单位代码：10293 密　级：

南京郵電大學

硕 士 学 位 论 文

论文题目：基于启发式算法的恶意代码检测系统研究与实现

学　　号　Y004091137

姓　　名　雷迟骏

导　　师　王海艳 教授　王汝传 教授/博导

学 科 专 业　计算机软件与理论

研 究 方 向　软件技术及其在通信中的应用

申请学位类别　工学硕士

论文提交日期　二零一二年三月

南京邮电大学学位论文原创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京邮电大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

本人学位论文及涉及相关资料若有不实，愿意承担一切相关的法律责任。

研究生签名：_____ 日期：_____

南京邮电大学学位论文使用授权声明

本人授权南京邮电大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档；允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索；可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。本文电子文档的内容和纸质论文的内容相一致。论文的公布（包括刊登）授权南京邮电大学研究生院（筹）办理。

涉密学位论文在解密后适用本授权书。

研究生签名：_____ 导师签名：_____ 日期：_____

南京邮电大学

硕士学位论文摘要

学科、专业：工科、计算机软件与理论

研究方向：基于网络的计算机软件应用技术

作者：二零零九级 硕士研究生 雷迟骏 **指导教师**：王海艳 教授 王汝传教授/博导

题目：基于启发式算法的恶意代码检测系统研究与实现

英文题目：The Research and Realization of Unwanted Code monitoring system based on Heuristic algorithm

关键词：恶意代码，检测，启发式算法，模式匹配，虚拟技术

英文关键词：Pattern matching, Heuristic algorithm, Virtual machine technology, Cloud computing, Unwanted Code

论文选题来源：

1. 国家863计划项目“新型对等计算安全软件平台的设计与实现”（编号2006AA01Z439）
2. 华为赛门铁克公司企业项目“ARES协议分析”（编号YBSS2009401）

中文摘要

随着计算机技术的发展，尤其是计算机网络的发展，恶意代码也在不断的发展中。当前的恶意代码的数量比起过去有着呈几何增长。传统的恶意代码基本都以病毒形式出现，而当前恶意代码形式多种多样,如蠕虫、病毒、木马、恶意插件等。从功能上分析，传统的恶意代码一般功能单一，大多以数据破坏为主，而当前的恶意代码还具有数据窃取、篡改等功能，并且运用了大量的反调试、反跟踪、反检测等技术来保护自身。由此可见，当前恶意代码对计算机发展的危害已经越来越显著，同时也是检测越来越困难。

在检测恶意代码方面，传统的杀毒软件仅仅采用的是二进制特征代码匹配技术来进行检测。由于该方法必须要得到恶意代码的二进制特征代码，一旦恶意代码通过加密等方式改变了其二进制特征代码，该方法将彻底失效。传统的杀毒软件面对当前的恶意代码已经显得力不从心。

本论文在阐述模式匹配算法、启发式扫描算法以及虚拟机技术等的基本理论及关键技术的基础上，研究了目前恶意代码检测引擎的模式匹配算法的不足，提出改进的方案。此外在基于启发式扫描算法和虚拟机技术的特征行为引擎研究下，本论文建立了一个结合二进制特征匹配、行为特征匹配以及云端检测的新型恶意代码检测方法的原型系统。

本文的研究成果能有效提高系统资源，并且充分利用网络资源，抵御恶意代码入侵,对维护互联网的健康环境，进而营造出一个和谐的网络社会有着积极意义。

关键词： 模式匹配，启发式，虚拟机技术，云计算，恶意代码

ABSTRACT

Nowadays, with the development of computer technology, especially computer networks, malicious code are constantly developing. The number of current malicious code has exponential growth than in the past. The traditional malicious code are basically in the form of the virus, but the current malicious code are all kinds of forms, such as worms, viruses, Trojan horses, malicious plug-ins. From the functional analysis, the traditional malicious code generally functioned specially on data breaches, but the current malicious code function on data theft, tampering and other functions, and also use of a large number of anti-debugging, anti-tracking , anti-detection techniques to the protect themselves. Evidently, the development of malicious code on computers has become increasingly significant harm and makes it more and more difficult to detecting them.

To detect the malicious code, traditional anti-virus software only uses the method of binary characteristics of the code-matching techniques. Since the method must get the binary characteristics code of the malicious code, once the malicious code changes the characteristics code by encrypting its binary code, the method will completely fail. The traditional anti-virus software seems powerless to the current malicious code.

Based on the very explanation of basic theory and key technology, such as pattern matching algorithm, heuristic scanning algorithms and virtual machine technology, the paper has researched deeply on the shortages of pattern matching algorithm of the current malicious code detection engine, and put forward the improved method. In addition, based on the research of heuristic algorithms and the engine using virtual machine technology to match the behavior characteristics, the paper set up a prototype system which combined the binary feature matching, feature matching and the method of behavioral detection of cloud.

This research can improve the system resources and make full use of network resources to defend against malicious code intrusion, maintain a healthy environment of the Internet, thus and make positive sense to creating a harmonious network social.

Key words: Pattern matching, Heuristic, Virtual machine technology, Cloud computing,

Unwanted Code

目 录

第一章 引 言

1.1 研究背景

恶意代码是指没有作用却会带来危险的代码，一个最安全的定义是所有不必要的代码都看作是恶意的。近些年随着计算机技术的发展，尤其是计算机网络的发展，编程技术已经慢慢的普及化。编写病毒之类的恶意代码已经不再是计算机高手的专利，随之而来的是恶意代码的数量呈指数级增长，对社会的危害越来越大[1]。另一方面，恶意代码作者由于经济利益的驱使，在恶意代码编写的过程中利用了更多高深的技术来达到保护自身的作用，rootkit 技术的迅速发展就是一个很好的证明[2]。

传统的恶意代码检测系统主要采用的是二进制特征代码匹配技术，首先病毒分析专家通过分析确定一些已知恶意代码的二进制特征代码，保存到数据库即病毒库中，在检测中利用模式匹配算法对带检测的代码与病毒库中的特征代码进行匹配，匹配成功确定其为恶意代码，否则视为安全代码。

但当前随着恶意代码的数量的增多，病毒库越来越庞大，利用传统的模式匹配算法说耗费的时间不断的增大，已经不能满足实际应用。另外随着恶意代码编写技术的发展，当前的恶意代码大多采用加密等技术对自身进行保护，这样就更近一步降低了传统检测方法的准确性。为了解决这一问题，最有有效的应为归纳总结出一类恶意代码的行为特征，将传统的二进制特征匹配提升到行为特征匹配，才能利用尽可能少的特征检测出尽可能多的恶意代码。除此之外，充分的利用网络的资源共享优势，利用云计算技术进行辅助检测，从而有效的降低了误报率。

1.2 课题来源及本人所做工作

本文所涉及的课题得到国家 863 计划项目（编号 2006AA01Z439）和华为赛门铁克公司委托开发项目的资助。

本人所做的工作可陈述如下：

(1)探讨目前诸多模式匹配算法，并通过比较，突出恶意代码检测系统的特点，分析了目前诸多算法的不足，提出了改进算法。

(2)分析了虚拟机技术，全面的分析了目前几种流行的虚拟机技术，经过比对得出了更适合于恶意代码检测引擎的。

(3)学习启发式算法，探讨了启发式技术在恶意代码中的实际运用，给出了原型系统的总体设计框架。

(4)探讨云计算技术，提出了结合本地防御，网络资源共享的整体系统架构。

(5)设计原型系统，并在现有条件下对系统进行测试。

1.3 论文结构

论文共分为七章，其组织结构如下：

第一章 引言。首先介绍了恶意代码检测技术的历史背景和发展现状，并根据目前恶意代码发展的趋势特点指出了它的广阔应用前景，以及主要的研究机构等。最后指出本课题的来源、本人所做工作以及论文采取的结构。

第二章 模式匹配算法概述。本章介绍了包括单模式匹配算法 KMP 和多模式匹配。然后结合当前恶意代码检测系统的特点，分析了 AC 与 WM 两种多模式匹配的不足。之后提出了改进算法 AW 算法，并通过实验得出数据进行对比，证明 AW 算法可以支持当前环境下恶意代码特征码检测引擎。

第三章 虚拟机技术概述。本章全面介绍了虚拟机技术。根据系统架构的不同，对当前虚拟机产品进行了分类。详细研究了目前流行的 x86 PC 系统的体系结构，进而研究了虚拟机工作的原理。 随后对基于指令模拟的两款虚拟机产品 BOCHS 和 QEMU 进行了对比分析，分析了静态翻译和动态翻译的差异。着重研究了动态翻译，为整个恶意代码监测系统中系统仿真打下基础。

第四章 基于语义的启发式检测模型。本章全面介绍了基于语义的启发式恶意代码检测技术。首先阐述了根据当前恶意代码发展的趋势，利用行为特征分析取代传统二进制特征码定位的优势。介绍了静态启发式扫描和动态启发式扫描两种扫描方式。此外，提出了基于语义的可信模型和信息熵判断模型。

第五章 高性能云服务器设计。本章全面介绍了高性能服务器的构建，为监测系统的云服务平台提供支持。着重研究了windows平台下网络应用程序的六种socket I/O模型。此后，详细研究了windows完成端口网络模型，分析了其整个工作流程。

第六章 原型系统实现与测试。本章主要完成对检测原型系统的设计实现和测试。在根据前面研究的模式匹配、虚拟机技术、启发式检测技术和高性能服务器技术的基础上，设计了高性能、稳定、可靠的基于云计算技术的恶意代码检测系统。之后对系统的查杀率和误报率的两方面问题进行了测试，并与国内外的一些著名杀毒软件测试的结果进行了对比，基本达到预期效果。

第七章 总结和展望。对本课题实现的恶意代码检测系统作出总结，分析了其实现方法及应用前景。并针对当前系统中尚存在有待改善的地方，以及对如何开展进一步的工作做出展望。

第二章 模式匹配算法概述

模式是指按照某种结构组织起来的多个元素的集合[3]。模式匹配是指将两个模式作为输入，计算模式元素之间语义上的对应关系的过程。模式匹配作为计算机科学中的一个重要研究领略，被广泛应用于搜索引擎、海量数据处理、网络入侵检测等领域[3]。尤其是目前的杀毒软件的检测引擎重要是基于模式匹配[4]。

2.1 模式匹配算法研究现状

首先出现的模式匹配算法为BF算法，这是一种单模式匹配算法，由于采用的只是简单的逐个字符比较，因而效率很低。1977年Boyer和Moore提出的单模式BM算法，由于BM算法主要是针对当字符串比模式的长度大的情况，造成了在移动对于小的字符串情况下不是特别有效。此后Horspool根据BM算法的思想提出通过最右边的字符的移动来计算BM算法的移动距离，即BMH算法。在诸多单模式匹配算法中，最为著名的要属Knuth，MorriS和Pratt提出了KMP算法。该算法和前面算法不同，KMP算法在比较时，如果某次匹配不成功，模式串可能右移多个字符，而且右移后，可以不必从模式起点处进行匹配[5]。

随着计算机的发展，简单的但模式匹配算法已经不能满足实际需求。随着模式数量的增加，采用单模式匹配需要对每个模式进行重复匹配，检测算法的时间复杂度将呈指数级增长。为了提高检测效率，需要一种高效算法能够完成一次对多个模式进行匹配操作，即对 N 个模式进行匹配时，只需要进行扫描一次数据流操作即多模式匹配算法。1975 年，Aho 和 Corasick 提出了著名的基于有限状态自动机的 AC 算法，这种算法最早应用于图书馆的书目查询中，它采用一种有限自动机把所有的模式串构成一个集合，可以一次查找多个模式。但由于 AC 算法空间复杂度过大，在模式集合过大时，实际效果并不理想。上世纪九十年代吴升（台湾）和他的导师 Udi Manber 提出了 Wu-Manber 算法。

2.2 模式匹配算法介绍

模式匹配算法从待匹配集合的数量上分类，可分为单模式匹配算法和多模式匹配算法。单模式匹配算法以 KMP 为代表[6]，多模式匹配算法从设计思路上分类大致有 AC 算法和

WM 算法，其他诸多算法大多是基于这两种的改进。

2.2.1 KMP

设有两个串 S 和 P，其中 S 为主串，P 为子串，又称为模式。在简单模式匹配中，一旦出现了匹配失败的情况，算法将回溯到模式的原点进行重新匹配，而 KMP 引入了失败函数，避免了匹配过程中的回溯，从而有效的提高了匹配效率。如图 2-1 所示，在 S[i]≠P[j]即到达匹配失败时。在简单模式匹配算法中，主串 S 要回到原来的 i+1 的位置（称为回溯），模式串 P 要回到第一个字符的位置，然后继续匹配如图 2-2 所示。每次到达匹配失败点时，串 S 和串 P 的指针 i ，j 都要回溯，因而效率不高。而在 KMP 算法中在第一趟到达匹配失败点，即 S[4]≠P[4]时，匹配失败点前的 4 个字符是两两相等的。由于 S[1]=P[1](='b')，而 P[0]≠P[1]，因此 P[0]≠S[1]，这样 i 回溯到 1 和 j 回溯到 0 的第二趟匹配是毫无意义的；同理，第 3 趟也是没有意义的。而第 4 趟匹配，i 可以不回溯,j 只要回溯到 1，如图 2-3 所示。

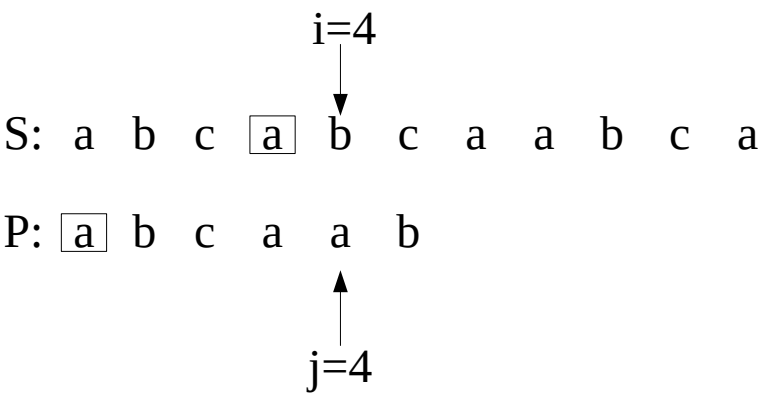

图 2-1 到达匹配失败点

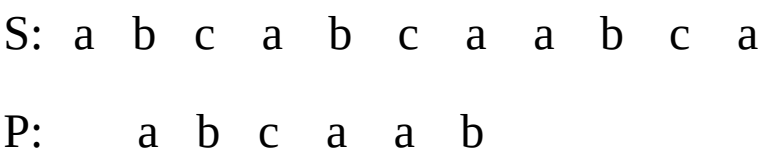

图 2-2 简单匹配算法的下一次匹配

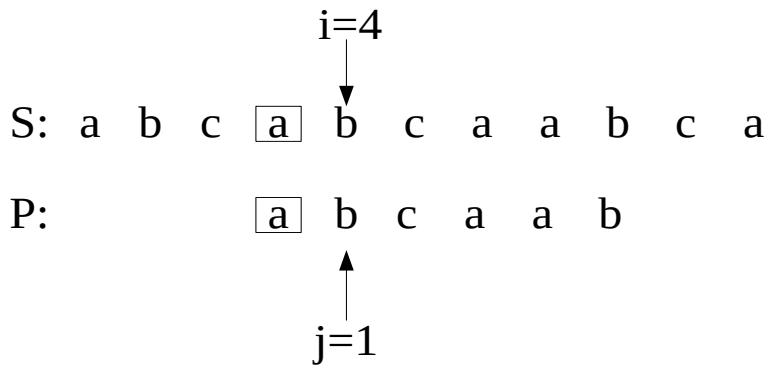

图 2-3 KMP 算法的下一次匹配

又上可见，问题的核心在于 j 回到什么位置。因而 KMP 算法的关键是求出模式串 P 的所有字符的最大 k 值，k 是匹配失败是 j 需要向前回溯的最少位置。一旦在该字符上出现了匹配失败时，下一趟匹配可以直接从 S[i]和和 P[k]开始。设主串 S="$s_0 s_1 ... s_{n-1}$"，模式串 P="$p_0 p_1 ... p_n$"，并设在 $s_i \neq p_j$ 处匹配失败。分析 P 串，如果发现 $p_0 p_1 ... p_{k-1} = p_{j-k} p_{j-k+1} ... p_{j-1}$，我们称在串 $p_0 p_1 ... p_{j-1}$ 中，串 $p_0 p_1 ... p_{k-1}$ 为前缀子串，串 $p_{j-k} p_{j-k+1} ... p_{j-1}$ 为后缀子串。由于匹配在 $s_i \neq p_j$ 处失败，所有必有 $p_{j-k} p_{j-k+1} ... p_{j-1} = s_{i-k} ... s_{i-2} s_{i-1}$ 和 $p_0 p_1 ... p_{k-1} = s_{i-k} ... s_{i-2} s_{i-1}$。所以下一趟匹配可以直接从 s_i 和 p_k 开始，即将模式串中 k 位置的字符与主串中 i 位置的字符对齐后开始下一趟匹配。其中匹配算法中的重要代码为

```
int    FindKMP(int I,String &P)
{
      if(i<0 || i>n-1)
      {
            cout << "Out of bounds!" << endl;
            return -1;
      }
      int j = 0,m=P.n;
      while(i<n&&j<m)
      {
            if(j == -1||str[j] == P.str[j])
            {
                  i++;j++;
```

```
            }
            else{
                j = P.f[j];
            }
        }
        return ((j == m)?i-m:-1);
}
```

失败函数代码

```
void    Fail()
{
        int j = 0, k = -1;
        f[0] = -1;
        while(j < n)
        {
            jf((k == -1) || (str[j] == str[k]))
            {
                j++;k++;
                f[j] = k;
            }
            else{
                k = f[k];
            }
        }
}
```

由上述代码可知，假设模式串的长度为 m，则失败函数的时间复杂度为 O(m)。如果将失败函数的计算时间包括在 KMP 算法的匹配过程中，则 KMP 算法中，当主串的长度为 n 时，算法的时间复杂度为 O(m+n)。

2.2.2 Aho-Corasick 算法

Aho-Corasick 自动机算法（简称 AC 自动机）于 1975 年产生于贝尔实验室，是用来处理模式集合大于 1 的多模式匹配算法。该算法和巧妙的运用了有限状态机机制，将字符匹配转化成了状态转移[7]。

该算法的基本思想为：

AC 算法大致思想为，根据待匹配的多模式串建立一个有限状态机，即一个树形数据结构，将主串作为输入，放置到状态机中进行状态转换，当到达某些特定的状态时，模式匹配成功[8]。

在预处理阶段，算法建立了三个函数，跳转函数 goto，失败函数 failure 和输出函数 output，由此构造了一个有限自动机。

在搜索查找阶段，则通过这三个函数的交叉使用扫描主串 S，定位出模式串 P_i 在主串 S 出现位置。

此算法有两个极其显著的特点，一个是扫描主模式串 S 时完全不需要回溯，另一个是不需要对模式集中的模式进行逐个匹配，时间复杂度为 O(n)。

算法步骤如下：

(1) 根据模式串 P_i 建立有限状态机。

设多模式集合 P_i ={“she”, “he”, “his”, “hers”}，如图 2-4 为对应的有限状态机。

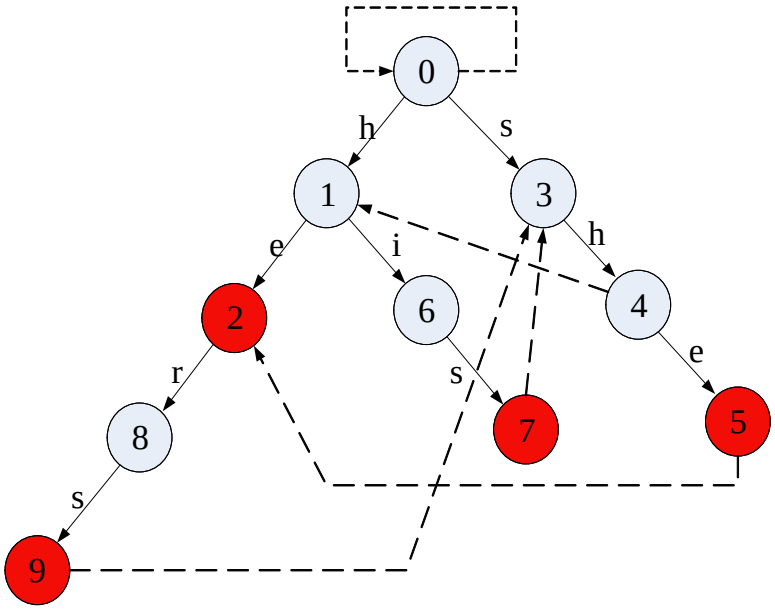

图 2-4 有限状态机图

在该阶段，AC 算法建立了三个函数，失败函 failure、跳转函数 goto 和输出函数 output。

跳转函数 goto，用来确定有限状态机中个个状态的跳转关系。goto(Previous, x)=next, 状态 Previous 在输入一个字符 x 后转换为状态 next。如果在模式串中不存在这样的转换，这说明当前匹配失败，则 next=fail。如图 2-4，当状态机当前处于状态 0，输入字符为 h 时，则进入状态 1，输入字符为 s 时，则进入状态 3.

失败函数 failure，用来判断状态转换的成功与否。在构造转向函数时，用 fail 表示不存在的转换，一旦出现 fail 而整个匹配并没有完全结束时，根据失败函数状态机会跳转到相应的状态，从而使匹配继续。如图 2-4，当状态机当前处于状态 6 时，说明当前的匹配过程已经成功的匹配出了串“hi”，而下一个输入值不为‘s’则说明整个匹配失败，此时状态机回到初始状态状态 0。

(2) 输入主串 S 进行匹配

设主串 S=“ushers”，状态机初始状态为 0，当输入值为‘u’时，匹配失败，状态机回到初始状态 0。输入值为‘s’时，状态机跳转到状态 3。输入值为‘h’时，状态机跳转到状态 4。输入值为‘e’时，状态机跳转到状态 5，匹配成功输出“she”。

由上可见，AC 算法避免了多模式匹配过程中，模式串集合需要多次和模式主串匹配的情况，有效的降低了匹配所耗费的时间，模式主串 S 的长度为 n 时，算法的时间复杂度为 O(n)。

2.2.3 WM 算法

WM 算法在多模式匹配中，引入了坏字符跳转的概念，从而实现了模式匹配中的大幅度跳转。该思想源自于 BM 单模式匹配算法，但是在单模式应用场景，很少有哪个模式串会包含所有可能的输入字符，而在多模式匹配场景，如果模式集合的规模较大的话，很可能会覆盖很大一部分输入字符，导致坏字符跳转没有用武之地。所以 WM 算法中使用的坏字符跳转，不是指一个字符，而是一个字符块，或者几个连续的字符。通过字符块的引入，扩大了字符范围，使得实现坏字符跳转的可能性大大增加。WM 算法与 AC 算法一样分为预处理阶段和搜索阶段[9]。

预处理阶段需预先对模式串集合进行处理，并构造三个表：SHIFT 表、HASH表、PREFIX 表。WM 算法中SHIFT 表主要是用来确定匹配串滑动窗口可以跳跃多少个字符位置。WM 算法将文本串以B 个字符长度分块，称该B 个字符为1 个块字符，B 为块字符的长度，B 通常取2 或3。当匹配串滑动窗口的末字符块与模式串集合中的任何一个模式串的末字符块不匹配时，文本滑动窗口可以安全地移动m-B+1 个字符位置（其中B 是

字符块的长度，m是模式串集合中长度最小的模式串的长度）。HASH 表是建立在模式串的前缀基础上(前缀就是滑动窗口中对应字符串的首字符块)，HASH 函数把每个模式串的长度相等的前缀映射成一个合适的值，同时有较好的冲突处理方法。HASH表用来确定某个模式串是否是一个待匹配的模式串。

算法搜索过程的步骤如下[10]：

(1) 计算所有待匹配模式串的长度，得到其中的最小值，记为m,这里将 m 称为匹配窗口的大小；

(2) 根据所有待匹配模式串的 m 个字符，计算其尾块字符T[m-B+1,…, m]的散列值h；

(3) 检查 SHIFT[h]的值,如果SHIFT[h]>0，将窗口向右移动SHIFT[h]大小位置，返回第(2)步，否则，进入第(4)步；

(4) 计算所有待匹配模式串对应窗口“前缀”的散列值，记为 text_prefix；

(5) 对 符 合 HASH[h] ≤ P<HASH[h+1] 的每一个p 值， 检验是否存在 PREFIX[p]=text_prefix.如果相等，对文本和模式串进行完全匹配；

搜索过程的主要代码如下：

```
while(text <textend)
{
    hashVal = hashBlock(text);
    shift_distance = SHIFT[hashval];
    if(shift_distance == 0)
    {
        shift_distance = 1;
        p = HASH[hashval];
    }
    text += shift_distance;
}
```

当字符表的长度为L，模式串长度总和为M，主串长度为B，窗口大小为m，WM 算法在预处理阶段的时间复杂度为O(L+M)，匹配阶段的时间复杂度为O(B*n/m)。

2.3 多模式匹配算法改进

2.3.1 AC 与 WM 算法比较

由 2-2 节介绍可见，AC 算法的核心在于利用有限状态机将模式集 P 连成整体，然后将主串 S 放入状态机进行匹配，由于不需要对模式集合的模式进行逐个匹配因而大幅缩短匹配时间。WM 算法的核心在于，利用字符块增大匹配失败概率，来避免逐个模式匹配带来的大量无效匹配[11]。

根据以上研究，通过具体实验对AC和WM算法进行对比分析。利用一个随机生成的200M的文件充当主串S，随机生成若干模式组成模式集P，作为测试数据集。分别采用AC和WM算法作为模式匹配检测引擎，测试这两种情况下每个数据报的平均模式匹配时间与模式数目的变化关系。实验环境：CPU 1.66 GHz*2，内存1 GB，操作系统windows xp。表2-1比较了两种算法的平均匹配时间随着模式数目增加的变化情况

表 2- 1 AC 和 WM 算法效率对比

模式数量（个）	AC/us	WM/us
10	0.4134	0.4334
15	0.5432	0.6321
20	0.6340	0.6732
30	0.8324	0.9483
40	1.0349	1.0949
50	1.1325	1.1439
100	1.7023	1.7849
200	1.9899	1.9689
500	3.2090	2.9902
1000	4.3980	3.9943
2000	---------	5.8930
5000	---------	7.8898
10000	---------	9.8090
20000	---------	13.7932
50000	---------	---------

当主串长度为 B，窗口大小为 m 时，由于|B|>m, WM 算法的时间复杂度 O(|B|*n/m)大于

AC 算法的时间复杂度。如表 2-1，在模式数量小于 100 时，AC 算法的效率明显优于 WM 算法。但当模式数量大于 100 时，由于建立有限状态机所需的内存不断增大，AC 匹配过程的状态跳转带来系统内存频繁切换造成在实际匹配过程中，效率反而比 WM 算法低。但模式数量达到 2000 时，由于系统内存的限制，已经无法成功建立有限状态机，而造成 AC 算法就此失效[12]。同样，当模式数量到达 50000 时，WM 算法也因系统内存限制问题失效。

2.3.2 AM 算法

有 2.3.1 节可知，AC 算法在理论上优于 WM 算法，但 AC 算法对系统硬件配置有较高要求。因而造成随着模式数量的增加，AC 算法在实际匹配中效率低于 WM 算法，甚至在模式过多的情况下 AC 算法会失效。而在当前杀毒软件与恶意代码的对抗中，病毒库（即模式集合 P）不断的增大，数量甚至超过几十万，达到百万级别。显而易见，AC 算法和 WM 算法都已经没有办法满足当前恶意代码检测引擎的要求。

根据 AC 算法和 WM 算法的优缺点，我们结合两者对多模式匹配算法进行了改进，提出了 AW 算法。

引入 WM 算法字符块概念，来产生大量失败匹配，从而避免无效匹配。同时引入 AC 算法中状态机原理，避免对模式集合 P 中的模式逐个匹配现象。

假设模式集 P={ $p_0 p_1 ... p_n$ }，函数 f(P_i)为 P_i 模式的串长度，b =min(f(P_i))，(i=0，1，…，n)，我们以长度 b 对模式集中所有模式进行分割，其中长度不足 b 的分割块用十六进制 0x00 填充。例如，P={“qwqew”, “wi”, “ncxk”}，b = 2，其中 P_0 被分割为{‘q’, ‘w’}、{‘q’, ‘e’}和{‘w’, 0x00}， P_1 被分割为{‘w’, ‘i’}， P_2 被分割为{‘n’, ‘c’}和{‘x’, ‘k’}。同时，我们引入 WM 算法的 HASH 函数，对所有分割出来的块进行 HASH 运算得到集合 Q，其中 HASH 算法使用最为简单的异或运算。原因有二，计算机对异或运算处理最快，异或运算能保证使用 0x00 填充的块与原串等效。根据 Q 集建立有限状态机，对主串 S 中的字符串进行逐个 HASH 运算得到输入值，放入状态机进行匹配。

假设主串为 S=“qwiqewdskjskdsjkds”， P={“qwqew”, “wi”, “ncxk”}，我们根据 P 建立得到集合 Q，如表 2-1 所示：

P	{‘q’, ‘w’}	{‘q’, ‘e’}	{‘w’,0x00}	{‘w’, ‘i’}	{‘n’, ‘c’}	{‘x’, ‘k’}
Q	0x32	0x38	0x80	0x83	0x56	0x69

根据 Q 建立有限状态机如图 2-5 所示：

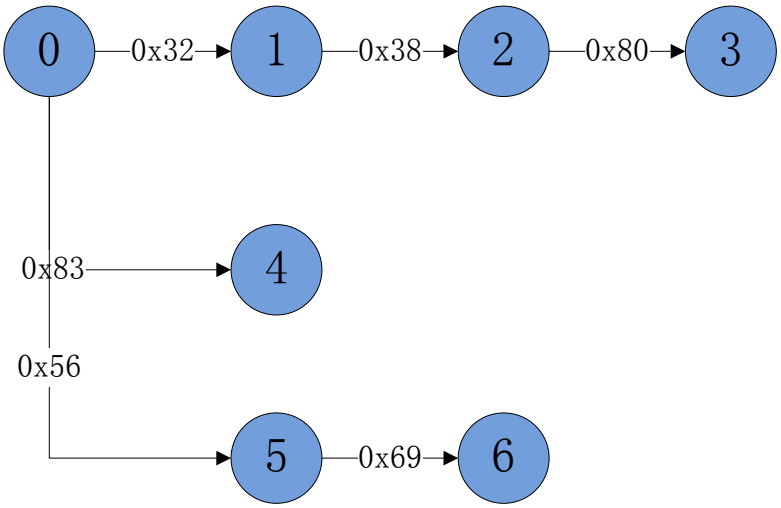

图 2-5 AW 匹配算法状态转换

在AC算法中，模式集合P中一个Bit位将会被转换成一个状态，假设模式集合P中模式的模式总长度为n，则空间复杂度为O(n^2)。在实验对比中，当模式个数为1000，平均模式长度为8时，算法消耗的系统内存达到700M。当模式个数为2000时，预计消耗系统内存为2400M，而实验机器的最大内存为1G，小于2400M，所以算法初始有限状态机失败，匹配无法进行。AW算法采用HASH技术，将AC算法中的几个状态合并成一个状态，从而大大减小算法实际消耗系统的内存量。

根据以上研究，通过具体实验对AC、WM和AW算法进行对比分析。利用一个随机生成的200M的文件充当主串S，随机生成若干模式组成模式集P，作为测试数据集。分别采用AC、WM和AW算法作为模式匹配检测引擎，测试这三种情况下每个数据报的平均模式匹配时间与模式数目的变化关系。实验环境：CPU 1.66 GHz*2，内存1 GB，操作系统windows xp。表2-2比较了三种算法的平均匹配时间随着模式数目增加的变化情况

表 2- 2 AC、WM 和 AW 算法效率对比

模式数量（个）	AC/us	WM/us	AW/us
10	0.4134	0.4334	0.4401
15	0.5432	0.6321	0.6398
20	0.6340	0.6732	0.6804
30	0.8324	0.9483	0.9532
40	1.0349	1.0949	1.1709
50	1.1325	1.1439	1.2201
100	1.7023	1.7849	1.9043
200	1.9899	1.9689	2.0032

500	3.2090	2.9902	3.1023
1000	4.3980	3.9943	4.0921
2000	---------	5.8930	7.0031
5000	---------	7.8898	7.9994
10000	---------	9.8090	9.9790
20000	---------	13.7932	13.8201
50000	---------	---------	19.8943
100000	---------	---------	26.0348
1000000	---------	---------	37.8205

由实验数据可见，AC 算法支持的最大模式数为 1000-2000，WM 算法支持的最大模式数为 20000-30000。AC 算法可以支持百万级别的模式匹配，在具体实验中峰值可以达到 300 万。

2.4 本章小结

本章首先对模式匹配算法进行了简单的介绍，包括单模式匹配算法 KMP 和多模式匹配。然后结合当前恶意代码检测系统的特点，分析了 AC 与 WM 两种多模式匹配的不足。之后提出了改进算法 AW 算法，并通过实验得出数据进行对比，证明 AW 算法可以支持当前环境下恶意代码特征码检测引擎。

第三章 虚拟机技术概述

3.1 虚拟机技术简介

虚拟机技术是一项一直以来都受着高度重视的技术，通过软件模拟或是部分硬件支持来实现一个具有完整硬件系统功能的系统，让在其上运行的软件完成隔离于物理系统，这是虚拟机技术所需实现的目标。其中最为通俗的用途为，在一台安装了虚拟机软件的机器上可以同时运行不同的操作系统，让用户尽可能地充分利用昂贵的大型机资源。典型代表：IBM 7044 计算机[13]。同阶段，还流行另一种虚拟化技术仿真处理器，也称为 p-code 机，该模型后来被广为人知的 VB、Java、.net、prel 等利用。随着计算机技术的发展，尤其是云计算技术的提出，研究人员越来越多的关注虚拟机技术，虚拟机技术也在不断的发展中。为了满足不同的功能需求，出现了不同的虚拟化解决方案，由于其采用不同的实现方式和抽象层次，使得这些虚拟化系统呈现不同的特征。虚拟化技术的分层完全同步计算机系统的分层机构。

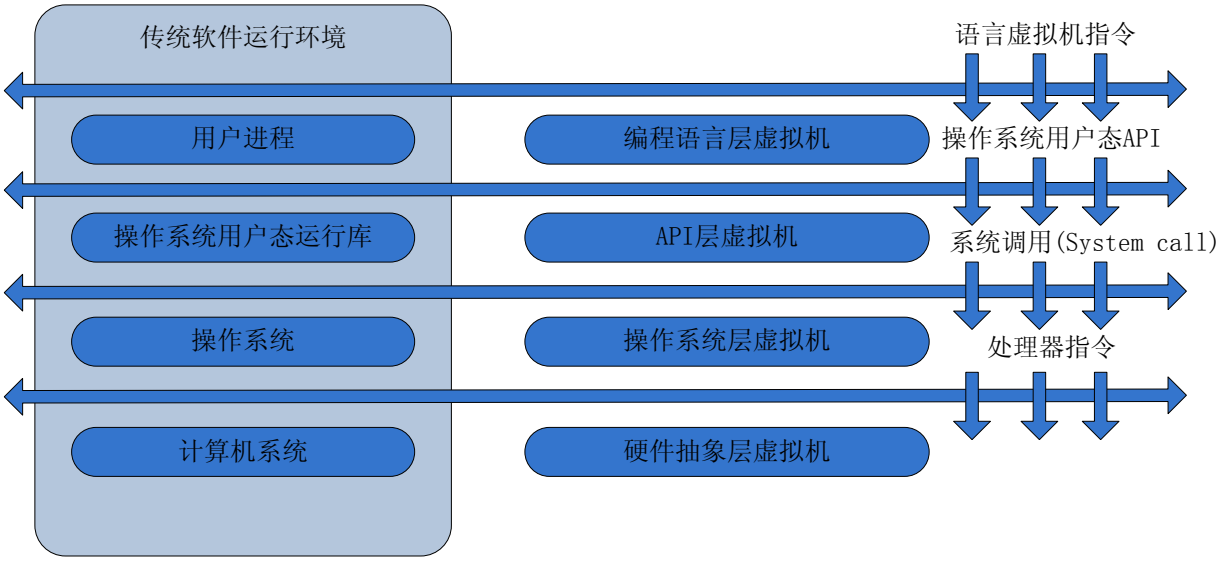

图 3-1 层次化虚拟机分类

如图 3-1，根据计算机系统对应分类，虚拟机可以分为硬件抽象层虚拟机、操作系统层虚拟机、API 层虚拟机和编程语言层虚拟机。从实际虚拟机产品上分类，稍微有点区别，大致为以下分类：

（1）指令集虚拟化

指令集虚拟化通过纯软件方法，模拟出与实际运行的应用程序（或操作系统）所不同的指令集去执行，采用这种方法构建的虚拟机一般被称为模拟器。模拟器通过将客户虚拟机发出的所有指令翻译为本地指令集，然后再真实硬件上执行。当然，一个模拟器要成功的模拟一个真实的机器，它必须能够模拟真实及其所能做的一切事情，包括读 ROM，重启，系统关闭等。典型代表： BOCHS / Crusoe / QEMU / BIRD

（2）硬件虚拟化

硬件虚拟化，实际和指令集虚拟化非常相似，但不同之处在于，这种虚拟化所考虑的是一种特殊情况：客户执行环境和主机执行环境具有相同的指令集，并充分利用这一特点，让绝大多数的客户指令在主机上直接执行，从而大大提高执行的速度。这种技术要正确工作，所构建的虚拟机必须对其中的一些特权指令（如修改页表等操作）进行处理，执行时候产生陷入并把它传递给下层的虚拟机监视器(VMM)执行。典型代表： VMware / VirtualPC / VirtualBox / Denali / Xen / KVM / UML /CoLinux[14][15]

（3）操作系统虚拟化

操作系统虚拟化，其特点是采用在 OS 和用户插入一层，为用户提供多个独立，隔离的应用环境。典型代表： Jail / Ensim[16]

（4）编程语言虚拟化

基于 p-code 原理典型代表: Java / .Net CLI

3.2 x86 体系结构与 PC 系统概要

3.2.1 x86 CPU 结构

（1） 冯诺依曼架构和 CISC 指令集

当前的计算机系统基本都遵循冯·诺伊曼结构。冯·诺伊曼结构，是一种将程序指令存储器和数据存储器合并在一起的存储器结构。程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，因此程序指令和数据的宽度相同，Intel 公司x86 就是使用了这个结构[17]。

从指令集划分，计算机可以分为复杂指令集计算机（CISC）和精简指令集计算机（RISC）。

从命名上就可以看出，CISC包含了一套复杂的指令集。往往一条指令就可以完成一套复杂的操作，而在RISC中，这些操作需要几条指令才能完成。CISC和RISC各有利弊，一直称为计算机结构讨论的热门话题，CISC功能强大但译码器复杂，RISC简洁但需要更多的指令完成相同的操作。最终结论是CISC和RISC应该互相借鉴优点，而不应该在划分界线上争论，事实上处理器的发展也正是沿这个趋势进行的[18]。

（2） CPU 结构

从物理结构上看，x86 CPU 结构比较复杂的，概括的说包括运算器，控制器以及辅助设备如MMU，FPU 以及高速缓存等，这也是一个处理器的基本结构，如图3-2所示。其核心单元是取指-译码-执行单元，所有的IA-32 指令都在这个单元内执行；控制单元起调度作用，控制着执行单元的节拍；外部中断信号可以使执行单元发生转移，处理中断；FPU 辅助执行单元处理浮点指令，MMU 实现虚拟地址，另外，为了提高执行效率，CPU 增加了指令高速缓存和数据高速缓存来减少总线访问的次数，毕竟总线访问会占用大量的时间。

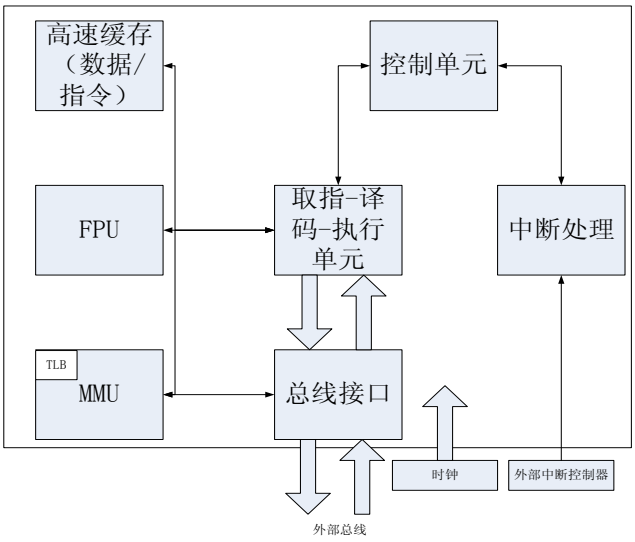

图3-2 CPU结构框图

实际的 CPU 还运用了很多加速技术，如流水线、乱序执行等。如果从逻辑层面上去描述，所有的CPU 操作都是围绕寄存器进行的。X86-32 的主要寄存器包括一组通用寄存器、段寄存器、控制寄存器，CPU 状态FFLAGS 以及程序指针PC。虚拟机的实现的主要工作就是维护这些寄存器的值，即通过IA-32 的指令去修改和更新它们。

3.2.1.3 CPU 工作模式

x86 CPU 共有三种不同的工作模式加上一个准模式，即实模式、保护模式、系统管理模式和虚拟86 模式。实模式是 x86 CPU 复位后首先进入的模式，实模式下的32bit 处理

器按照16bit 8086 处理器的方式执行指令，通过段寄存器左移4 位加上偏移量进行寻址。

在保护模式下段地址寄存器中内容的不再象实模式那样是段的基地址，而只是描述符表中的一个索引，段的真正信息（基地址、限长、访问权限等）放在描述符表中，当访问一数据时CPU 会从描述符表取出段的描述信息来检查访问是否合法，不合法就产生异常，合法则允许访问。每次访问都要读出描述符信息再检查是一个比较费时的过程，为了提高内存访问的速度，Intel 公司在CPU 中为每个段寄存器配备了一个高速缓冲器来存放段的描述符信息，这样访问内存时就不用频繁地访问描述表，只要从高速缓冲进行校验就行，只有在改变段寄存器的值时才访问描述符表将新的段描述符装入高速缓存中。

3.2.2 x86 体系结构概览

x86 体系结构经过二十几年的发展，已经具备了非常成熟的分段分页机制和存储保护机制，图3-3 为32 位x86 系统的主要架构。

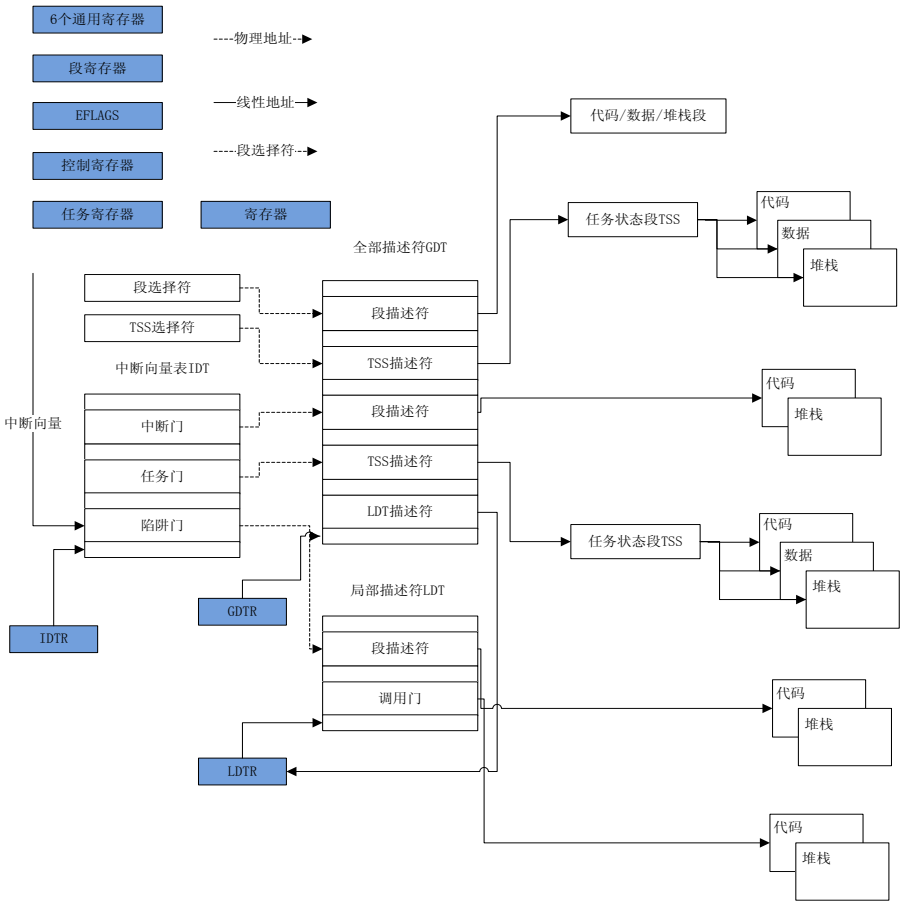

图3-3 x86 体系结构概览

图 3-3的上半部分描绘了x86 体系结构的主要寄存器和系统数据结构的关系。保护模式下，内存存在3种类型的表，它们是全局描述符表（GDT）、中断描述符表（IDT）和局部描述符表（LDT）。其中GDT 和IDT 各只有一个，而LDT 则有多个，分别对应于不同

的进程。寄存器GDTR 和IDTR 指向GDT 和IDT，LDTR 则指向当前进程的LDT。选择符和描述符是两个用的最多的数据结构，保护模式下的段寄存器存放着各个段选择符。中断门、陷阱门、调用门等各种门也是选择符，它们存在于IDT 和LDT 中。

图 3-3的下半部分则是以4K 页面大小作为例子，给出了32 位线性地址到32 位物理地址的映射过程。X86 为二级页表结构，需要经过页目录和页表两次查找再加上偏移量得到物理地址。CR3 寄存器存放着页目录的基地址。当发生进程切换时，改变CR3 的值就相当于切换了整个32 位线性空间。

3.2.3 PC 系统

（1）PC 系统概述

随着 CPU 性能的提升和外设的复杂性增大，PC 系统也变得越来越庞大且越来越复杂，就框架来说都是基于Intel 南－北桥结构的。图3-4 描绘了基于Intel 815E 芯片组的系统架构：

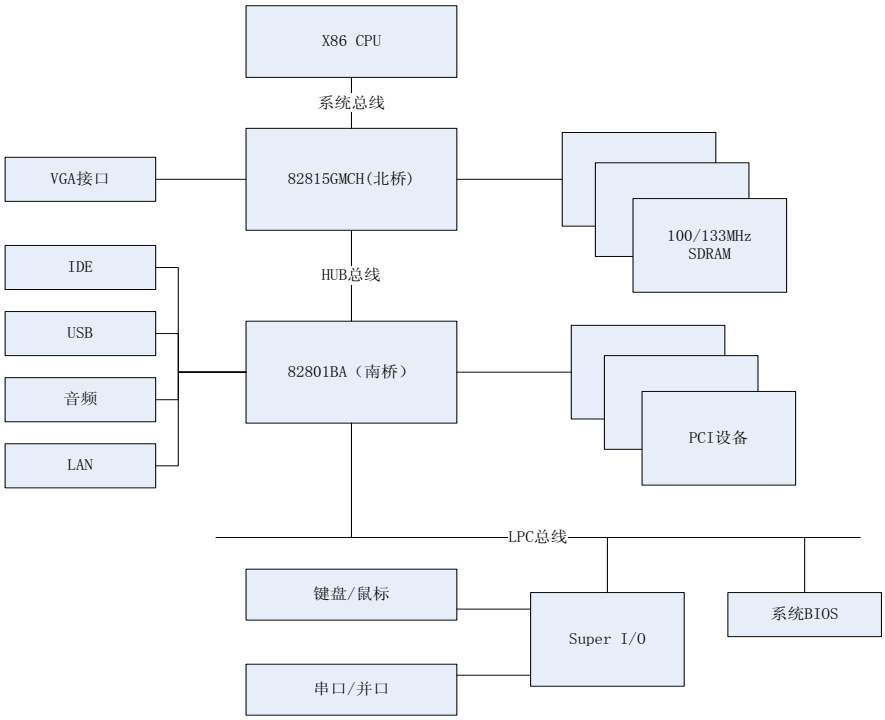

图3-4 由815E 芯片组构成的系统框图

815E芯片组由北桥芯片82815（GMCH）和南桥芯片82801BA（ICH2）组成，GMCH为图形和内存控制中心的简称，它作为CPU、存储器和外设交换的主要通路，它拥有和CPU的接口（64 位系统总线）、SDRAM 接口和ICH2（输入/输出控制中心）的Hub 接口。82815GMCH 除了提供AGP 扩展功能外，内部还集成了显卡功能。除了显示功能，其他绝

大多数的外设功能由82801BA 提供，PC/104 需要用到的功能包括PCI 总线，IDE 接口，LAN，音频和USB，传统外设诸如键盘、鼠标、串/并口由挂接在LPC 总线（一种ISA 总线的换代版本）的Super I/O 芯片连接。

（2）总线拓扑

PC 作为一个完整的计算机系统，发展到至今，出现了各种各样的总线，如处理器本地总线、AGP 总线、PCI 总线、ISA 总线以及USB 总线等。在进行PC 虚拟的时候，必须理清除它们之间的关系，包括物理关系和逻辑关系（主要是逻辑关系）。从图 3-4 我们已经大致了解了这些总线的物理连接关系，处理器本地总线连接CPU 和北桥芯片，南桥芯片实际是和北桥协同工作的，可以作为一个整体（芯片组）来看待。芯片组里包含了丰富的功能，如PCI 主桥，显示控制器，IDE 控制器等，实际上这些都是连接在处理器总线上。因此，这些设备的寄存器分布在CPU 的IO 和存储器空间范围内。

3.3 BOCHS 与 QEMU 技术分析

Bochs 即为一个x86 CPU 的模拟器，它是一个用C++语言写成的高移植性的开源IA-32架构的PC模拟器，可以在很多的流行操作系统上运行。它模拟了Intel x86 CPU，常见IO 设备以及一个BIOS。当前， Bochs 可以编译为模拟386，486，Pentium，Pentium Pro 或AMD64 CPU，包括可选的MMX，SSE，SSE2 和3DNow!指令集。Bochs 中可以运行的操作系统包括Linux，DOS，Windows。Bochs 由Kevin Lawton 在1994 年开始编写。它最初是作为商业产品开发的，但到了2000 年3 月，Mandrakesoft 公司买下了Bochs，使之成为遵循GNU LGPL 的开源软件。2001 年3 月，Kevin 帮助一些开发者把Bochs 移到了sourceforge.net 网站。

QEMU是一款基于软件指令模拟的虚拟机软件，其作者是Fabrice Bellard。在实现原理上它与Bochs和PearPC类似，但在性能上比这两者有着明显的优势。尤其是运用了与其配套的加速器kqemu，QEMU所模拟出的系统能够达到接近真实的物理系统的速度。QEMU能模拟整个电脑系统，包括中央处理器及其他周边设备。它使得为系统源代码进行测试及除错工作变得容易。其亦能用来在一部主机上虚拟数部不同虚拟电脑。

BOCHS与QEMU一样都采用的是指令集虚拟化技术。指令集虚拟化，实际就是将某个硬件平台的二进制代码转换为另一个平台上的二进制代码，从而实现不同指令集间的兼容，这个技术也被形象的称为“二进制翻译”。和实际 CPU 的原理一样，指令集虚拟化虚

拟机CPU的运行方式为: 取指→执行→输出结果。CPU复位后进入实模式，EIP = 0x0000FFF0; CS的cache内的Base= 0xFFFF0000，两种相加得到CS:EIP=0xFFFFFFF0, 该位置是BIOS ROM所在位置. 用户程序从该处开始运行。进入CPU循环后，除了取指和执行工作外，每次还要检测标志位，如果没有复位或退出命令， CPU循环将会一直进行下去。具体流程如图3-5所示[19]。

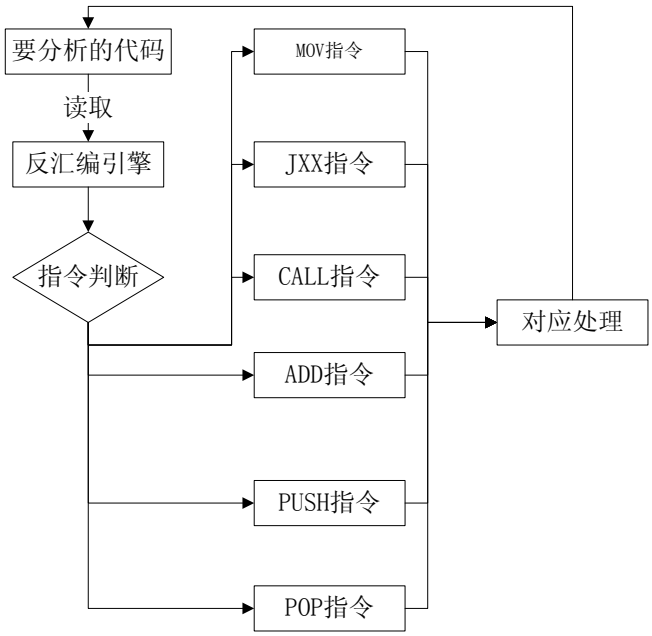

图3- 5 指令集虚拟化流程

二进制翻译是一个复杂的过程，涉及许多技术的结合，包括编译技术，计算机系统结构技术，自适应软件技术等。二进制翻译可以在运行时动态完成，也可以在编译时静态完成，可以仅从翻译用户级代码也可以翻译整个系统代码。动态二进制翻译是在程序运行期间把代码片断从原始指令集翻译为目标指令集，静态二进制翻译则是脱机状态下进行翻译工作，然后再运行时执行翻译过的代码。

一个完正的指令集包括很多部分，如寄存器组织，存储器结构，指令以及自陷和中断方式等。二进制翻译必须完整地再现这些接口和功能。从根本上讲，二进制翻译是一个软件过程，它的实现通常有 2 种不同方式：静态翻译，动态翻译。

3.3.1 静态翻译

静态翻译，类似高级语言的解释器。解释器开发相对容易，也比较容易实现与已有体系结构的高度兼容，但代码执行效率很差。解释执行流程：“取指令--->分析指令--->完成指令所需的操作并修改处理器状态”[19]。这种方式通常也被称为“译码并转发”。这个方式代码可读性很强，但是由于有大量的 switch，导致执行效率很低，并且这些分支指令的

行为很难预测。为此有人提出“线程解释”的优化方法，该方法特点，将译码并转发结构中的转发代码分布到不同类型的指令的解释函数，从而消除一部分 switch(即 goto)。这方法确实解决了些问题，但是对 dispatch 数组的访问成了限制性能提高的一个瓶颈，因为每一段解释代码都必须访问该数组获取下一个指令执行的入口[20]。

其中 BOCHS 就是采用静态模拟来完成对指令的模拟的。使用了一个名为是 BX_CPU_C 的类来完成对 x86 CPU 的 300 多个执行普通指令、近 100 个执行浮点指令和 SSE 指令集的模拟[21]。如图 3-6 为 BOCHS 的虚拟 CPU 的逻辑结构图。图 3-7 为 BOCHS CPU 模拟程序的流程图。

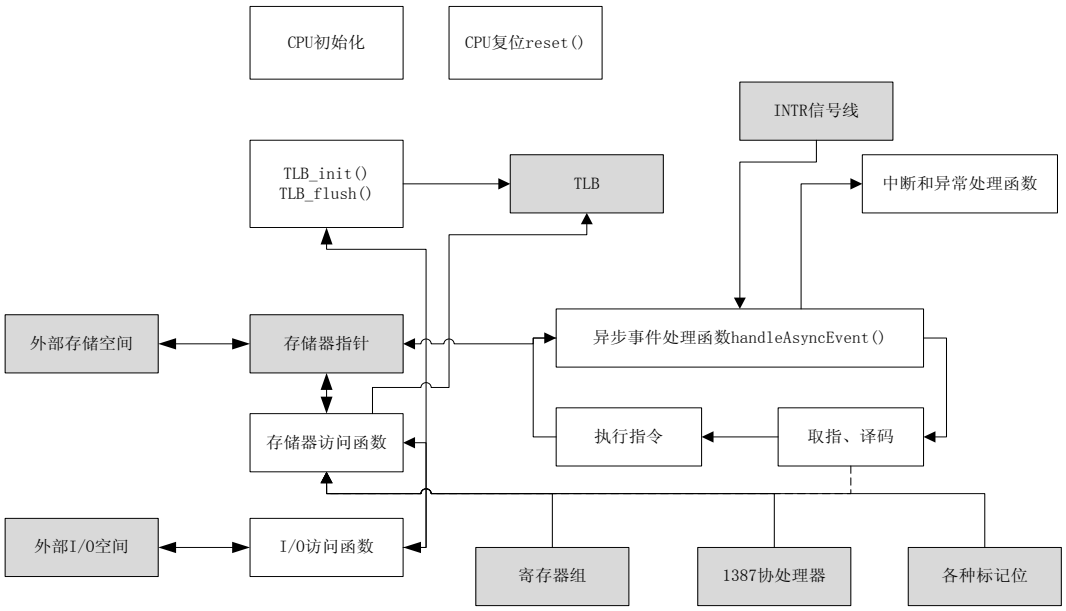

图3-6 BOCHS的虚拟CPU的逻辑结构

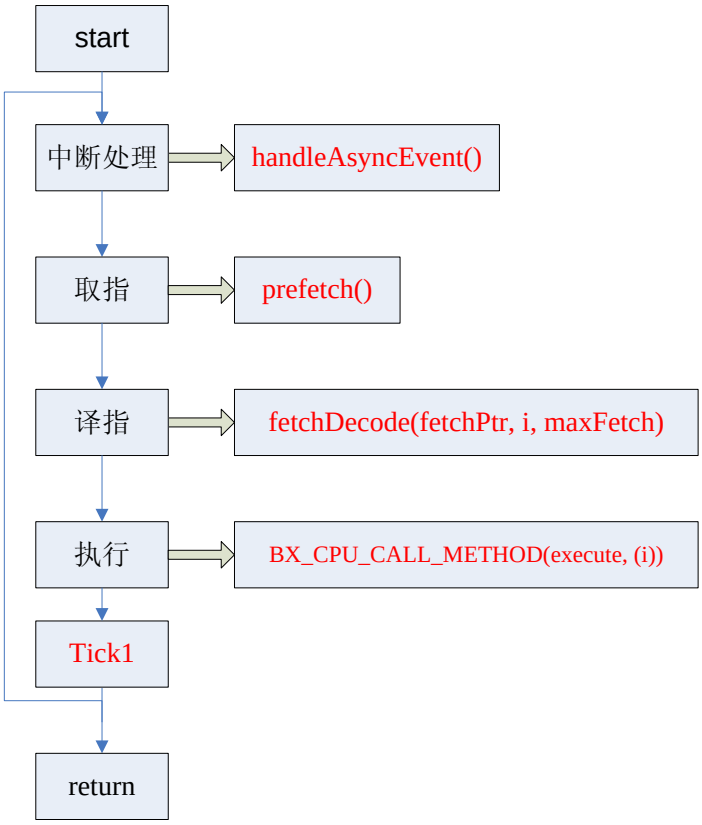

图 3-7 BOCHS CPU 模拟程序流程图

由上可知，静态模拟的流程和真实 CPU 执行流程几乎一致。就是每次取指，判断，然后用真实指令去完成模拟。也正因为如此，使得 BOCHS 在模拟的程序中显得异常的缓慢。根据 BOCHS 源码分析和实验测试，除去内存模拟和 I/O 模拟部分，BOCHS 使用了大约 100-200 条指令模拟一条 CPU 真实指令，在加上实际模拟中的内存模拟和 I/O 模拟，可见 BOCHS 仿真器的速度实现当慢。因而在实际应用中，BOCHS 大多是被开发人员用于替代硬件操作系统来对系统进行调试[22]。

3.3.2 动态翻译

动态翻译，基本思想就是把每一条x86指令切分成为若干条微操作，每条微操作由一段简单的C代码来实现，然后通过中间工具提取相应的目标文件来生成一个动态代码生成器，最后把这些微操作组合成一个函数[23]。

QMUE采用的正是动态翻译技术来完成指令模拟的。正是由于这个强劲的引擎，使QEMU可以在不使用任何加速技术的情况下也能达到良好的速度，并能够横跨多种平台运行，借助于特定版本的GCC编译器，还能够仿真多种架构的处理器。这里我说的动态翻译指的是QEMU早期版本使用的“dynamic translation”，因为从0.10版本开始使用的是

“TCG”[24]，摆脱了对GCC版本的依赖，并且不再需要编译中间工具，做到了真的动态翻译。TCG的全称为“Tiny Code Generator”。实际上TCG的作用也和一个真正的编译器后端一样，主要负责分析、优化Target代码以及生成Host代码。Target指令 → TCG → Host指令。如图3-8所示。

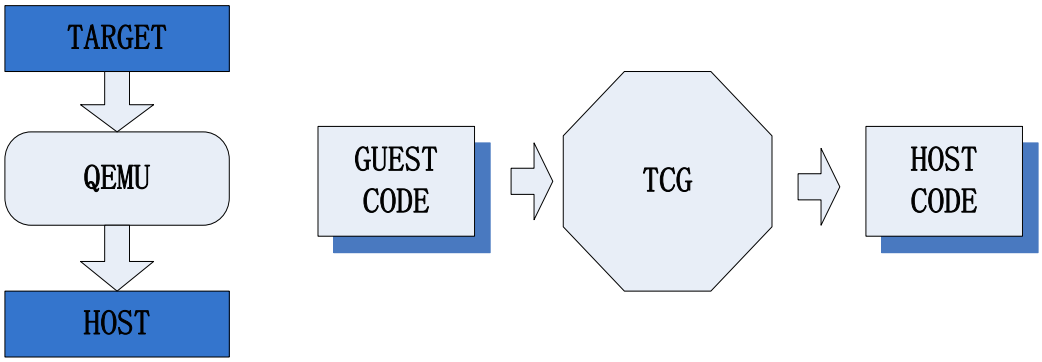

图 3-8 TCG逻辑结构图

每一条Target指令切分成为若干条微指令，每条微指令由一段简单的C代码来实现，运行时通过一个动态代码生成器把这些微指令组合成一个函数，最后执行这个函数，就相当于执行了一条Target指令，如图3-9所示。这种思想的基础是因为CPU指令都是很规则的，每条指令的长度、操作码、操作数都有固定格式，根据前面就可推导出后面，所以只需通过反汇编引擎分析出指令的操作码、输入参数、输出参数等，剩下的工作就是编码为目标指令了。

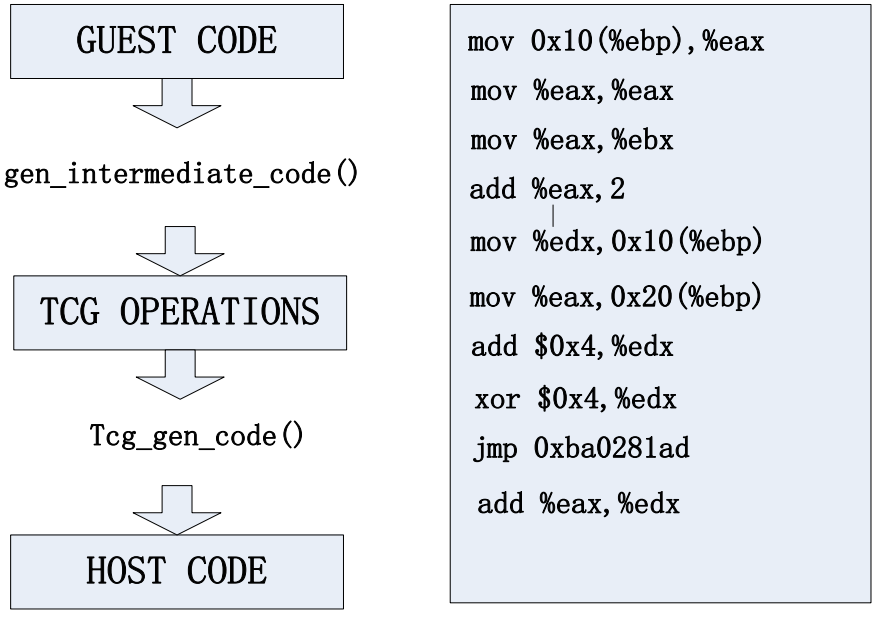

图3-9 TCG解码流程图

当新的代码从 TB 中生成以后， 将会被保存到一个 cache 中，因为很多相同的 TB 会被反复的进行操作，所以这样类似于内存的 cache，能够提高使用效率如图 3-10 所示。而 cache 的刷新使用 LRU 算法。编译器在执行器会从源代码中产生目标代码，像 GCC 这种

编译器，它为了产生像函数调用目标代码会产生一些特殊的汇编目标代码，他们能够让编译器需要知道在调用函数。需要什么，以及函数调用以后需要返回什么，这些特殊的汇编代码产生过程就叫做函数的 Prologue 和 Epilogue[25]，这里就叫前端和后段吧。函数的后端会恢复前端的状态，主要作用为恢复堆栈的指针，包括栈顶和基地址和修改 cs 和 ip，程序回到之前的前端记录点。

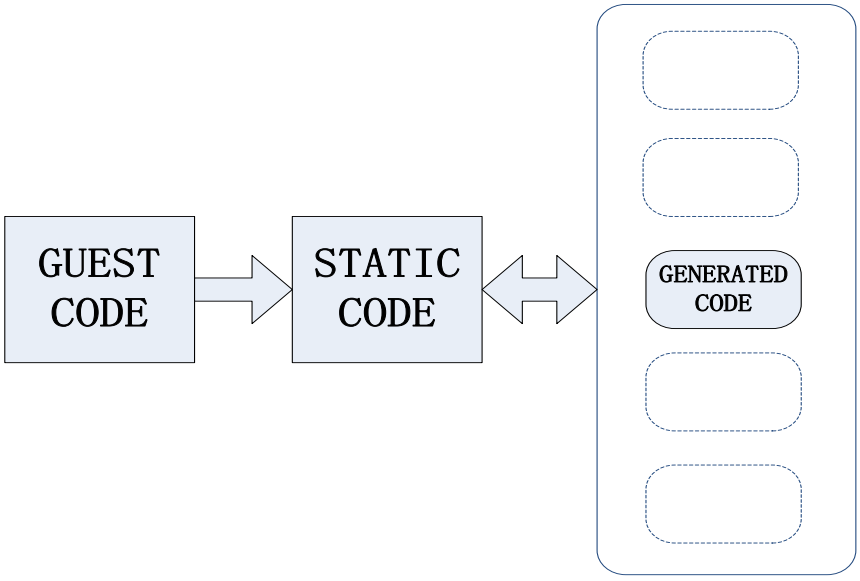

图3-10 Code Cache 结构图

TCG 就如编译器一样可以产生目标代码，代码会保存在缓冲区中，当进入前端和后端的时候就会将 TCG 生成的缓冲代码插入到目标代码中。从代码 cache 到静态代码再回到代码 cache，这个过程比较耗时，所以在 QEMU 中涉及了一个 TB 链将所有 TB 连在一起，可以让一个 TB 执行完以后直接跳到下一个 TB，而不用每次都返回到静态代码部分[26]。具体过程如图 3-11 所示。

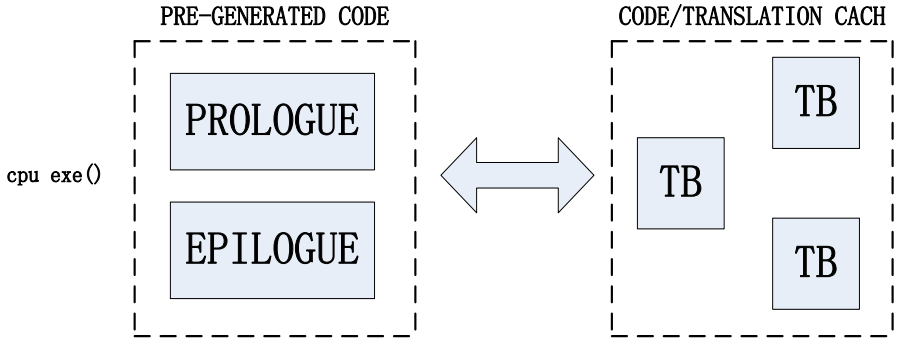

图3-11 TB 链结构图

3.4 本章小结

本章全面介绍了虚拟机技术。根据系统架构的不同，对当前虚拟机产品进行了分类。详细研究了目前流行的x86 PC系统的体系结构，进而研究了虚拟机工作的原理。 随后对基于指令模拟的两款虚拟机产品BOCHS和QEMU进行了对比分析，分析了静态翻译和动态翻译的差异。着重研究了动态翻译，为整个恶意代码监测系统中系统仿真打下基础。

第四章 基于语义的启发式检测模型

“启发式”一词最初由希腊人创造，其词根为“heuriskein”，意思是“to discover”。现在这个术语成为计算机科学的一个专有名词，多指快速而有效地解决复杂问题的算法。反病毒专家首先将这个新概念引入到计算机安全领域中，提出了“启发式扫描(Heuristic Scanning) ”方法。启发式算法是一种基于直观或是经验构造的算法，在可接受的花费(指计算时间、占用空间等)下给出待解决优化问题每一个实例的一个可行解[27]。

4.1 引言

在当前的恶意代码检测领域，检测方法有很多，但最为典型的方法就是基于恶意代码特征指令序列的静态判定方法。这种方法依然被大多杀毒软件广泛利用，并且被作为一种最为主要的方法[28]。

就目前最为流行的恶意代码而言，其主要是以可执行文件形式存在。所以代码中的特征指令序列对应的是相应的机器指令，即汇编指令。而在实际运用中，指令本身是不存在任何危害性的，真正存在危害的是一些相关的系统 API 或是一些 API 的操作组合。很多恶意代码制造者根据这一特性可以通过一些简单的汇编指令变形完成操作一些敏感 API 函数，从而骗过当前杀毒引擎的检测。因而，真正有效的恶意代码检测方案应是基于行为特征的，而不是基于指令序列特征的。根据代码的上下文环境，分析代码的流程、语义、行为，从而判定代码是否是恶意代码。

4.2 相关研究

启发式指的是具有自我发现的能力、运用某种方式和方法来判定事物的知识和技能。启发式分析就是利用计算机病毒的行为特征，结合以往的知识和经验，对未知的可疑病毒进行分析与识别。启发式代码扫描技术实际上就是把这种经验和知识移植到一个反病毒软件的具体程序中。它的基本原理是通过对一系列病毒代码的分析，提取一种广谱特征码，即代表病毒的某一种行为特征的特殊程序代码，当然仅仅是一段特征码还不能确定一定是某一种病毒，通过多种广谱特征码，也就是启发式规则的判断，最后做出合理的判断[29]。

启发式扫描主要分为两类：静态启发式扫描和动态启发式扫描。

4.2.1 静态启发式扫描

静态启发式检测技术主要基于对代码片段的分析来检测病毒。通过对文件外部静态信息进行分类，结合病毒感染形式，模拟跟踪代码执行流程判断是否是病毒。静态检测中包括针对程序存在异常的检测方案，和针对恶意行为规则的检测方案。

针对异常的检测方案中，会对病毒木马所使用的技术进行分类处理，例如入口点模糊隐藏，感染等等方式进行抽象分析，同时归纳出这些异常点，结合分析算法来达到未知病毒检测的效果[30]。

针对恶意行为规则的监测方案中，会对已经形成的病毒家族行为进行归纳和规则演绎，同时对恶意代码进行逆向处理，模拟跟踪代码执行流程，解析特定函数及参数等有效信息来形成规则，匹配已有知识库的规则完成对新增病毒变体的有效检测。

4.2.2 动态启发式扫描

动态启发式检测主要是基于反病毒虚拟机技术。通过模拟Intel CPU、部分计算机硬件（内存、硬盘等）和Windows操作系统构造一个反病毒虚拟机，然后把待检测程序加载到仿真的系统中运行。虚拟机中设置有若干行为和怪异特性监控点，对程序行为和怪异特性进行实时监控，根据是否含有恶意行为或者某些怪异特性侦测病毒。由于被检测程序是在虚拟机中运行，病毒并不会威胁真实计算机系统[31]。流程如图4-1所示。

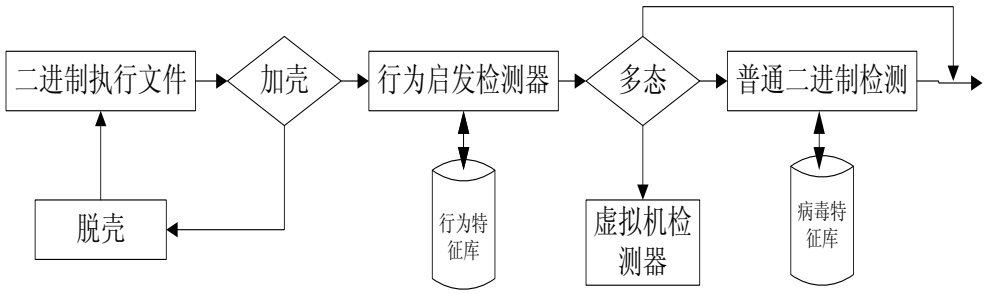

图4-1 动态启发式扫描流程

利用虚拟机引擎模拟 Intel CPU，仿真 CPU 的机器码识别系统、寻址系统和指令解释执行系统。机器码识别系统负责对程序的指令识别，然后把识别出来的指令传递给指令解释系统执行。在指令的执行过程中如果用到内存需要使用寻址系统来寻址访问内存。这个过程中异常捕获系统要实时监测虚拟 CPU 中可能导致的异常，并做成实时响应。简单模拟少数硬件，如硬盘信息、网卡信息等。模拟 Windows 操作系统：该模块主要由 PE 加载系统、API 系统、文件系统、注册表系统和任务调度系统等组成，病毒在运行前需要用 PE

加载系统加载，然后移交给虚拟 CPU 执行，程序执行过程可能需要的 API 等系统的支持，如果病毒程序是多线程的，还需要任务调度系统的支持。模拟的 Windows 操作系统中还应具有相应的异常处理系统，当虚拟 CPU 监测到异常的时候，交由虚拟的 Windows 操作系统进行异常响应。

确定检测的行为点，通过行为点的组合来报毒，需要捕获的一些行为有创建进程、枚举进程列表、结束进程、枚举 AV 进程窗口、释放 PE 文件、提升权限、注册全局消息钩子、拷贝自己到系统目录、添加启动项、加载服务、联网下载、安装 SPI、注册组件、注入远程线程操作 SFC 等[32]。

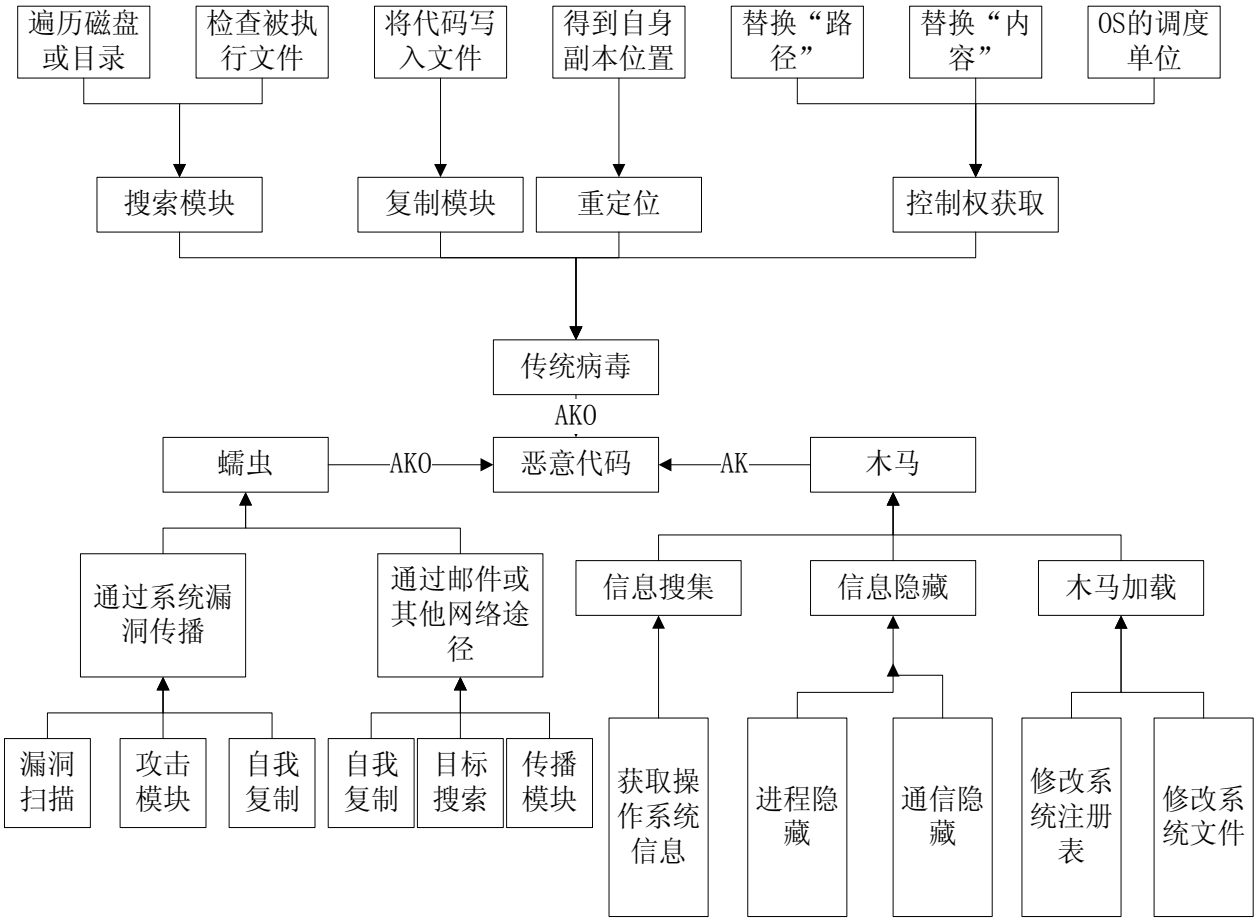

图 4-2 动态启发式扫描行为监测点

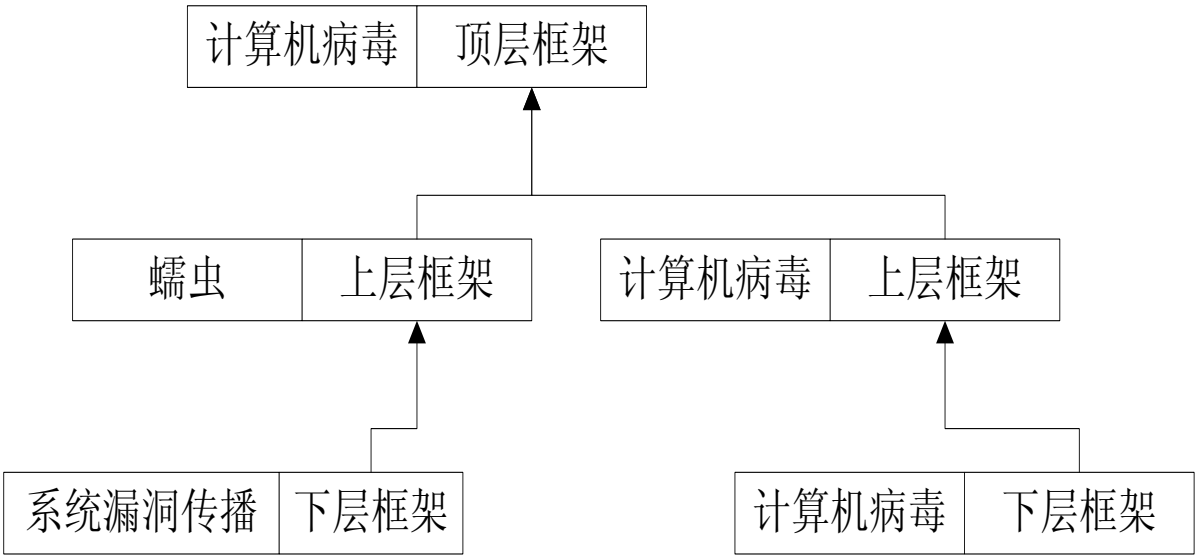

图 4-3 三层框架结构

定义 1.待检测代码的危害值 F = $\sum_{i=1}^{\infty}(n_i q_i)$

其中q_i为对应的特征行为危害权值，n_i为相应的特征行为出现的次数。

根据待检测程序的行为特征，提取相应的特征字段并对每个行为特征增加一个模糊权值，比如对于 M 行为特征q_i，对整个程序执行过程进行统计。因此，恶意代码行为特征库设计的首要任务是对各种行为特征模糊量化处理，这项工作可以根据病毒分析专家的经验或是借助人工智能的技术辅助处理。其次，注意分析每个行为特征的关联,根据专家的分析设计简单的权植推理规则，比如 C 特征(修改计算机基本配置)如果只是单独出现此行为，则认为属于正常，因为这可能是某软件的安装程序；如果 C 与 A 特征同时出现,则该文件恶意代码的概率就提高了,因此其检测权值相应地要提高。例如：如下所示的规则。

```
Rule1:
If (C == something and other == 0)
then
Filecheck.Weight = 0;
Rule2:
If( C == something and A == something)
then
Filecheck.Weight=(C.Weight+A.Weight)*2；
```

这种考虑行为特征关联而动态调整文件检测权值的方法有利于提高综合分析恶意代

码行为的准确性。在改进的病毒检测引擎中，恶意代码行为分析器根据简单的规则，分析文件的检测权值，如果超出一定的阈值就送虚拟机处理，作为疑似恶意代码处理；反之，认为是健康文件，放弃该文件的检测工作，因此节省了大量的恶意代码特征码匹配时间。这样不仅改善了运用虚拟机分析检测恶意代码的效率瓶颈，而且根据恶意代码行为特征检测可以预警未知的恶意代码。

由此可见，静态启发式扫描态技术并不精确，是特征码扫描技术的延伸。它是一种静态的行为检测技术，即把病毒可能执行的一些行为也作为特征码加入了病毒库中。动态启发式扫描技术主要增加了对 CPU 的模拟。对于加密病毒来说，直接使用静态启发式扫描是不能判定的。动态启发式扫描则解决了这个问题，它模拟了一个基本运行环境的计算机，对可能是病毒的文件先进行模拟运行，等加密病毒自解密后再进行静态启发式扫描。

4.3 基于语义的可信模型

4.3.1 基本可信模型

A 符号定义：定义基本的符号，表示程序和操作。

元素集：$E=\{e_1,e_2,e_3,...,e_n\}$ 每一个程序都是一个元素。

对象集：$O\subseteq E$ O 是 E 的子集，每个操作对象都是 O 中的一个元素。

主题集：$S\subseteq E$ S 是 E 的子集，每个操作主题都是 S 中的一个元素。

信任元素集：$T\subseteq E$ T 是 E 的子集，每个 T 中的元素都可被无条件信任。

结果集：$R=\{r\mid r\in yes,no,error\cup 2^E\}$，R 是一个操作结果集，如果这个操作是验证操作，则结果应该 yes，no，error 中的一个；如果这个操作是一个度量操作，则结果应该被设为唯一元素或者 error，如果两者都不是的话，结果应该是一个元素集或者 error。

操作识别集：操作识别被定义为两个集合：$A_1=\{boot,stop,measure,verify\}$

所有操作的集合是：$OP=OP_1\cup OP_2$

时间：$T=(0,1,...,t,...)$ ，时间定义了时间序列

操作列表：$x\in X=OP^T$,x 被描述为: $x=(x^0,x^1,x^2,...x^t,...)$

结果列表：$y\in Y=R^T$ ，其中 y 可以被描述为 $y=(y^0,y^1,y^2,...y^t,...)$ [33]

B 定义信任关系矩阵，度量映射函数：

信任关系矩阵：信任关系是一个有序的二元序对，描述了主题$e_s \in S$对$e_o \in O$的信任，信任关系矩阵如下定义：$M = \{m \mid m \subseteq s*o$且$\forall(e_s, e_o) \in m,$表示$e_s$信任$e_o\}$，每个矩阵都是一个信任关系集合。m 是一个 E 上的信任关系矩阵，为了定义 m 的闭包。首先定义$\forall x \in S$，$y \in O$，$(x, y) \in m^n \Leftrightarrow \exists e_1, e_2, e_3, ..., e_{n-1} \in S \cap O$，$(x, e_1), (e_1, e_2), ..., (e_{n-2}, e_{n-1}), (e_{n-1}, y) \in m$，

信任关系矩阵闭包可以如下描述：$c(m) = U_{i=1,2,...m^i}$

度量映射被定义为一组映射：$f \in F : s*o \to o$,而每一个 f 可以这样定义：$f(e_i, e_j) = e_k$,其表达主题e_i度量对象e_j得到结果e_k。

C 定义系统的状态，系统状态跃迁规则和系统：

1.系统状态：

$v \in V \subseteq 2^E * 2^T * M *$，且每个状态 v 可以被描述为四元组

v= (E_v, T_v, m_v, F_v)，E_v代表运行的软件程序和硬件，

T_v代表可信任的元素，

m_v代表 v 的信任矩阵，

F_v代表 v 的度量映射，

2.系统状态跃迁规则：

$\delta : O P * V \to R *$，这个操作是归因于系统状态间的跃迁，并且系统状态改变也满足定义的规则。

3.系统：

$\sigma(x, y, W_0, v)$，x 表示操作列表，y 表示结果列表，$W \subseteq OP * R * V * V$ 受规则$\omega \subseteq \delta$。且只有一个 i 满足$1 \le i \le s, (r_j, v') = \delta_i(op_k, v)$,r∈R，$op_k \in OP$。$v_0$是初始系统状态。

由上理论可知，通过建立代码行为可信模型，根据代码运行的具体行为建立度量映射来记录代码跃迁规则，从而确定代码的各状态，以此来判断代码的善恶度[34]。

4.3.2 基于信息熵的判断模型

由 4.3.1 可知，我们可以通过代码的具体行为来记录代码的各状态，根据具体的状态

值来判断代码的善恶度。但在有些情况下，代码运行的都是大量的重复的行为，通过上述基于可信机制传递的方式在具体应用中会消耗大量的系统资源，而且效果并不明显。我们引入了基于信息熵的判断模型[35]。

4.3.2.1 信息熵定义

在信息论中，熵被用来衡量一个随机变量出现的期望值。它代表了在被接收之前，信号传输过程中损失的信息量，又被称为信息熵。信息熵也称信源熵、平均自信息量。

一个 X 值域为{x1, ..., xn}的随机变量的熵值 H 定义为：$H(X)=E(I(X))$，其中，E 代表了期望函数，而 I(X) 是 X 的信息量（又称为信息本体）。I(X) 本身是个随机变量。如果 p 代表了 X 的机率质量函数（probability mass function），则熵的公式可以表示为：$H(X)=\sum_{i=1}^{n} p(x_i)I(x_i)=-\sum_{i=1}^{n} p(x_i)\log_b p(x_i)$，在这里 b 是对数所使用的底，通常是 2, 自然常数 e，或是 10。当 b = 2，熵的单位是 bit；当 b = e，熵的单位是 nat；而当 b = 10,熵的单位是 dit。pi = 0 时，对于一些 i 值，对应的被加数 $0*\log_b 0$ 的值将会是 0，这与极限一致[36]。

4.3.2.2 代码二进制信息熵

在 windows 操作系统中，可执行文件是以 PE 格式存在的。如图 4-4 可知，PE 格式文件分为不同的区块。文件的开头是 DOS 首部，这个是 PE 格式文件的标记位。紧接着的是 PE 文件头，用来标记该文件的大小、基址、格式版本等信息。其后是 PE 文件的具体数据部分，比如块表、块和调试信息。区块中包含着可执行文件的具体代码或是数据，考虑到内存加载的问题，各个区块是一个连续的就够，按页边界对齐，大小上没有限制。

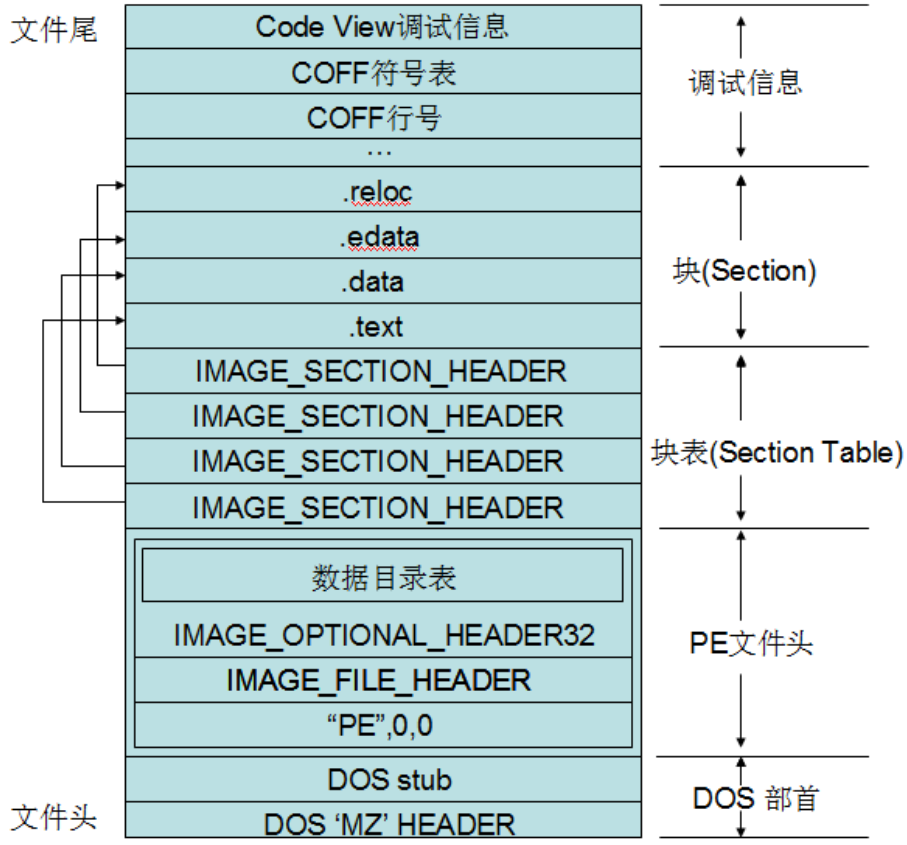

图 4-4 windows PE 结构图

如图 4-5 所示，PE 文件在加载到系统内存中时，系统的 PE 加载器回去文件做一点适当的处理。大致的文件结构顺序没有改变，将文件较高的偏移位加载到较高的内存地址中。只是，系统会根据系统内存对齐大小来处理文件，这个值通常比文件块对齐大小大。所以造成了，内存中的 PE 文件和磁盘中的 PE 文件结构布局一致，但具体数据的偏移可能发生了改变。

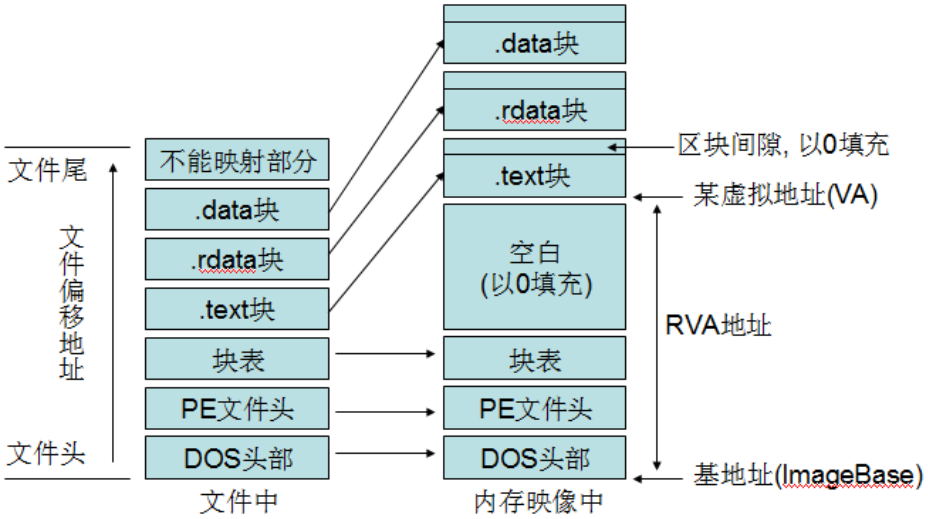

图 4-5 PE 文件内存示意图

由于 PE 结构的限制与编译器的特点，实际运用的绝大部分正常 PE 文件中的信息都有冗余。而真正大部分恶意代码的作者为了避免杀毒软件的轻松定位，会采取一些加密、压

缩手段对代码进行特殊处理。这样就造成了 PE 文件的信息熵发生了比较明显的变化。

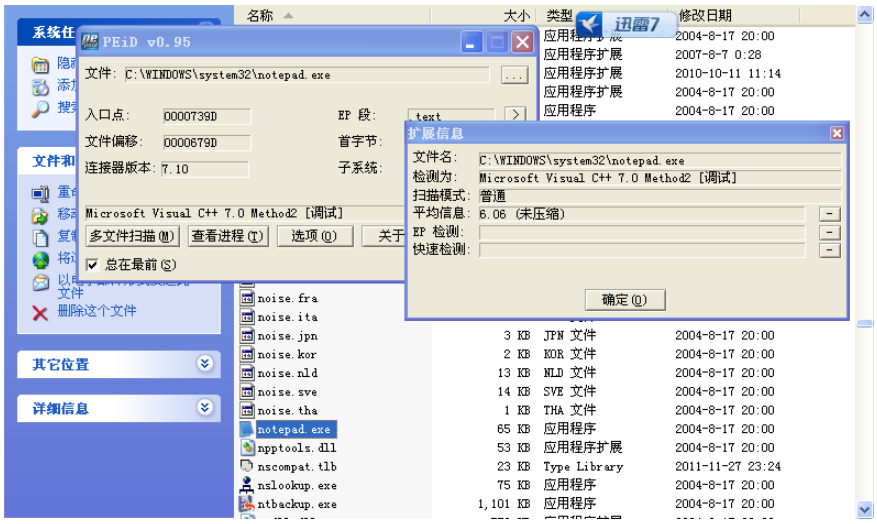

图 4-6 未加密文件信息熵

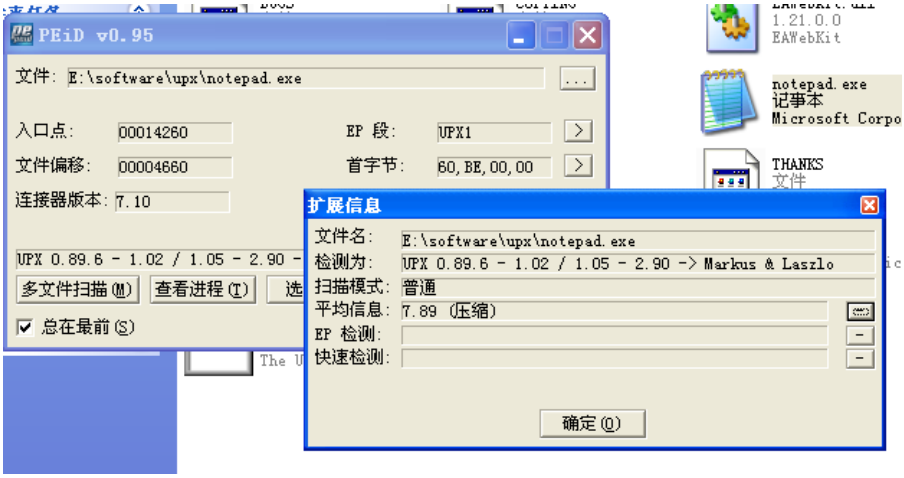

图 4-7 加密文件信息熵

4.3.2.3 代码行为信息熵

研究发现，我们出了可以对未知代码整体二进制代码信息熵进行分析判断，还可以对代码行为信息的熵值进行分析。这里我们将未知代码的行为视为一种随机事件，同一个随机事件发生，对每一个收信人都有不同的意义。如果每一个收信人对各事件都考虑一个意义的因素，定量地给出该事件的权重，那么香农的信息熵（平均信息量）就应该修正为加权熵。于是有了加权熵的定义：

设一个随机变量 X 包含了一系列（q 个）随机事件

$a_1, a_2, a_3, ..., a_q$，（q 为自然数）

对于某收信人来说，每个事件权重相应的为

$w_1, w_2, w_3, ..., w_q$，（$w_i \geq 0$，i=1,2,3,…,q）

如果事先知道每个事件发生的概率依次为

$p_1, p_2, p_3, ..., p_q, (0 \le p_i \le 1, i=1,2,3,\ldots,q)$

且这些概率满足完备条件 $\sum_{i=1}^{q} p_i = 1$

即所有可能的事件发生的概率总和为 1，则这个随机变量 X 可用下述的概率空间来描述：

$$\begin{bmatrix} x \\ w \\ p \end{bmatrix} = \begin{bmatrix} a_1, a_2, a_3, ..., a_q \\ w_1, w_2, w_3, ...w_q \\ p_1, p_2, p_3, ..., p_q \end{bmatrix} \quad (4\text{-}1)$$

那么，加权熵定义为

$$H_w(X) = -\sum_{i=1}^{q} w_i p_i \log p_i$$

我们将一下可以确认完全不可能有危害的行为的权值定位 0，未知行为定位 1，其他存在危害的行为根据危害程度的不同相应确定好其权值。然后通过虚拟系统模拟代码执行过程，确定代码的行为熵值来作为判断依据[37][38]。

例如，一些感染性病毒会不断的枚举磁盘上的可执行文件，并予之感染。因而代码会频繁调用 FindNextFile、CreateFile、ReadFile 和 WriteFile 四个 API，这四个 API 是常用 API，我们不能因为代码调用了这些 API 而断定代码为恶意代码。但通过代码行为信息熵去分析，我们会发现感染性病毒和一般正常代码的不同。设这四个 API 的权值为 5，一些可以确定的不存在危害的 API（如窗口消息处理 API）的权值设为 0，其他一些未知的 API 权值为 1。通过虚拟仿真系统对类似感染性病毒分析可以发现，代码行为信息熵值接近 10，而正常代码的信息熵值接近 1，之间有明显区别。

4.4 本章小结

本章全面介绍了基于语义的启发式恶意代码检测技术。首先阐述了根据当前恶意代码发展的趋势，利用行为特征分析取代传统二进制特征码定位的优势。介绍了静态启发式扫描和动态启发式扫描两种扫描方式。此外，提出了基于语义的可信模型和信息熵判断模型。

第五章 高性能云服务器设计

为了充分利用网络的共享性，来达到全民动员共同抵制恶意代码入侵，需要建设一台高性能服务器或是一个高性能服务器群[39]。对于一位程序员来说，由于具体编程中考虑到多线程编程，以及多线程之间的同步异步问题，要编写出高性能服务器程序为成千上万用户提供服务是一件不简单的事。

对于服务器程序，I/O 是制约系统性能最为关键的因素。对于需要处理大量连接的高并发服务器程序，异步 I/O 几乎是不二的选择。Linux 和 Windows 都为异步 I/O 构建了大量的基础设施。

当线程向设备发起一个 I/O 异步 I/O 请求后，这个 I/O 请求传递到设备驱动程序，后者在完成 I/O 后通知线程。所以，线程不用等待 I/O 完成而挂起，可以继续做其它的操作。当然，最终会在某个点等待 I/O 完成，比如，需要对返回数据进行处理时[40]。

5.1 Windows Socket I/O 模型

在Windows应用程序中共有六种socket I/O模型对网络I/O进行管理包括blocking(阻塞)、select(选择)、WSASsynSelect(异步选择)、WSAEventSelect(事件选择)、overlapped(重叠)以及 completionport(完成端口)。

其中阻塞模型的特点比较简单比较直接。在处理 I/O 时，每个套接字连接使用一到两个线程。之后每个线程都将发生阻塞操作以等待相应的 I/O 完成，如 send 和 recv 操作。

阻塞模型的优势是其简洁性。其非常适用于简单的应用程序，因为创建多线程会消耗宝贵的系统资源。缺点是很难将他扩展以处理大量连接应答请求。

Select 模型是异步 I/O 模型中最为基础、简单的一种，尤其在 UNIX 和 LINUX 操作系统中使用的较为之多。之所以称其为“Select 模型”，是因为该模型的核心部分就是一个 select 函数。Select 模型就是通过 select 函数来实现对 I/O 的管理。该模型最早是在 UNIX 操作系统中提出的，后在 windows 操作系统中，Select 模型已集成到 Winsock 1.1 中。

另外在 windows 操作系统中，有一些比较特殊的异步 I/O 模型。由于 windows 系统是基于窗口的，根据这一特点，系统开发人员开发了一种基于 windows 窗口消息机制的异步 I/O 模型，即 WSASsynSelect 模型。利用这一模型，应用程序会像处理 windows 窗口消息一样，用一个窗口处理例程处理网络 I/O 消息。

由于考虑到窗口消息机制的通用性，尤其是服务器程序，并不一定有窗口，windows 还提出了 WSAEventSelect 模型。和 WSAAsyncSelect 模型类似的是，它也允许应用程序在一个或多个套接字上，接收以事件为基础的网络事件通知。不同的是，在 WSAAsyncSelect 模型中，网络 I/O 消息被视为 windows 窗口小心，而在 WSAEventSelect 模型中网络 I/O 消息被视为更为一般的事件消息。应用程序在使用该模型时，只需要创建相应的网络事件，通过投递和等待事件相应在处理所有网络 I/O 消息。

在 WSAEventSelect 模型的基础上，研究者发现不停的投递和等待事件对系统的消耗也很大。为了客服这一问题系统工作者研究出了重叠 I/O 模型，该模型原理大致和 WSAEventSelect 模型类似，不同的是 WSAEventSelect 模型每次只投递一个 I/O 请求，而重叠 I/O 模型一次投递一个或是多个 I/O 请求。应用程序可以根据提交的请求，在完成之后由相应的例程去处理。例如 WSASend 和 WSARecv[41]。

5.2 windows 完成端口模型

实验表明，上述异步 I/O 模型在不考虑人为出错的情况下，可以提供最多 2000 客户端的同时 I/O 应答。随着网络的发展，这已经远远不能满足现行网络服务的要求。为此 windows 系统开发者在 windows NT 操作系统以后退出了一套新型的异步 I/O 模型，即完成端口模型。完成端口模型是迄今为止 windows 平台上最为复杂，也是最为高效的一种 I/O 模型。从理论上而言，完成端口模型最多可以同时提供 65535 客户端的同时 I/O 应答。而在实际运用中，在不考虑人为出错的情况下，一般的普通配置的 PC 完全能够同时处理 20000 以上客户端的 I/O 应答[42]。

5.2.1 完成端口简介

Windows 完成端口是系统的一种内核对象，完成端口实质也是一种异步重叠 I/O 模型。当完成断后充分的利用了线程池的优势，避免反复创建线程的开销。一般而言，根

据实验数据证明，线程池中线程的数量为 CPU 个数的两倍时，多线程的优势发挥的最为理想。完成端口的最大优点在于其管理海量连接时的处理效率，通过操作系统内核的相关机制完成 IO 处理的高效率。如图 5-1 为完成端口模型结构图。

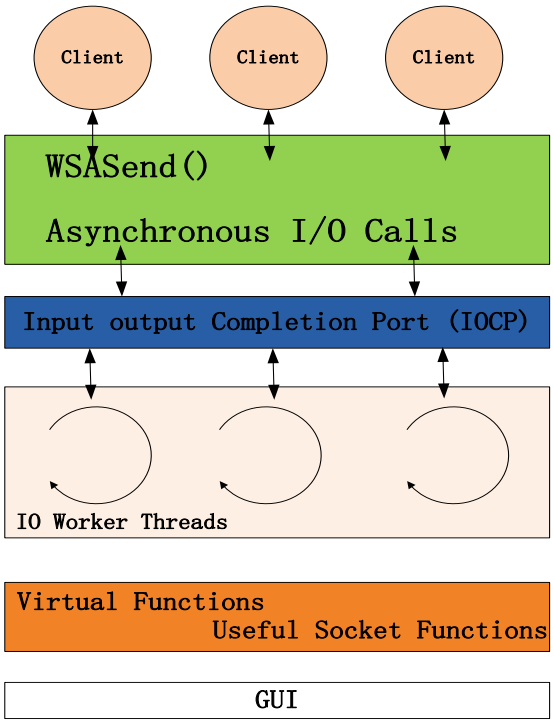

图 5-1 完成端口整体结构图

5.2.1.1 创建完成端口

由上所述，完成端口实质是一个系统内核对象，具体表象其为一个句柄数据结构，在使用时首先应将其和一个具体的设备句柄关联起来，CreateIoCompletionPort 就是它的创建函数：

```
HANDLE CreateIoCompletionPort(
    IN HANDLE FileHandle,
    IN HANDLE ExistingCompletionPort,
    IN ULONG_PTR CompletionKey,
    IN DWORD NumberOfConcurrentThreads
    );
```

具体使用分两步：

第一步，创建一个新的完成端口内核对象，可以使用下面的函数：

```
HANDLE CreateNewCompletionPort(DWORD dwNumberOfThreads)
{
    return
  CreateIoCompletionPort(INVALID_HANDLE_VALUE,NULL,NULL,dwNumberOfThreads)
  ;
 };
```

第二步，将刚创建的完成端口和一个具体的设备句柄关联起来，可以使用下面的函数：

```
bool AssicoateDeviceWithCompletionPort(HANDLE hCompPort,HANDLE
hDevice,DWORD dwCompKey)
{
    HANDLE h=CreateIoCompletionPort(hDevice,hCompPort,dwCompKey,0);
    return h==hCompPort;
};
```

a) CreateIoCompletionPort 函数也可以一次性的既创建完成端口对象，又关联到一个有效的设备句柄

b) CompletionKey 是一个可以自己定义的参数，我们可以把一个结构的地址赋给它，然后在合适的时候取出来使用，最好要保证结构里面的内存不是分配在栈上，除非你有十分的把握内存会保留到你要使用的那一刻。

c) NumberOfConcurrentThreads 通常用来指定要允许同时运行的的线程的最大个数。通常我们指定为 0，这样系统会根据 CPU 的个数来自动确定。

创建和关联的动作完成后，系统会将完成端口关联的设备句柄、完成键作为一条纪录加入到这个完成端口的设备列表中。如果你有多个完成端口，就会有多个对应的设备列表。如果设备句柄被关闭，则表中自动删除该纪录。

5.2.1.2 完成端口线程的工作原理

完成端口可以帮助我们管理线程池，但是线程池中的线程需要我们使用_beginthreadex来创建，其中关键函数为 GetQueuedCompletionStatus。该函数原型：

```
BOOL GetQueuedCompletionStatus(
        IN HANDLE CompletionPort,
        OUT LPDWORD lpNumberOfBytesTransferred,
        OUT PULONG_PTR lpCompletionKey,
        OUT LPOVERLAPPED *lpOverlapped,
         IN DWORD dwMilliseconds
   );
```

这个函数试图从指定的完成端口的 I/0 完成队列中抽取纪录。只有当重叠 I/O 动作完成的时候，完成队列中才有纪录。凡是调用这个函数的线程将被放入到完成端口的等待线程队列中，因此完成端口就可以在自己的线程池中帮助维护这个线程。

完成端口的 I/0 完成队列中存放了当重叠 I/0 完成的结果纪录，该纪录拥有四个字段，前三项就对应 GetQueuedCompletionStatus 函数的 2、3、4 参数，最后一个字段是错误信息 dwError。我们也可以通过调用 PostQueudCompletionStatus 模拟完成了一个重叠 I/0 操作。

当 I/0 完成队列中出现了纪录，完成端口将会检查等待线程队列，该队列中的线程都是通过调用 GetQueuedCompletionStatus 函数使自己加入队列的。等待线程队列很简单，只是保存了这些线程的 ID。完成端口会按照后进先出的原则将一个线程队列的 ID 放入到释放线程列表中，同时该线程将从等待 GetQueuedCompletionStatus 函数返回的睡眠状态中变为可调度状态等待 CPU 的调度。所以线程要想成为完成端口管理的线程，就必须要调用 GetQueuedCompl- etion Status 函数。出于性能的优化，实际上完成端口还维护了一个暂停线程列表。

5.2.2 数据重组

由于完成端口的异步性质，我们不能保证每次收到的数据都为完整的数据，这次就需要在传送数据时加上一些额外的标记，以便对方可以识别，然后对数据进行重组后在进行处理。如图 5-2 所示。

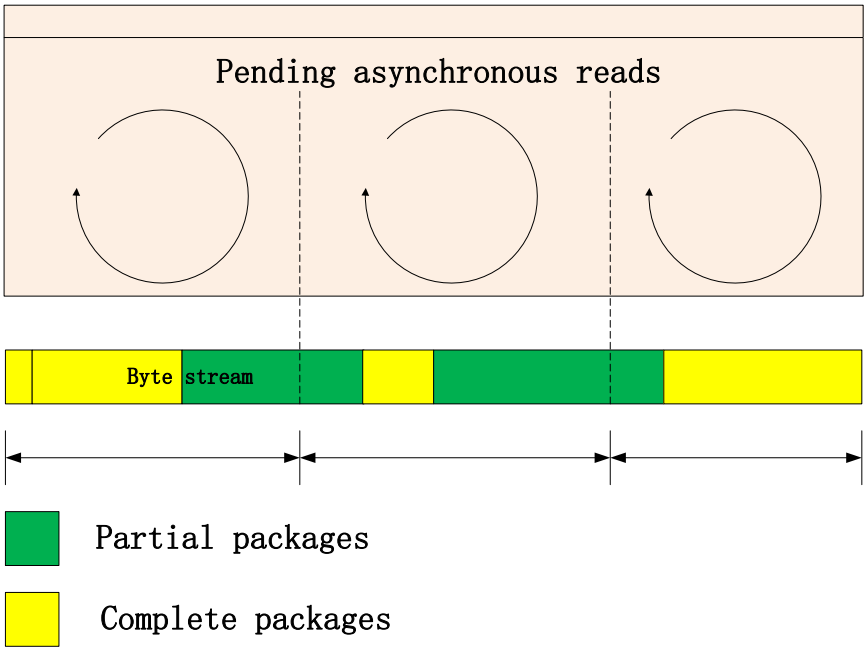

图 5-2 数据重组示意图

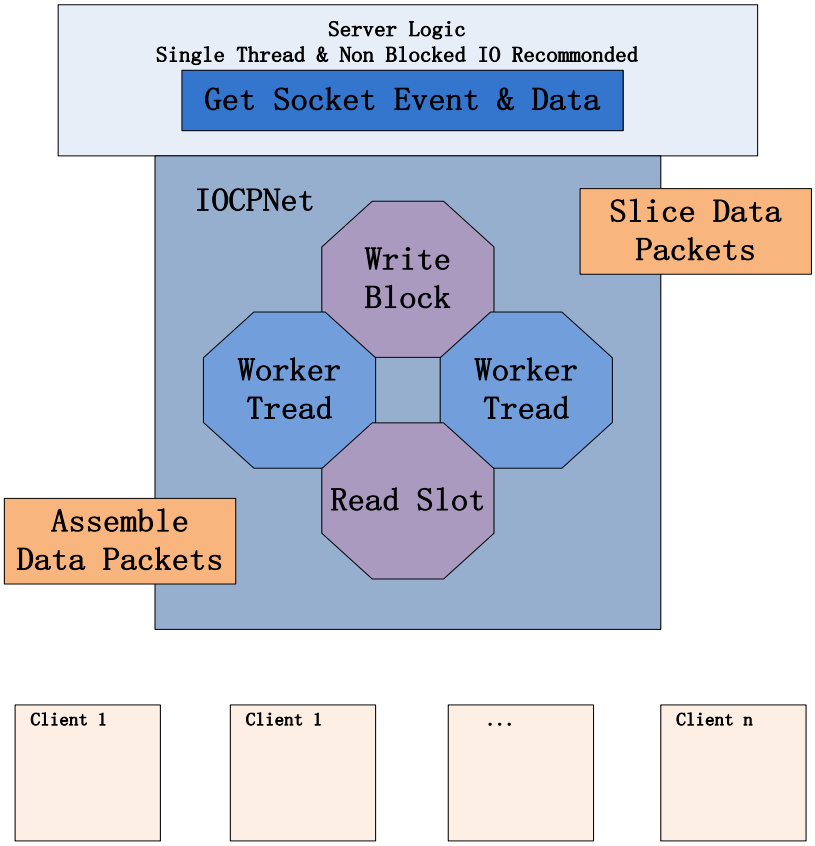

图 5-3 完成端口工作交互图

5.3 本章小结

本章全面介绍了高性能服务器的构建，为监测系统的云服务平台提供支持。着重研究了windows平台下网络应用程序的六种socket I/O模型。此后，详细研究了windows完成端口网络模型，分析了其整个工作流程。

第六章 原型系统实现与测试

6.1 基于云计算的恶意代码监测系统实现

6.1.1 体系结构

原型系统的实现主要分为两部分应用：用户端应用和服务器端应用。服务器端应用：这里的服务器端应用泛指的是多个不同功能的服务器相互联接组成的服务器集群，对外表现为一个整体，相互配合共同完成相应功能，包括哈希匹配服务器、恶意代码检测服务器和特征行为提取服务器。客户端应用为一个功能简单的杀毒引擎，包括二进制特征码匹配引擎、行为特征匹配引擎、小型的二进制以及行为特征数据库和可疑代码提交系统。

以下为系统的几个重要组件：

哈希匹配服务器：主要功能为根据客户端提交的代码的哈希值，在服务器的哈希值数据库中进行查找，并返回结果，从而对客户端代码执行识别。

恶意代码检测服务器：主要功能为对客户端提交的未知代码，通过服务器强大的计算能力，以及配合人工分析对其进行确定。

特征行为提取服务器：对确定的恶意代码行为进行分析，提取特征行为，并进行储存。

二进制特征码匹配引擎：用于对未知代码进行简单的二进制特征匹配。

行为特征匹配引擎：用于对未知代码进行深层次的行为特征匹配。

小型的二进制以及行为特征数据库：用于对常见的恶意代码进行快速识别。

可疑代码提交系统：对用户认为的可疑的未知代码提交给云服务器进行识别。

6.1.2 系统工作流程

系统的主要工作流程如下：

1. 用户首先通过网络下载用户端程序，安装完成后用户电脑会拥有一个功能简单的

杀毒引擎，包括二进制特征码匹配引擎、行为特征匹配引擎、小型的二进制以及行为特征数据库和可疑代码提交系统。

2. 当用户选定对一个未知代码进行检测时，杀毒引擎会根据本地的小型二进制特征码数据库，对代码进行快速识别。

3. 二进制特征码匹配引擎匹配成功，告诉用户该代码为恶意代码。否则利用行为特征匹配引擎，对未知代码进行深层次分析。

4. 行为特征匹配引擎成功，告诉用户该代码为恶意代码。否则利用哈希算法计算出未知代码的哈希值，提交给哈希匹配服务器，

5. 哈希匹配服务器根据用户提交的代码哈希值，利用服务器储存的哈希数据库进行匹配。匹配成功告诉用户，该代码为恶意代码。

6. 用户通过可疑代码提交系统将认为可疑的未知代码提交到恶意代码检测服务器，服务器利用自身强大的计算能力，以及结合人工分析手段对未知代码进行分析判断。确定为恶意代码，则计算出该代码的哈希值，提交给哈希匹配服务器，保存入数据库。并将该代码提交给特征行为提取服务器。否则不做任何处理。

7. 特征行为提取服务器对恶意代码检测服务器提交的恶意代码进行行为特征分析，提取行为特征保存，并更新给用户的行为特征数据库。

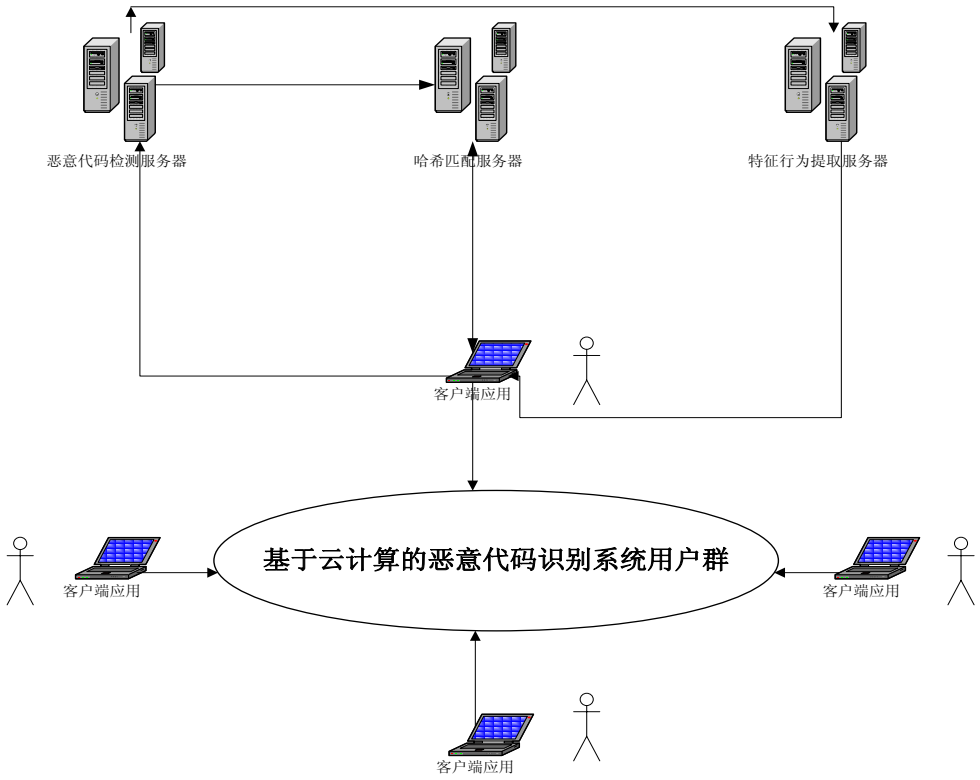

图 6-1 监测系统示意图

有益效果：本发明方法提出了基于云计算的恶意代码识别方案，主要用于快速准确的识别恶意代码，达到保护系统不受恶意代码攻击的目的。下面我们给出具体的说明。

1．高效性

由于客户端只有二进制特征码匹配引擎和简单的行为特征匹配引擎，并且特征数据库中的特征比较少。如果没有识别出，客户端会简单的计算出代码的哈希值，提交给哈希匹配服务器进行匹配。整个过程比现行的杀毒引擎要快速高效。

2．高识别率

由于采用了云计算服务，任何客户端识别出的恶意代码，系统都会计算出该代码的哈希值保存到哈希匹配服务器，以供其他客户端识别。另外，在客户端无法识别的未知代码，会通过可疑代码提交系统提交给云端恶意代码检测服务器。恶意代码检测服务器依靠自身的高性能处理能力，以及加上病毒分析专家的人工分析对被提交的代码做极其准确的判别。

3．低误报率

由于采用了云计算服务，大量的对未知代码检测工作交由恶意代码检测服务器以及加上病毒分析专家的人工分析，判断的结果相对而言较为准确。

4. 轻巧性

本系统充分的发挥了云计算的优势，不再像传统的杀毒软件那样经常的更新病毒特征库。而是在客户端保存一份简单常用的特征库，而在云端服务器上更新一份比较全面完善的特征库。特征行为提取服务器根据大量类似的恶意代码提取出行为特征更新给用户数据库。

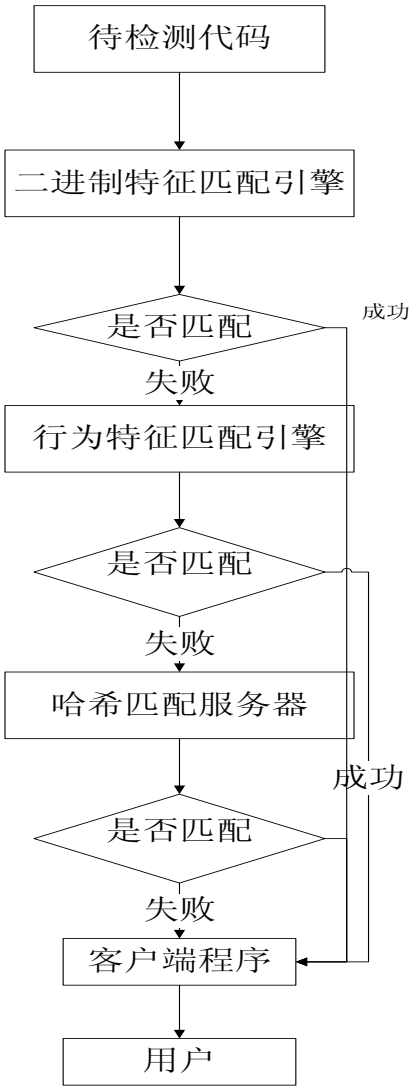

图 6- 2 客户端检测流程图

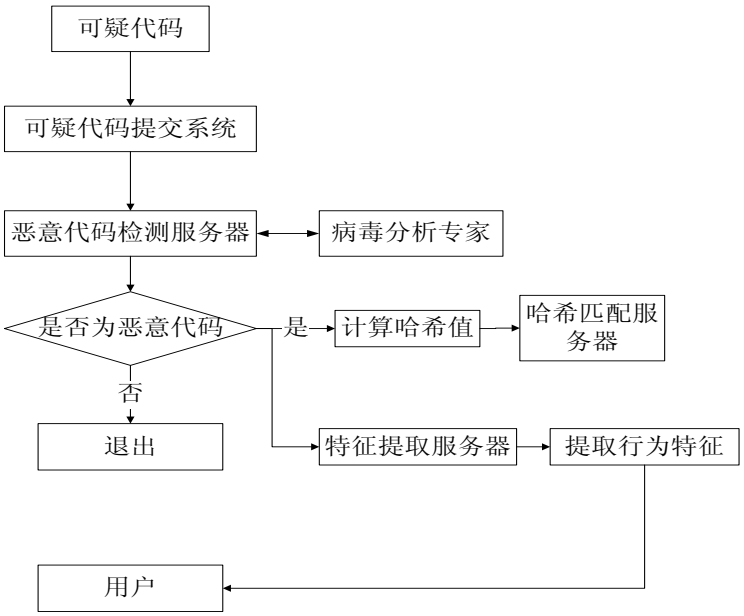

图 6-3 服务器端检测流程图

6.2 恶意代码监测系统测试

系统做了查杀率和误报率两方面的测试,并于国内外的一些知名杀毒软件测试结果做对比。

6.2.1 查杀率测试

系统采用 2011 年 11 月 5 日至 2011 年 11 月 12 日,由 360 杀毒引擎上报的 2813 个病毒样本，作为系统查杀率测试数据。

图 6-4 查杀率测试样本

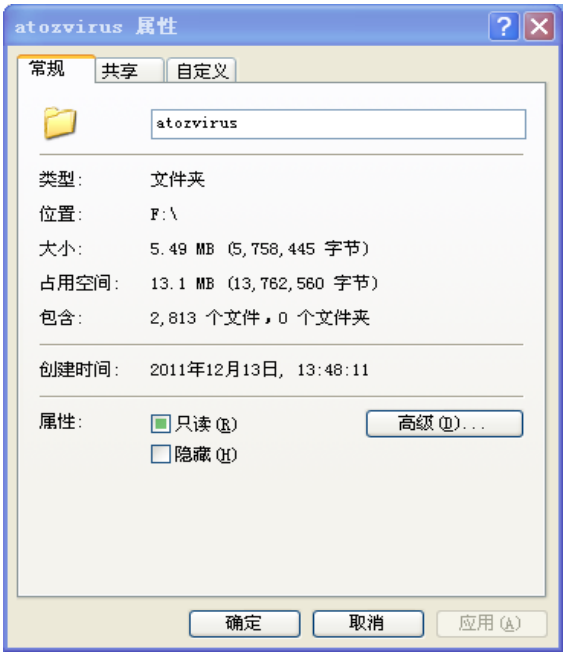

图 6-5 查杀率测试样本基本信息

图 6-6 查杀率测试 360 测试结果图

图 6-7 查杀率测试小红伞测试结果图

图 6-8 查杀率测试金山毒霸测试结果图

图 6-9 查杀率测试瑞星测试结果图

图 6-10 查杀率测试 AVG 测试结果图

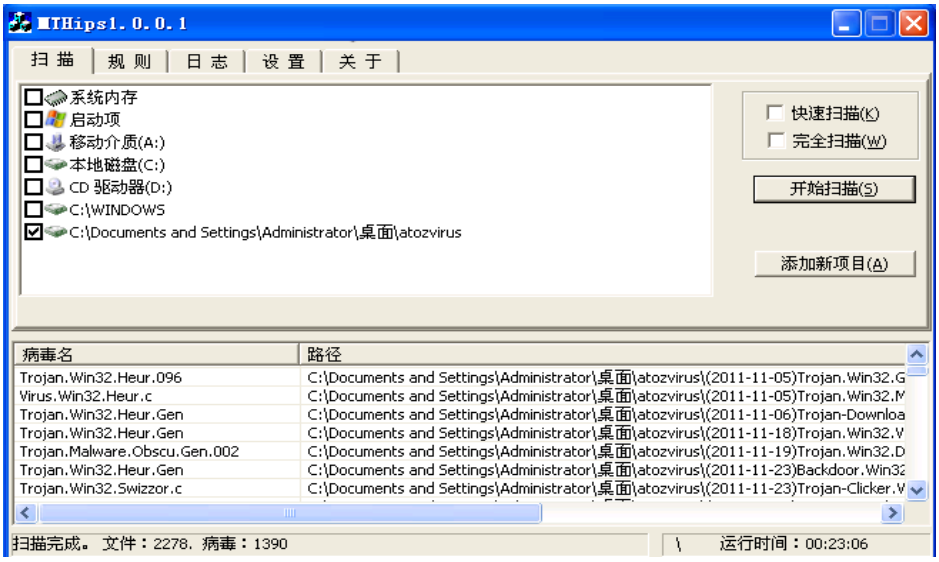

图 6-11 查杀率测试原型系统测试结果图

表 6-1 查杀率对比

系统名称	待检测本数	实际检测样本数	恶意代码数量	查杀率	耗时
360	2813	2813	2792	99.25%	00:20:00
小红伞	2813	2818	2796	99.21%	00:11:00
金山毒霸	2813	2813	2	0.07%	00:00:04
瑞星	2813	2817	4	0.14%	00:00:02
AVG	2813	2814	2515	89.37%	00:00:25
原型系统	2813	2278	1390	61.02%	00:23:06

由表 6-1 可知，当前的杀毒软件在掌握了最新样本的二进制特征码的情况下，查杀率很高，反之查杀率相当的低。而原型系统的没有样本的最新二进制特征码的情况下依然能保证相对较高的查杀率。

6.2.2 误报率测试

系统采用新装系统的系统文件夹 windows 目录下的 17260 个文件作为测试样本。

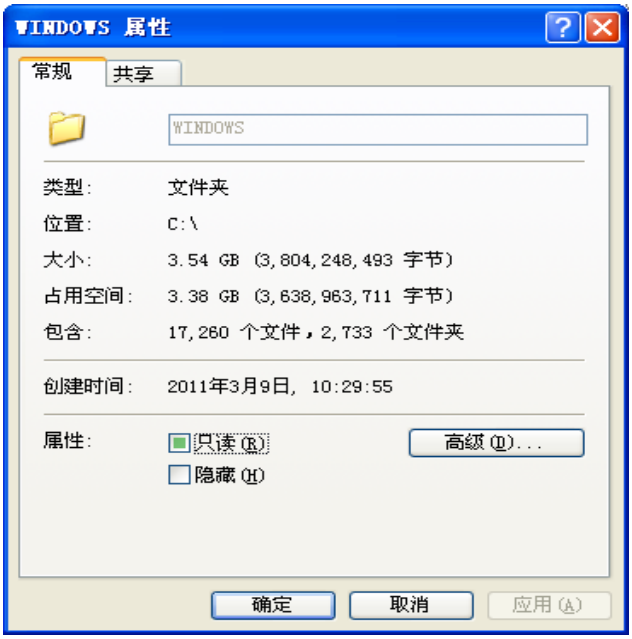

图 6-12 误报率测试样本基本信息

图 6-13 误报率测试 360 测试结果图

图 6-14 误报率测试小红伞测试结果图

图 6-15 误报率测试金山毒霸测试结果图

图 6-16 误报率测试瑞星测试结果图

图 6-17 误报率测试瑞星测试结果图

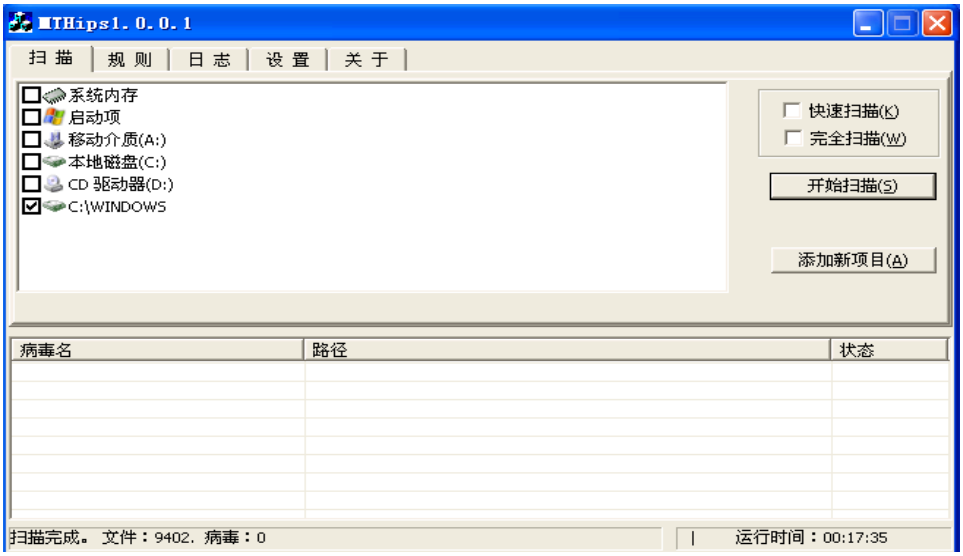

图 6-18 误报率测试原型系统测试结果图

表 6-2 误报率对比

系统名称	待检测样本数	实际检测样本数	恶意代码数量	误报率	耗时
360	17260	76404	0	0%	00:29:00
小红伞	17260	38796	0	0%	00:05:46
金山毒霸	17260	39123	0	0%	00:10:40
瑞星	17260	17015	0	0%	00:03:37
AVG	17260	15691	0	0%	00:07:15
原型系统	17260	9402	0	0%	00:17:35

由表 6-2 可知，原型系统的误报率方面达到预期效果。

6.3 本章小结

本章主要完成对检测原型系统的设计实现和测试。在根据前面研究的模式匹配、虚拟机技术、启发式检测技术和高性能服务器技术的基础上，设计了高性能、稳定、可靠的基于云技术的恶意代码检测系统。之后对系统的查杀率和误报率的两方面问题进行了测试，并与国内外的一些著名杀毒软件测试的结果进行了对比，基本达到预期效果。

第七章 总结与展望

7.1 总结

本文在介绍传统的恶意代码检测技术，指出了其不足。由于该方法必须要得到恶意代码的二进制特征代码，一旦恶意代码通过加密等方式改变了其二进制特征代码，该方法将彻底失效。传统的杀毒软件面对当前的恶意代码已经显得力不从心。

在阐述模式匹配算法、启发式扫描算法以及虚拟机技术等的基本理论及关键技术的基础上，研究了目前恶意代码检测引擎的模式匹配算法的不足，提出改进的方案。进而提出了一个结合二进制特征匹配、行为特征匹配以及云端检测的新型恶意代码检测方法的方案。

结合理论基础和实际操作，设计实现了原型系统，并通过与目前国内外的几款比较著名的杀毒软件系统进行了对比测试，基本满足预期效果。

7.2 展望

入侵检测技术一直是一个热点研究领域，随着恶意代码的发展，入侵检测技术也在不断的发展。传统的简单的基于二进制特征码匹配的检测方法已经无法满足现行的要求。本课题的研究为相关领域的进一步研究奠定了良好的基础。在接下来的进一步工作计划中，将会在以下几个方面做出进一步的努力来完成本课题：

(1) 本课题目前实现的多模式匹配 AW 算法是对 AC 和 WM 算法的改进，其主要目的是为了找到一种能适合万以上级别的模式匹配算法。AW 算法基本已经达到了预期目的，但仍有不足之处，比如其时间复杂度要劣于 AC 和 WM 算法，在实际测试中发现耗时要比 AC 和 WM 算法的大。下一步计划在对其进行优化，进一步降低其时间复杂度。

(2) 本课题通过研究 QEMU 和 BOCHS 两款虚拟机产品，详细的研究了虚拟机实现的机制，研究了代码二进制翻译技术，研究了代码静态翻译和动态翻译技术。在原型系统的实现中，采用了 QEMU 的二进制代码动态翻译技术，实际证明速度要优于代码静态

翻译技术。但让没有达到想 VMWare、VirtualPC 和 VirtualBox 等基于硬件模拟的虚拟机的效率。下一步打算结合 VirtualBox 源码，对原型系统的虚拟机组件进行优化。

致 谢

随着毕业设计接近尾声，我的大学时光也即将结束，回首研究生生活，是充实的三年，在这里，我汲取了大量的专业知识，增强了自身素质，各方面的能力都得到了不同程度的锻炼。真诚的感谢给予过我关心和照顾的每一个人，没有他们，就不可能有我今天所取得的成绩。

首先，我要感谢我的父母多年来的养育之恩，无论何时何地，无论成功失败，他们都给我无私的关怀与爱心，正因为他们，我才拥有了今天。

其次，我衷心地感谢我的指导老师王汝传老师对我的关怀和支持。在毕业设计的过程中，我始终得到王老师悉心的指导和热情的鼓励。王老师认真负责的工作态度、严谨的治学风格，使我深受启发，这不仅使我得以完成毕业设计，而且我相信在今后的学习和工作中也将有深远的影响，使我受用终身。

在此我特别感谢李致远博士。感谢实验室同学给我的指导和帮助，给我提供良好的开发环境，没有他们的帮助和支持，我的毕业设计及论文将无法完成。我也要感谢我身边给我支持的每一个人，从他们身上，我学会了很多东西。他们的经验和智慧让我克服了很多困难也让我少走了许多弯路。

最后，特别感谢审阅本论文的教授、专家们，感谢他们在百忙之中抽出时间阅读全文，以及他们宝贵的建议和意见。

攻读硕士学位期间的学术论文

一、已发表的学术论文

[1]雷迟骏，王汝传，李致远. 基于虚拟技术启发式杀毒引擎的研究. 《全国计算机新科技与计算机教育论文集（第 19 卷）》，2011.

攻读硕士学位期间参加的科研项目

[1]南京高技术项目，电信网中 P2P 流量检测与控制软件(2007 软资 127).

[2]国家 863 项目，新型对等计算网络安全关键技术(2006AA01Z439).

[3]华为赛门提克, ARES 协议分析系统(YBSS2009401).

缩略词

缩略词	英文全称	译文
AC	Aho-Corasick	Aho-Corasick 算法
AGP	Accelerated Graphic Ports	快速图像接口
API	Application Programming Interface	应用程序编程接口
CISC	Complex Instruction Set Computer	复杂指令系统计算机
CPU	Central Processing Unit	中央处理器
FPU	Float Point Unit	浮点运算单元
GDT	Global Descriptor Table	全局描述符表
GDTR	Global Descriptor Table Register	全局描述寄存器
GMCH	Graphics & Memory controller hub	北桥
IDE	Integrated Drive Electronics	电子集成驱动器
IDT	Interrupt Descriptor Table	中断描述符表
I/O	input/output	输入输出端口
ISA	Industrial Standard Architecture	ISA 总线
KMP	Knuth-Morris-Pratt Algorithm	KMP 算法
LAN	Local Area Network	局域网
LDT	Local Descriptor Table	局部描述符表
LDTR	Local Descriptor Table Register	局部描述寄存器
LPC	Low Pin Count	LPC 接口
MMU	Memory Management Unit	内存管理单元
PC	personal computer	个人计算机
PCI	Pedpherd Component Interconnect	周边元件扩展接口
PE	Portable Executable	可执行文件
RISC	reduced instruction set computer	精简指令集
ROM	Read-Only Memory	只读内存

SDRAM	**Synchronous Dynamic Random Access Memory**	**同步动态随机存储器**
TCG	**Tiny Code Generator**	**内存管理单元**
USB	**Universal Serial BUS**	**通用串行总线**
VB	**Visual Basic**	**Visual Basic 编程语言**
VMM	**VMM Virtual Machine Monitor**	**虚拟机监视器**
WM	**Wu-Manber**	**Wu-Manber 算法**

图表清单

参考文献

[1] S. Shanbhag, T. Wolf, Accurate anomaly detection through parallelism, IEEE Network Special Issue on Recent Developments in Network Intrusion Detection 23 (1) (2009) 22-29.

[2] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, S. Vassiliadis, Packet pre-filtering for network intrusion detection, in: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, 2006, pp. 183-192.

[2] S. Acharya, B.N. Mills, M. Abliz, T. Znati, J. Wang, Z. Ge, A.G.Greenberg, Optwall: a hierarchical traffic-aware firewall, in: NDSS,2007, pp. 528-533.

[3] 边肇祺, 阎平凡, 杨存荣. 模式识别(第一版)[M].北京:清华大学出版社,1988.

[4] Bilke A,Nauman F.Schema matching using duplicates. In:Kitagawa H,Ishikawa,eds.Proc.of the 18th Int'l Conf.on Data Engineering. Los Alarnitos:IEEE ComputerSociety,2005.69-80.

[5] S.-G.M. Hamid Sarbazi-Azad, Behrooz Parhami, S. Hessabi, A multi- Gb/s parallel string matching engine for intrusion detection systems, in: Advances in Computer Science and Engineering, vol. 6, 2008, pp.847-851.

[6] 陈慧南，数据结构—使用 C++语言描述[M].人民邮电出版社，2006.

[7] 李伟男, 鄂跃鹏, 葛敬国, 钱华林. 多模式匹配算法及硬件实现[J]. 软件学报.2006,12 期，2403-2411

[8] 孙钦东, 黄新波, 王倩. 面向中英文混合环境的多模式匹配算法[J]. 软件学报. 2008,3 期，674-686.

[9] 董迎亮, 玄雪花, 王德民. 基于 WM 算法改进的多模式匹配算法

[10] N. Hua, H. Song, T. Lakshman, Variable-stride multi-pattern matching for scalable deep packet inspection, in: INFOCOM 2009,IEEE, 2009, pp. 415-423.

[11] L. Vespa, N. Weng, B. Soewito, Optimized memory based accelerator for scalable pattern matching, Microprocessesors and Microsystems 33 (7-8) (2009) 469-482.

[12] J. Hrisson, G. Payen, R. Gherbi, A 3d pattern matching algorithm for dna sequences, Bioinformatics 23 (6) (2007) 680-686.

[13] 刘真. 虚拟机技术的复兴[J]. 计算机工程与科学. 2008, 2 期, 105-109.

[14] 叶海波.SMART-VMM: 基于 VT-x 的虚拟机监控器设计与实现[D].浙江大学.

[15] 孟江涛.Xen 虚拟机研究[D].上海:上海交通大学.

[16] 时卫东. 基于内核的虚拟机的研究[D].吉林大学.

[17] 邵时. 微机接口技术[M]. 清华大学出版社, 2000.

[18] 居晓波, 李志斌, 宁兆熙, 程君侠, 王永流. 一种新型 CISC 微处理器指令译码设计方法 [J]. 微电子学报. 2003, 2 期,154-156.

[19] 董渊, 任恺, 王生原, 张素琴. 字节码虚拟机的构造和验证[J]. 软件学报, 2010, 02 期.

[20] 张昊. 基于虚拟机扩展的软件调试技术研究 [D].浙江大学.

[21] 李杰聪. 可移植动态翻译技术研究和实现 [D].浙江大学.

[22] 程芳. 轻量级操作系统内核研究 [D].重庆大学.

[23] 吴浩.二进制翻译系统 QEMU 的优化技术 [D].上海交通大学.

[24] 刘安战. 二进制翻译中自修改代码的缓存策略研究 [D].华中科技大学.

[25] 柏志文. 基于动态二进制翻译的污点检测设计和实现 [D]. 西安电子科技大学.

[26] 刘涛. 基于动态二进制翻译的逆向调试器的设计与实现 [D]. 西安电子科技大学.

[27] Tang Jiafu, Pan Zhendong, Gong Jun, Liu Shixin Combined heuristics for determining order quantity under time-varying demands , Journal of Systems Engineering and Electronics, Volume 19, Issue 1, February 2008, Pages 99-111

[28] J. Caballero, Z. Liang, P. Poosankam, D. Song, Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration, in: RAID’ 09, 2009, pp. 161-181.

[29] D. Tao, H. Ma. Coverage-Enhancing algorithm for directional sensor networks [C]. Proc. of the 2nd Int’l Conf. on Mobile Ad-Hoc and Sensor Networks. Berlin: Springer-Verlag, 2006, 256-267.

[30] 苏璞睿, 杨轶. 基于可执行文件静态分析的入侵检测模型. 2006，9 期，1570—1576

[31] 陈 友，沈华伟，李 洋，程学旗. 一种高效的面向轻量级入侵检测系统的特征选择算法. 计算机学报 2007，8 期，1398-1408.

[32] 田新广，段淼毅，程学旗. 基于 shell 命令和多重行为模式挖掘的用户伪装攻击检测. 计算机学报 2010，4 期，697-705.

[33] 沈昌祥. 可信计算平台与安全操作系统[J]. 网络安全技术与应用. 2005, 4.5(4):P8~9.

[34] 王斌，谢小权. 可信计算机 BIOS 系统安全模块的设计[J]. 计算机安全.2006 09. 35-40.

[35] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of TCG-based integrity measurement architecture. In Proceedings of the 13th USENIX Security Symposium, 2004, P223~238.

[36] 王军选, 田小平, 曹红梅. *信息论*基础与编码[M]. 人民邮电出版社, 2011.

[37] G. Wei, X. Zhou, and H. Zhang, “A trusted computing model based on code authorization,” in ISIP ’08: Proceedings of the 2008 International Symposiums on Information Processing.

Washington, DC, USA: IEEE Computer Society, 2008, P495~499.

[38] Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework[C]. Proceedings of ISSTA. 2007, P191~203.

[39] 刘鹏. 云计算(第一版)[M]. 电子工业出版社. 2010.

[40] 任泰明. TCP/IP 网络编程[M]. 人民邮电出版社, 2009.

[41] 恽如伟, 董浩. 网络游戏编程教程[M]. 机械工业出版社, 2009.

[42] 张晓明. 计算机网络编程技术[M]. 中国铁道出版社, 2009.