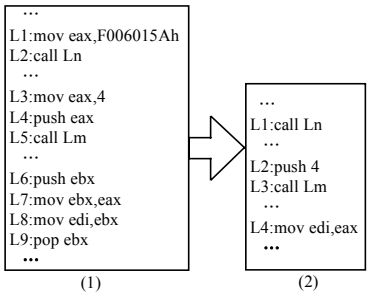

图 4 消除垃圾代码示例

4.4 指令重排的影响的消除

如果构造图 1(4)所示代码段的 CFG，可以发现指令重排使得 CFG 中增加了许多以 jmp 指令为出口语句的基本块，而且这些基本块都是其后继的唯一前驱。可通过合并该类型的相临基本块来规范 CFG 从而消除指令重排的影响，操作过程如下：首先构造代码段的 CFG，然后遍历图中每个节点，如果该节点的出口语句是 jmp 指令，并且该节点是其后继节点的唯一前驱，则将 jmp 指令删除，同时合并这两个节点，最后根据规范化后的 CFG 输出代码。

4.5 归一策略

为提高处理效率，须按一定策略来调用各归一化方法。

(1)由于指令重排增加了 CFG 复杂度，即节点数与边数增多，因此应先规范控制流图以利于后续分析处理。

(2)由于消除垃圾代码需进行数据流分析，而简单指令相对于复杂指令更利于该分析，因此在消除垃圾代码前应先规范代码指令。

(3)由于规范指令顺序，须分析基本块内数据依赖关系，提前消除垃圾代码将提高分析效率，因此规范指令顺序前应先消除垃圾代码。

归一器结构框图如图 5 所示。

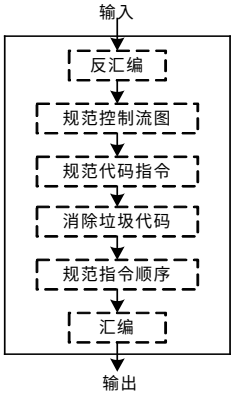

图 5 归一器结构框图

5 测试及讨论

5.1 测试

Win32.Evol 是一种典型的变形病毒，笔者分析提取了其变形引擎(具有等价指令替换和插入垃圾代码功能)，并以此来测试前述归一化模型的归一效果。

随机截取一段连续病毒体代码作为初始代码，送入变形引擎变形获得第 1 代变体，对该变体再进行变形得到第 2 代，依此总共获取 3 代，每一代均采集 5 个样本，另外对每个样本手工进行代码重排，最后将初始代码和各代样本送入归一器，并比较输出结果。为直观比较结果，测试中归一器直接输出归一化后的汇编码。表 1 列出了实验结果(平均值)，其中代码大小以指令数目计算。

表 1 实验结果

	大小	归一化后大小	不同点	处理时间/ms
初始	598	914	—	1 293
1 代	915	914	0	1 921
2 代	1 114	920	6	2 447
3 代	1 354	929	15	3 015

从表 1 可看出，样本代码大小随着代数逐渐增大，这是因为其变形引擎不会缩减代码。归一化后，1 代代码和初始代码的结果一致，而 2 代和 3 代则有轻微差别，这是因为目前原型在消除垃圾代码时还不够完善，另外处理时间也较长，对原型进一步优化可以提高归一化的准确性及处理效率。

5.2 讨论

恶意代码可使用复杂变形技术来增加归一化处理难度，例如变形时先将机器码转化为自定义伪代码，对伪代码进行变换，再将伪代码转化为机器码，对此机制进行归一化须分析伪代码。尽管如此，复杂变形技术同样也增加了恶意代码作者编写自动变形引擎的难度。除归一化方法外，通过定义和匹配抽象特征模式[4]也是检测变形恶意代码的一种思路，该方法同样难以应对复杂变形技术；动态行为检测[5]是另一种检测思路，由于需要模拟器或虚拟机支持，因此恶意代码可以探测自身运行环境以逃避检测。

6 结束语

综上所述，当前各种检测变形恶意代码的方法都不够完善，在现实应用中，单独依靠某种检测技术很难达到较好的检测效果，因此可以通过综合运用各种检测手段来克服它们各自的缺点，这是下一步的研究方向。 （下转第 190 页）