

2016 年全国大学生信息安全竞赛作品报告

作品名称：基于图像处理的恶意代码检测

组 员：秦梦军 高爽 陆嘉旻

提交日期：2016/5/30

填写说明

1. 所有参赛项目必须为一个基本完整的设计。作品报告书旨在能够清晰准确地阐述（或图示）该参赛队的参赛项目（或方案）。
2. 作品报告采用A4纸撰写。除标题外，所有内容必需为宋体、小四号字、1.5倍行距。
3. 作品报告中各项目说明文字部分仅供参考，作品报告书撰写完毕后，请删除所有说明文字。（本页不删除）
4. 作品报告模板里已经列的内容仅供参考，作者可以在此基础上增加内容或对文档结构进行微调。
5. 为保证网评的公平、公正，作品报告中应避免出现作者所在学校、院系和指导教师等泄露身份的信息。

摘 要

互联网的开放性给人们带来便利的同时也加快了恶意代码的传播，特别是人们可以直接从网站获得恶意代码源码或通过网络交流代码，很多编程爱好者把自己编写的恶意代码放在网上公开讨论，发布自己的研究成果，直接推动了恶意代码编写技术发展。出于经济政治利益驱动和各种新技术的使用，恶意代码的数量呈指数级增长，同时各种恶意代码变种层出不穷，给生产和生活造成了巨大的威胁。

恶意代码(malicious code)通过在不被察觉的情况下用户计算机或其他终端上的安装运行，达到破坏被感染电脑数据、运行具有入侵性或破坏性的程序、破坏被感染电脑数据的安全性和完整性的目的。恶意软件的传染的结果包括浪费资源、破坏系统、破坏一致性，数据丢失和被窃并能让客户端的用户失去信心。

恶意代码检测分为静态和动态行为检测两种，传统的静态检测技术主要为特征码识别，然而这种方法对恶意代码的变种检测效率很低，动态行为检测又会造成高昂的系统运行负担。因此开发一种能高效识别检测变种的恶意代码检测工具具有现实意义。基于此，我们团队提出了基于图像处理的恶意代码检测方法，即将处理成二进制灰度图像的未知代码与恶意代码二进制灰度图像库进行匹配，从而判断未知代码是否为恶意代码。处理过程分为反汇编、预处理、生成图像和图像匹配四步。这种方法既保留了传统静态检测的高效性，又摒除了其对变种识别率低的弊端。本作品的创新性主要体现在：（1）预处理阶段的去混淆技术，还原经过混淆处理的目标汇编代码，这是目前市面上的静态检测恶意代码的产品所没有的。（2）基于图像恶意代码检测，利用图像中的纹理特征对恶意代码进行聚类这一新概念最早于 2011 年由加利福尼亚大学的 Nataraj 和 Karthikeyan 所提出，我们以此为出发点融入多种技术将其运用到实践中。

我们成功利用图像处理技术检测出了目前几大流行病毒的变种，以 DCM 病毒为例，还比较和分析了运用去除混淆技术前后的效果，结果显示去除混淆后病毒变种匹配度整体提高了约一倍（详见测试报告），这是图像处理技术和去除混淆技术运用于恶意代码检测有效性和可行性的有力证明。

关键词：反汇编 去混淆 灰度图 图像匹配 静态检测

目 录

摘 要	3
目 录	4
第一章 作品概述.....	6
1.1 背景分析	6
1.2 相关工作	6
1.3 功能描述	6
1.3.1 反汇编.....	7
1.3.2 去除混淆代码.....	7
1.3.3 代码转二进制灰度图.....	7
1.3.4 图像匹配.....	7
1.4 可行性分析	7
1.4.1 技术可行性分析.....	7
第二章 实现方案.....	9
2.1 反汇编	9
2.1.1 可执行文件结构.....	9
2.1.2 相关数据结构.....	10
2.1.3 机器指令转化为汇编代码.....	12
2.2 去除混淆代码	12
2.2.1 混淆技术.....	12
2.2.2 去混淆算法.....	13
2.3 灰度图像转化	14
2.4 图像匹配	14
2.4.1 surf 图像匹配算法.....	14
2.4.2 对二进制灰度图像进行 surf 特征点匹配.....	20
第三章 系统测试.....	23
3.1 引言	23
3.1.1 项目简介.....	23

3.1.2 测试目的.....	23
3.1.3 项目模块结构图.....	23
3.1.4 开发及测试环境.....	24
3.2 功能测试	25
3.2.1 反汇编.....	25
3.2.2 去混淆.....	30
3.2.3 汇编代码转换为图像.....	31
3.2.4 图像匹配.....	31
第四章 创新性	34
4.1 对病毒程序的去混淆预处理.....	34
4.1.1 混淆代码的干扰问题.....	34
4.1.2 去混淆的作用以及对项目的影响.....	34
4.2 基于深度优先搜索的去混淆技术.....	34
4.2.1 深度优先搜索算法.....	34
4.2.2 几种混淆类型的去除.....	35
4.3 基于二进制灰度图像纹理的恶意代码聚类.....	35
4.4 基于 surf 特征点匹配算法的图像特征分析.....	36
第五章 总结	37
5.1 本系统的设计与开发.....	37
5.2 作品展望.....	37
参考文献	39

第一章 作品概述

1.1 背景分析

随着社会的不断进步，计算机和网络已经应用到了人类生活的各个层面，网络安全问题日益得到人们的重视。恶意代码是当前网络安全的主要威胁之一，出于经济利益驱动和各种新技术的使用，恶意代码的数量呈指数级增长，同时各种恶意代码变种层出不穷，导致安全威胁事件逐年上升。

2010 年布什尔核电站遭到“震网”病毒攻击，1/5 的离心机报废；2012 年火焰病毒入侵伊朗能源部门，窃取了大规模机密资料；2015 年引起极大关注的 xcodeghost 事件；就在我们着手准备这个项目期间，德国 Gundremmingen 核电站的计算机系统，在常规安全检测中发现了恶意程序，被迫关闭了发电厂。这一系列骇人听闻的恶性事件的元凶都直指恶意代码。

为抵制恶意代码贡献自己的一份力量，我们团队分析总结了恶意代码检测技术现状与存在的不足，提出了基于图像处理检测恶意代码的新思路。

1.2 相关工作

通过对国内外现有恶意代码检测技术和成果进行分析和研究，总结出主要的静态分析方法有基于特征码检测、基于语义检测、启发式扫描检测，动态分析方法主要有系统监控法、动态跟踪法。而这些方法仍存在以下不足：

1、静态检测方法中特征码检测对变种检测具有滞后性，基于语义检测需要大量人工操作，启发式扫描虽能提高变种检测的准确率但仍依赖于特征码。

2、恶意代码为了躲避检测常常使用了混淆手段躲避查杀软件的检测，而目前还没有令人满意的去除混淆的方法。

3、动态检测方法近年来发展迅速，各种杀毒软件也普遍采用这种方法，但系统运行负荷也相对较高，拖慢系统的整体运行效率。

针对这些不足，我们开发了基于图像处理的恶意代码检测系统。

1.3 功能描述

1.3.1 反汇编

把要检测的目标机器码转为汇编代码。

1.3.2 去除混淆代码

还原经过混淆处理的汇编代码，这里主要处理了两种形式的混淆：

1) 嵌入的无关代码

2) 跳转（jmp 指令）

1.3.3 代码转二进制灰度图

以二进制的形式读取代码输出为二进制灰度图像

1.3.4 图像匹配

运用图像匹配 surf 算法，将待检测代码灰度图像与病毒库代码灰度图像匹配。

1.4 可行性分析

1.4.1 技术可行性分析

●同种族恶意代码的二进制灰度图像纹理相似性

加利福尼亚大学 L. Nataraj 及其研究团队将恶意代码的二进制可执行文件可视化作为灰度图像并进行对比，发现相同族群的恶意代码的灰度图像的结构和布局非常相似。在此基础上，他们提出了通过提取恶意代码灰度图像的纹理特征，并根据这些特征对恶意代码进行分类。此后，许多学者对此方法进行了进一步的研究和分析。V. Yegneswaran 及其研究团队通过将该方法与其他现存的恶意代码分类方法进行对比，发现基于纹理特征的恶意代码分类方法不但在准确率上比其他恶意代码分类方法更精确，在速度上也有非常大的提高。

●图像匹配 surf 算法

Surf 算法是 sift 算法的改进，效率更高。

特征点的提取：Hessian 矩阵是 surf 算法的核心，利用 Hessian 矩阵，计算特征值 α

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix}$$

其中 $L_{xx}(\mathbf{x}, \sigma)$ 是高斯滤波后图像 $g(\sigma)$ 的在 x 方向的二阶导数，其他的 $L_{yy}(\mathbf{x}, \sigma)$ 、 $L_{xy}(\mathbf{x}, \sigma)$ 都是 $g(\sigma)$ 的二阶导数。根据是否为领域极大值判断特征点。

特征点的匹配：欲进行特征点的匹配，必须提取出特征点的特征向量再利用两个

向量的相似程度认为两个点是否为两幅图像相互对应的点。然后提取特征描述符，特征点的匹配，采用最简单的两向量内积最大值为最匹配的点，设定一阈值，只有当这个最大值大于该阈值方可认为两特征点匹配。

1.4.2 市场可行性分析

在互联网逐渐参与进人们日常生活的同时，许多恶意代码也掺杂其中如果不加防范很容易“中招”。

而目前的检测软件很难做到高变种识别率和低系统运行效率两全，此系统在恶意代码检测准确性、变种检测识别率及检测效率方面都有突出的优势，在完善界面和病毒库之后或许能够广泛投入应用。

第二章 实现方案

2.1 反汇编

对可执行文件进行反汇编处理是我们所有工作的第一步，所以反汇编结果的好坏很大程度上决定了最终检测结果的效果。

2.1.1 可执行文件结构

由于我们检测的对象是可执行（PE， Portable Execute）程序，所以对于 PE 文件的结构必须有一定的了解才能进行后续的工作。

PE 文件是 windows 可执行文件的总称，常见文件类型有 EXE，DLL，SYS，COM，OCX，PE 文件的结构一般入下图所示，

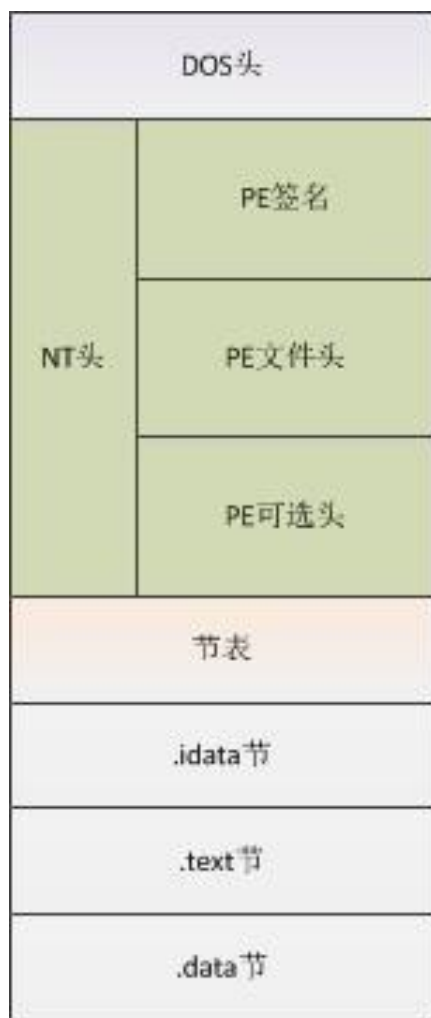


图 2-1 PE 文件的结构

下面我们将对其中的每一部分进行详细解释：

(1) DOS 头是用来兼容 MS-DOS 操作系统的，目的是当这个文件在 MS-DOS 上运行时提示一段文字，大部分情况下是：This program cannot be run in DOS mode. 还有一个目的，就是指明 NT 头在文件中的位置。

(2) NT 头包含 windows PE 文件的主要信息，其中包括一个‘PE’字样的签名，PE 文件头 (IMAGE_FILE_HEADER) 和 PE 可选头 (IMAGE_OPTIONAL_HEADER32)。

(3) 区段表：是 PE 文件后区段的描述，windows 根据区段表的描述加载每个区段。

(4) 区段：每个区段实际上是一个容器，可以包含代码、数据等等，每个区段可以有独立的内存权限，比如代码段默认有读/执行权限，区段的名称和数量可以自己定义，未必是上图中的三个。

当一个 PE 文件被加载到内存中以后，我们称之为“映象”(image)，一般来说，PE 文件在硬盘上和在内存在里是不完全一样的，被加载到内存以后其占用的虚拟地址空间要比在硬盘上占用的空间大一些，这是因为各个节在硬盘上是连续的，而在内存中是按页对齐的，所以加载到内存以后节之间会出现一些“空洞”。

因为存在这种对齐，所以在 PE 结构内部，表示某个位置的地址采用了两种方式，针对在硬盘上存储文件中的地址，称为原始存储地址或物理地址表示距离文件头的偏移；另外一种是针对加载到内存以后映象中的地址，称为相对虚拟地址 (RVA)，表示相对内存映象头的偏移。

然而 CPU 的某些指令是需要使用绝对地址的，比如取全局变量的地址，传递函数的地址编译以后的汇编指令中肯定需要用到绝对地址而不是相对映象头的偏移，因此 PE 文件会建议操作系统将其加载到某个内存地址（这个叫基地址），编译器便根据这个地址求出代码中一些全局变量和函数的地址，并将这些地址用到对应的指令中。

2.1.2 相关数据结构

首先是 DOS 头部的数据结构。

```

typedef struct _IMAGE_DOS_HEADER
{
    WORD    MagicNumber;           /* Magic Number must be "MZ" */
    WORD    BytesLPP;             /* Byte on last page of file */
    WORD    Pages;                /* Pages in file */
    WORD    Relocations;          /* Relocations */
    WORD    HeaderSize;           /* Size of header in paragraphs */
    WORD    MinParagraphs;        /* Minimum extra paragraphs needed */
    WORD    MaxParagraphs;        /* Maximum extra paragraphs needed */
    WORD    RegisterSS;           /* Initial (relative) SS value */
    WORD    RegisterSP;           /* Initial SP value */
    WORD    Checksum;             /* CheckSum */
    WORD    RegisterIP;           /* Initial IP value */
    WORD    RegisterCS;           /* Initial (relative) CS value */
    WORD    RelocationTable;      /* File address of relocation table */
    WORD    OverlayNumber;        /* Overlay Number */
    WORD    Reserved[4];          /* Reserved words */
    WORD    OEMIdentifier;        /* OEM identifier */
    WORD    OEMInformation;       /* OEM information */
    WORD    Reserved2[10];        /* Reserved words */
    WORD    PEHeader;             /* File Address of new exe header */
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

图 2-2 DOS 头数据结构

然后是 NT 头部信息。

```

typedef struct _IMAGE_NT_HEADERS
{
    DWORD    PESignature;         /* "PE\0\0" signature */
    IMAGE_FILE_HEADER FileHeader; /* File Header */
    IMAGE_OPTIONAL_HEADER OptionalHeader; /* Optional Header */
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;

```

图 2-3 NT 头部信息

最后是，可选头部信息。

```

typedef struct _IMAGE_OPTIONAL_HEADER
{
    WORD    Magic;                /* State of the image */
    BYTE    MajorLinkerVersion;   /* Major linker version */
    BYTE    MinorLinkerVersion;  /* Minor linker version */
    DWORD   SizeOfCode;           /* Size of code sections */
    DWORD   SizeOfInitializedData; /* Size of initialized data */
    DWORD   SizeOfUninitializedData; /* Uninitialized data size */
    DWORD   AddressOfEntryPoint;  /* RVA image execute from */
    DWORD   BaseOfCode;           /* RVA of code section */
    DWORD   BaseOfData;           /* RVA of data section */
    DWORD   ImageBase;            /* where image assumed to be loaded */
    DWORD   SectionAlignment;     /* Alignment of image in RAM */
    DWORD   FileAlignment;        /* Alignment of image in FILE */
    WORD    MajorOSVersion;       /* Major version of OS */
    WORD    MinorOSVersion;       /* Minor version of OS */
    WORD    MajorImageVersion;    /* Major version of Image */
    WORD    MinorImageVersion;    /* Minor version of Image */
    WORD    MajorSubsystemVersion; /* Major version of Subsystem */
    WORD    MinorSubsystemVersion; /* Minor version of Subsystem */
    DWORD   Reserved;            /* Reserved */
    DWORD   SizeOfImage;          /* Size of image form Imagebase */
    DWORD   SizeOfHeaders;        /* Size of PE Header and section tables */
    DWORD   CheckSum;             /* CRC checksum */
    WORD    Subsystem;            /* Which subsystem needed */
    WORD    DllCharacteristics;    /* When DllMain to be called */
    DWORD   SizeOfStackReserve;   /* Not all will be committed */
    DWORD   SizeOfStackCommit;    /* Stack committed */
    DWORD   SizeOfHeapReserve;    /* Not all will be committed */
    DWORD   SizeOfHeapCommit;     /* Heap committed */
    DWORD   LoaderFlags;          /* Debugging associated */
    DWORD   NumberOfRvaAndSizes;  /* Number of data directories */
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
};

```

图 2-4 可选头部信息

2.1.3 机器指令转化为汇编代码

由于 PE 文件的代码部分都在 .text 段，所以我们首先要做的就是找到 .text 段的基址，这种方法存在一个很大的缺点，就是如果源程序修改了默认的区段名字，比如将 .text 改为 upx 那么我们的程序将无法进行反汇编工作。不过由于时间的关系以及为了简化程序流程，并且我们的主要目标的对反汇编后的程序进行处理，所以没有对此进行深入研究。

得到代码段基址后，我们就可以逐字节进行扫描，并且按照 Intel 指令集将字节码翻译成对应的汇编指令，扫描的过程我们是递归进行的，所以耗时相对较长但是准确率却比较高。具体实验结果请参考测试部分。

2.2 去除混淆代码

2.2.1 混淆技术

由于目前学术上对于如何消除混淆代码极度缺乏关注，所以导致近几年几乎没有研究者对此，我们能够找到的最近的一篇论文还是国外研究者在 2006 年发表的，但是对于如何处理混淆也并没有给出详细解决方案，所以这部分工作对于我们本科生来说是极具挑战性的，也是我们整个项目的创新点之一。

首先，我们通过查找相关资料，了解到当前主流的针对汇编代码进行混淆的技术主要有以下几种：

- (1) 无用代码：无用代码是执行之后没有任何效果或者在代码生存期内始终都不会执行的代码。
- (2) 置换代码：置换代码的意思是通过打乱代码原有的执行顺序然后通过插入合适的跳转语句从而使得代码依然按照原始的顺序执行。
- (3) 指令替换：指令替换是指用相同功能但是不同的表现形式的代码来进行替换。
- (4) 寄存器替换：由于寄存器之间并无本质区别，所以如果某一段代码中没有使用某一个寄存器，例如 eax，那么我们就可以 eax 寄存器来替换这段代码中任意一个寄存器，并且仍然保持原有代码的功能。

原始代码	插入无用代码	置换代码	指令替换
call 0h pop ebx lea ecx, [ebx+42h] push ecx push eax push eax sidt [esp - 02h] pop ebx add ebx, 1Ch cli mov ebp, [ebx]	call 0h pop ebx lea ecx, [ebx+42h] nop (*) nop (*) push ecx push eax inc eax (**) push eax dec [esp - 0h] (**) dec eax (**) sidt [esp - 02h] pop ebx add ebx, 1Ch cli mov ebp, [ebx]	call 0h pop ebx jmp S2 S3: push eax push eax sidt [esp - 02h] jmp S4 add ebx, 1Ch jmp S6 S2: lea ecx, [ebx+42h] push ecx jmp S3 S4: pop ebx cli jmp S5 S5: mov ebp, [ebx]	call 0h pop ebx lea ecx, [ebx+42h] sub esp, 03h sidt [esp - 02h] add [esp], 1Ch mov ebx, [esp] inc esp cli mov ebp, [ebx]

图 2-5 混淆代码示例

如上图所示，最左边为原始代码，后面依次是几种混淆之后的代码。

2.2.2 去混淆算法

由于目前针对这些混淆技术都没有一个可行的解决方法，所以我们在实现的过程中自己设计了一种方法，是一种基于深度优先搜索的思想，在介绍具体算法之前有一个基本的定义。

基本块：我们定义一个基本块是一段连续的汇编代码，并且在整段中有且仅有一个改变程序执行流程的语句且位于段的最后一行。

我们具体对混淆代码进行处理的方法是，首先从第一行开始扫描，处理出所有的基本块，并且记录程序执行时入口函数的地址所在的块，以及每个基本块之间的连接关系，这些连接关系可以通过跳转语句的目标地址进行建立，最后所有的连接关系就可以构成一张有向图。然后我们就可以从程序入口点开始进行深度优先搜索，在搜索过程中标记出所有访问过的块，确保每个块仅被访问一次，并且记录下访问的顺序，搜索过程完成后，按照记录的顺序重新排列基本块作为最终的处理结果。

2.3 灰度图像转化

这部分的工作相对简单，实现的方法就是直接以二进制形式读取源文件，然后直接写入到图像中即可，由于 python 的库文件比较丰富，而且操作简洁，所以我们选择使用 python 进行这部分工作。具体实现时只需要调用 numpy 中的 `numpy.array` 函数就可以把字节数组转化为相应的矩阵，然后使用 `Image.save` 就可以保存为图片。详细实现细节请查看源代码中的 `asm2img.py` 文件，下图展示了一个转化后的灰度图像。

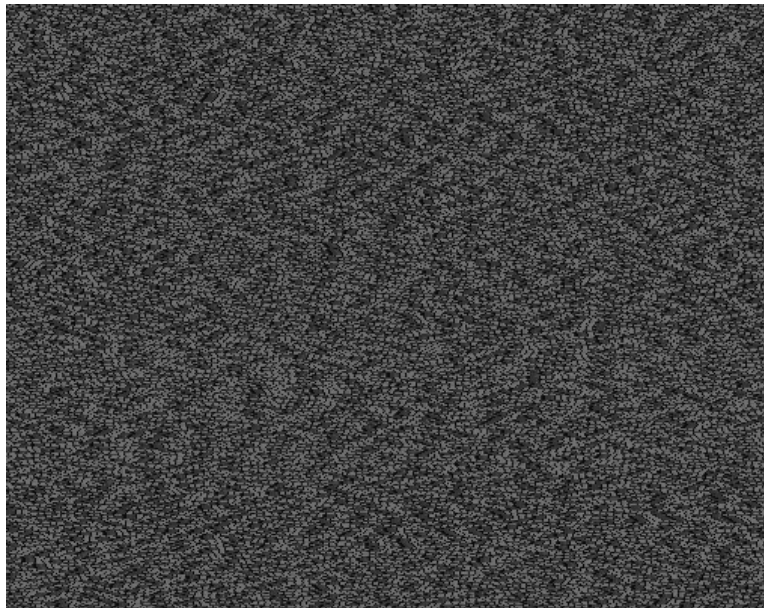


图 2-6 灰度图像

2.4 图像匹配

2.4.1 surf 图像匹配算法

surf 指的是加速的具有鲁棒性的特征，由 Bay 在 2006 年首次提出，这项技术可以应用于计算机视觉的物体识别以及 3D 重构中。surf 算子是由 sift 算子改进而来。一般来说，标准的 surf 算子比 sift 算子快好几倍，并且在多幅图片下具有更好的鲁棒性。surf 最大的特征在于采用了 Harr 特征以及积分图像 `integral image` 的概念，这样使得程序的运行速度得到了很大的提升。

2.4.1.1 形成 surf 特征描述子

(1) 构造高斯金字塔尺度空间

其实 surf 构造的金字塔图像与 sift 有很大不同，就是因为这些不同才加快了其检测的速度。Sift 采用的是 DOG 图像，而 surf 采用的是 Hessian 矩阵行列式近似值图像。某个像素点的 Hessian 矩阵，如下：

$$H(f(x, y)) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$$

即每一个像素点都可以求出一个 Hessian 矩阵。但是由于特征点需要具备尺度无关性，所以在进行 Hessian 矩阵构造前，需要对其进行高斯滤波。经过滤波后在进行 Hessian 的计算，其公式如下：

$$H(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix},$$

最终需要原图像的一个变换图像，因为我们要在这个变换图像上寻找特征点，然后将其位置反映射到原图中，是由原图每个像素的 Hessian 矩阵行列式的近似值构成的。其行列式近似公式如下：

$$\det(H_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2$$

由于求 Hessian 时要先高斯平滑，然后求二阶导数，这在离散的像素点是用模板卷积形成的，这两种操作合在一起用一个模板代替就可以了，比如说 y 方向上的模板如下：

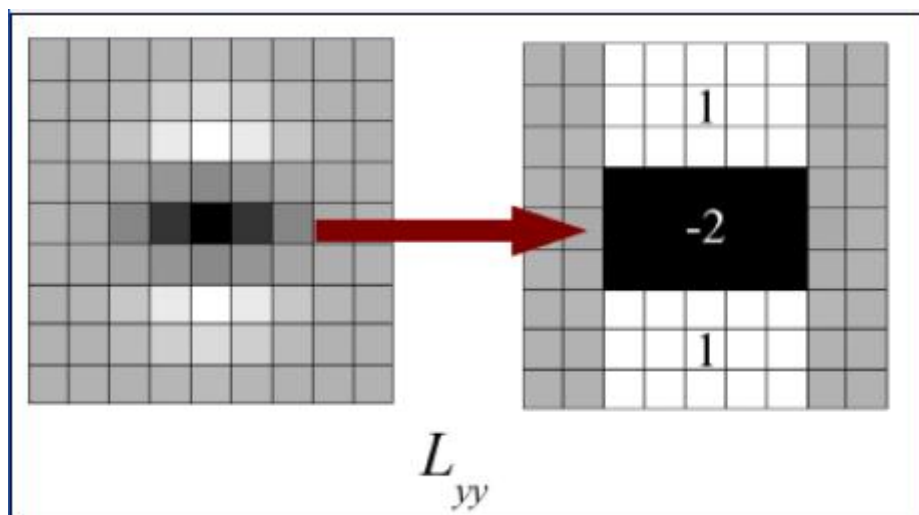


图2-7 y方向上的模版

该图的左边即用高斯平滑然后在 y 方向上求二阶导数的模板，为了加快运算用了近似处理，其处理结果如右图所示，这样就简化了很多。并且右图可以采用积分图来运算，大大的加快了速度。同理， x 和 y 方向的二阶混合偏导模板如下所示：

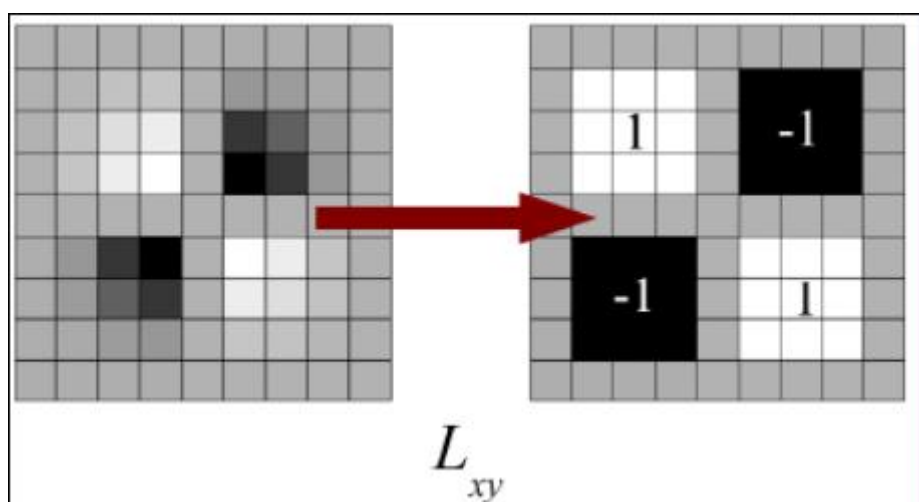


图2-8 x 和 y 方向的二阶混合偏导模板

最后得到了一张近似hessian行列式图。在surf中，图片的大小是一直不变的，不同的octave层得到的待检测图片是改变高斯模糊尺寸大小得到的，当然了，同一个octave中个的图片用到的高斯模板尺度也不同。Surf采用这种方法节省了降采样过其处理速度自然也就提上去了。其金字塔图像如下所示：

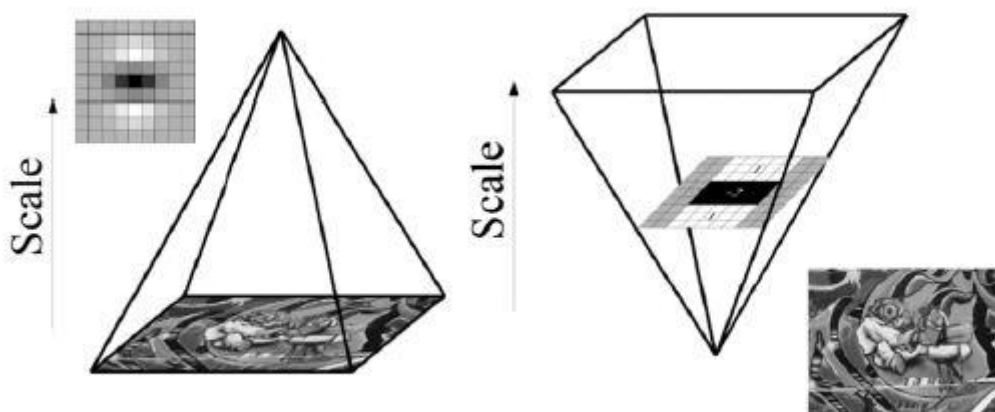


图2-9 surf图像金字塔

(2) 利用非极大值抑制初步确定特征点

将经过hessian矩阵处理过的每个像素点与其3维领域的26个点进行大小比较，如果它是这26个点中的最大值或者最小值，则保留下来，当做初步的特征点。

(3) 精确定位极值点

采用3维线性插值法得到亚像素级的特征点，同时也去掉那些值小于一定阈值的点。

(4) 选取特征点的主方向

统计特征点领域内的harr小波特征。即在特征点的领域(比如说，半径为 $6s$ 的圆内， s 为该点所在的尺度)内，统计 60° 扇形内所有点的水平haar小波特征和垂直haar小波特征总和，haar小波的尺寸变长为 $4s$ ，这样一个扇形得到了一个值。然后 60° 扇形以一定间隔进行旋转，最后将最大值那个扇形的方向作为该特征点的主方向。该过程的示意图如下：

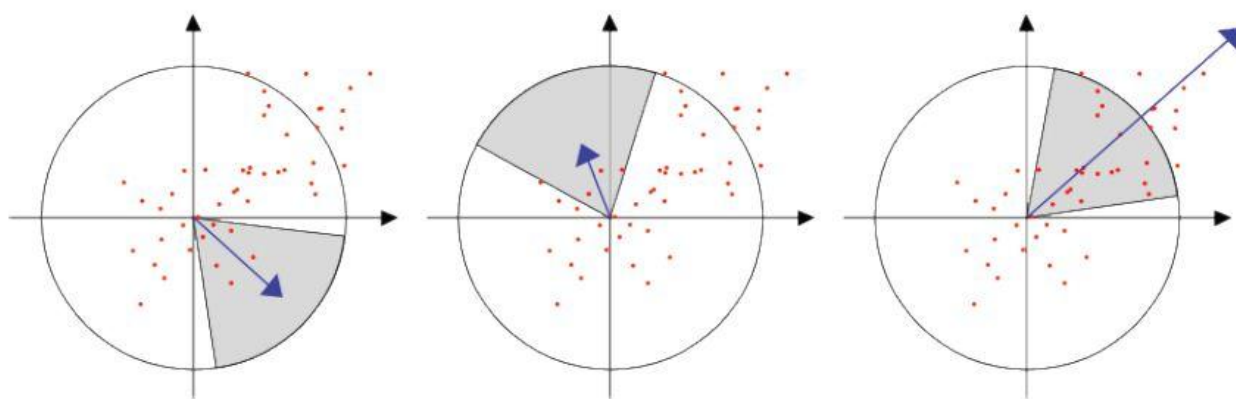


图2-10 确定特征点主方向

(5) 构造surf特征点描述算子

在特征点周围取一个正方形框，框的边长为 $20s$ (s 是所检测到该特征点所在的尺度)。该框带方向，方向当然就是第4步检测出来的主方向了。然后把该框分为16个子区域，每个子区域统计25个像素的水平方向和垂直方向的haar小波特征，这里的水平和垂直方向都是相对主方向而言的。该haar小波特征为水平方向值之和，水平方向绝对值之和，垂直方向之和，垂直方向绝对值之和。该过程的示意图如下所示：

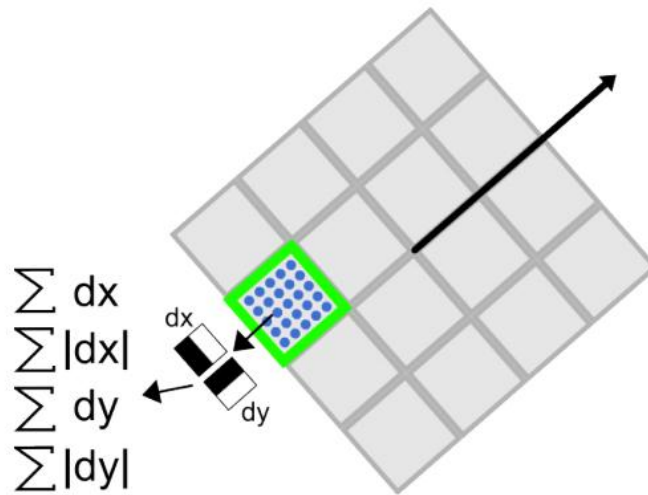


图2-11 构造特征点描述算子

这样每个小区域就有4个值，所以每个特征点就是 $16 \times 4 = 64$ 维的向量，相比sift而言，少了一半，这在特征匹配过程中会大大加快匹配速度。

2.4.1.2 特征点的匹配

(1) 寻找特征向量

欲进行特征点的匹配，必须提取出特征点的特征向量再利用两个向量的相似程度认为两个点是否为两幅图像相互对应的点。

以特征点为圆心半径为6像素建立圆领域，计算得出里面有109个像素点。计算这些点的harr特征harrx和harry.

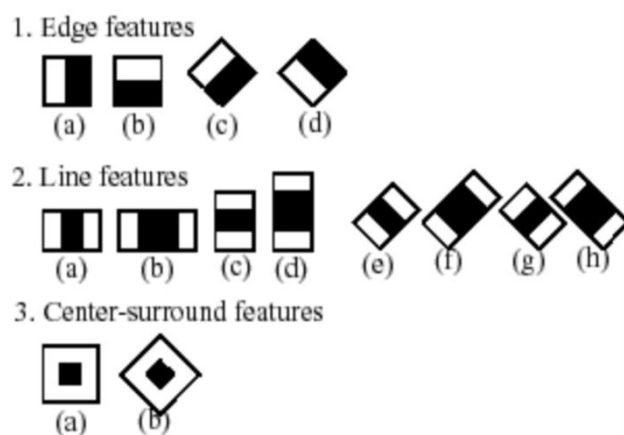


图 2-12 harr-like 特征

选取edge features前两个作为harrx和harry值,这个方向有些类似与梯度方向,不过这里的领域显然更广。至于计算么,依旧是利用积分图像。

对这109个像素点分别求出各自的向量的方向 $\text{angle} = \arctan(\text{harry}/\text{harrx})$, 根据最近邻原则将这些 angle划分到60, 120, ..., 300, 360等6个值上。划分在同一范围上的像素点分别将他们的harrx和harry相加即可。不过为了体现相邻像素点的更大影响,还需要考虑高斯权重系数。这样得到最大的harrx和最大的harry,组成了主方向向量。

(2) 提取特征描述符

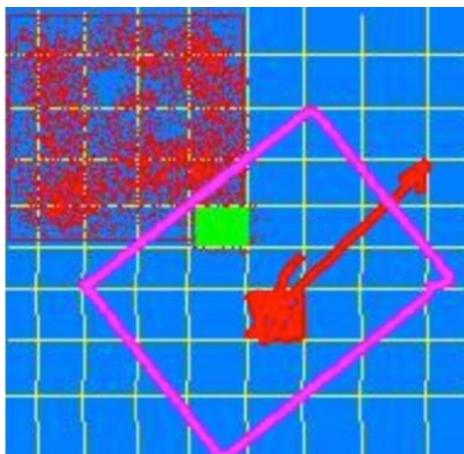


图 2-13 选取特征区域

图中红色箭头为上面计算出来的主方向,按上图所示选取该红色特征点的 8×8 邻域(紫色边框内部)计算得到 4×4 个像素块的梯度大小和方向(可以利用上面已经计算的harrx和harry),将 8×8 区域分割为 2×2 个区域T1, 2, 3, 4, 这样每个区域就包括了4个更小的由4个像素点组成的区域,



图 2-14 4 个更小的像素点组成的区域

harrx和harry就是利用白色部分像素灰度值减去黑色部分像素灰度值即可得到 (harrx, harry) 方向向量。这样的向量一共有16个，将这些方向向量的方向角归并到上下左右斜上下8个方向上，并在T1, 2, 3, 4中计算这8个方向的值。

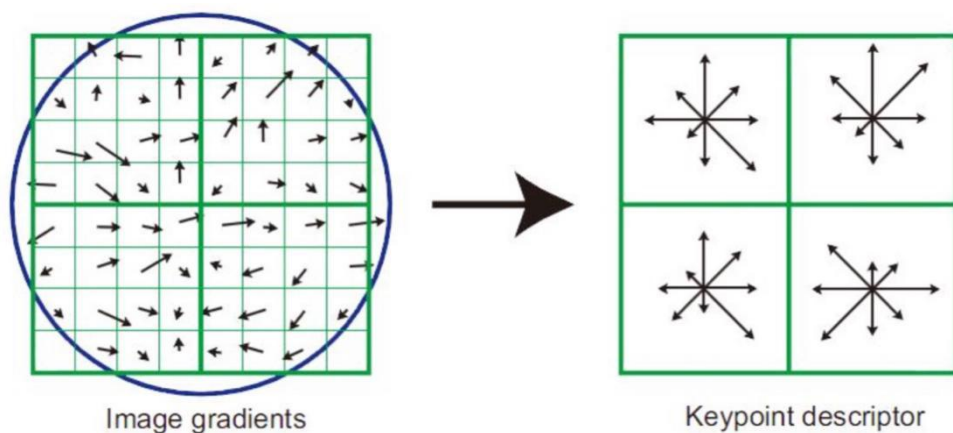


图 2-15

那么这个 $4 \times 8 = 32$ 维向量即为所求的特征描述符。

(3) 特征点的匹配

采用最简单的两向量内积最大值为最匹配的点，设定一阈值，只有当这个最大值大于该阈值方可认为两特征点匹配。

2.4.2 对二进制灰度图像进行 surf 特征点匹配

将现有的我们小组成员找到的DCM变种病毒进行反汇编与去混淆的步骤之后，生成二进制灰度图像，例如：



图 2-16 DCM 变种病毒 1 去混淆之后的二进制灰度图像

再寻找多个样本测试案例，进行反汇编后生成二进制灰度图像，样例1 的图像如下：

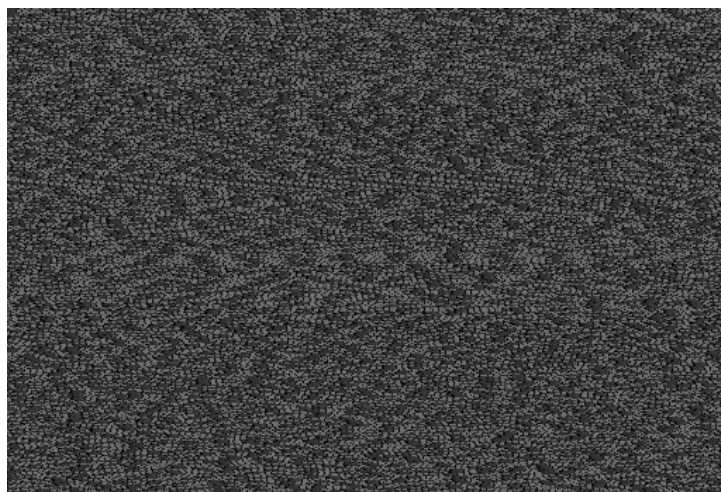


图 2-17 样例 1 的二进制灰度图像

将所得样例图像与病毒图像进行surf特征点匹配，所得局部匹配比率高，说明核心代码语义相似，可认定为是DCM新变种病毒，若匹配率小于5%可认为并非DCM族病毒，判断为安全程序。以下为部分图像匹配结果。



图 2-18 变种病毒 1 与变种病毒 2 的匹配结果图

```
The matching number is : 96
The rate of matching is: 44.04%
Program ended with exit code: 0
```

图 2-19 变种病毒 1 与变种病毒 2 的特征点匹配率



图 2-20 变种病毒 1 与样例 1 的匹配结果图

```
The matching number is : 7
The rate of matching is: 3.21%
Program ended with exit code: 0
```

图 2-21 变种病毒 1 与样例 1 的特征点匹配率

将样例一一与病毒图像进行特征点匹配，即可判断是否为现有病毒的变种，若与病毒库内所有二进制灰度图像的匹配率均小于 5%，可判定为安全程序。

第三章 系统测试

3.1 引言

3.1.1 项目简介

本项目主要结合反汇编和图像处理等相关技术对可执行文件进行恶意代码检测，并且是静态检测，不用执行目标程序，降低了对检测环境的要求，同时由于结合了图像特征值提取和匹配技术使得检测对象是针对整个可执行文件的特征，这个特性也就使得系统对恶意代码的相关变种也具有了一定的检测能力，从而避免了传统基于特征码检测技术的单一性。除此引入图像处理相关技术之外，我们还对反汇编之后的代码进行了去混淆处理，解决了由同一恶意代码经过不同混淆技术从而绕过杀毒软件的检测的问题。最后，虽然本系统在实验特定条件下取得了不错的检测效果，但是实际情况纷繁复杂，检测中可能会遇到加密、压缩等处理后的可执行程序，这些情况下我们的程序就无能为力。

3.1.2 测试目的

编写测试报告的主要目的是介绍系统的运行环境以及测试各个模块的功能是否符合预期的目标，并且当测试出某部分功能无法正常运行或者存在缺陷时提出相应的解决方案，使整个系统最后能够保证所有功能正常协调运行。此外，测试报告还能够提供给用户所有的测试细节，并给出对应的处理方法，以使用户自行安装测试时遇到类似问题可以及时解决，同时让用户明白整个系统的运作原理和存在的缺陷，进而给开发者提出相应意见使得系统功能更加完善。本文档面向的读者主要是系统开发人员、测试人员以及使用本系统的用户。

3.1.3 项目模块结构图

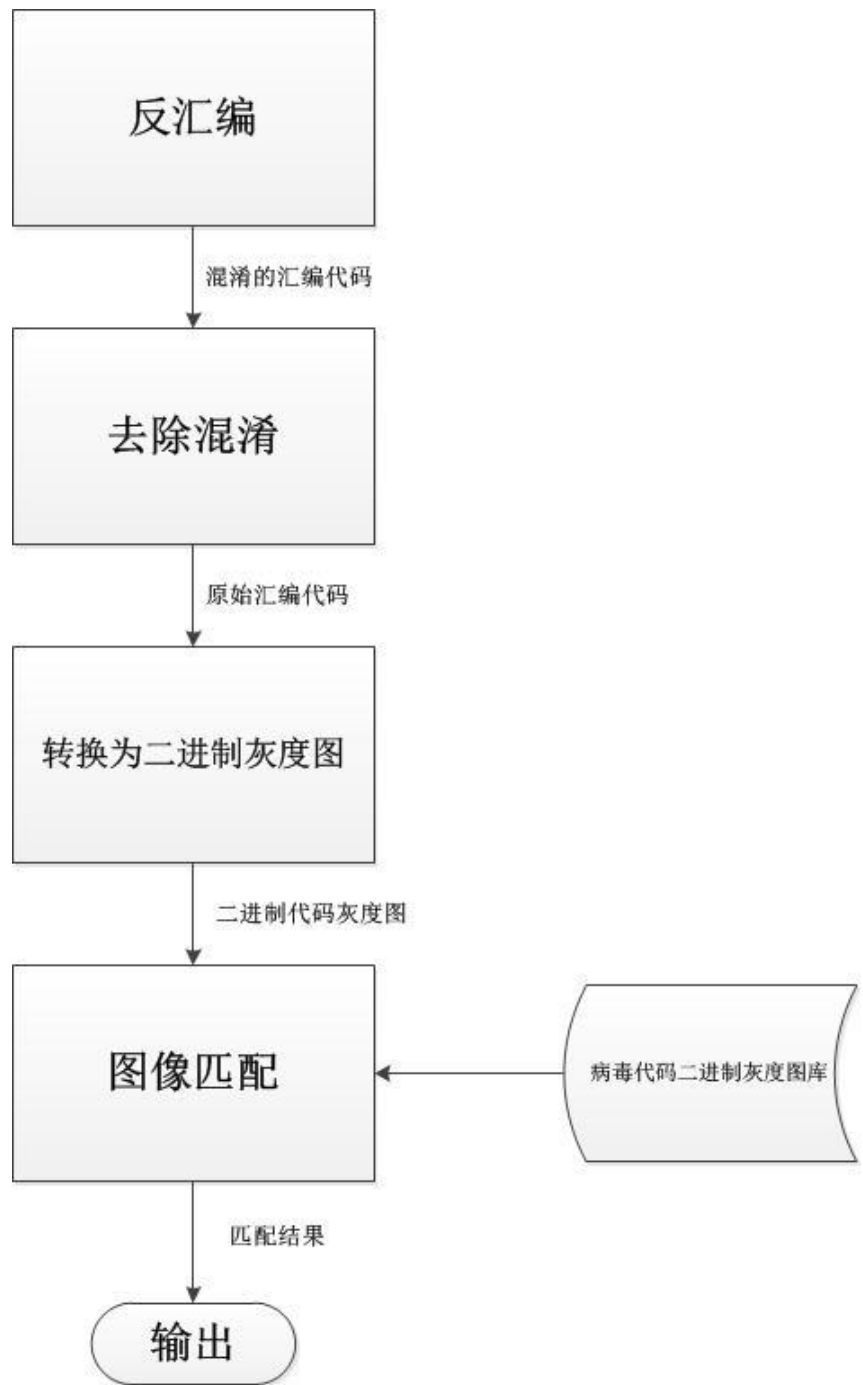


图 3-1 项目模块结构图

3.1.4 开发及测试环境

由于每个模块功能相对独立，并且运行在不同的环境下，所以没有将其整合到一起，每个模块的运行环境如下表所示，

表 3-1 开发环境

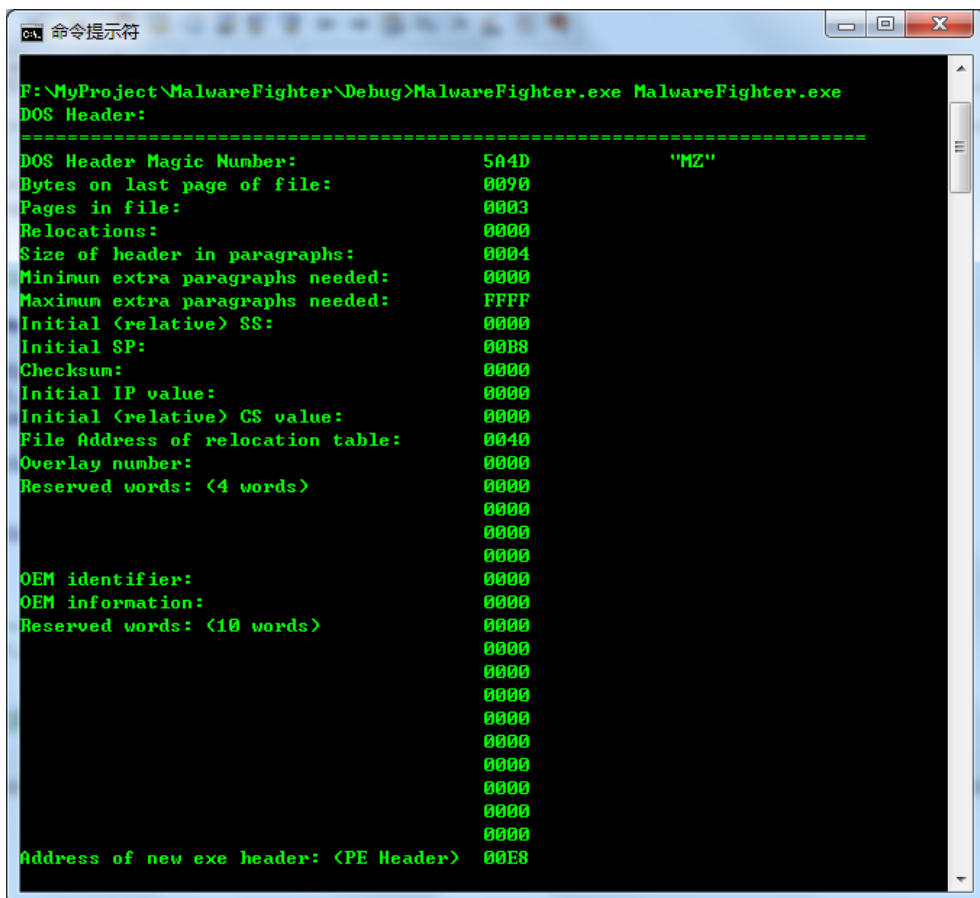
模块	开发环境
反汇编、去混淆	Windows, Microsoft VC++ 6.0
汇编代码转化为灰度图像	Python 2.7.8
图像匹配	OpenCV 3.0

3.2 功能测试

3.2.1 反汇编

对于要检测的目标可执行程序，我们首先要做的就是对其进行反汇编，解析出相应的 PE 结构以及程序执行时的入口点地址。

3.2.1.1 DOS 头

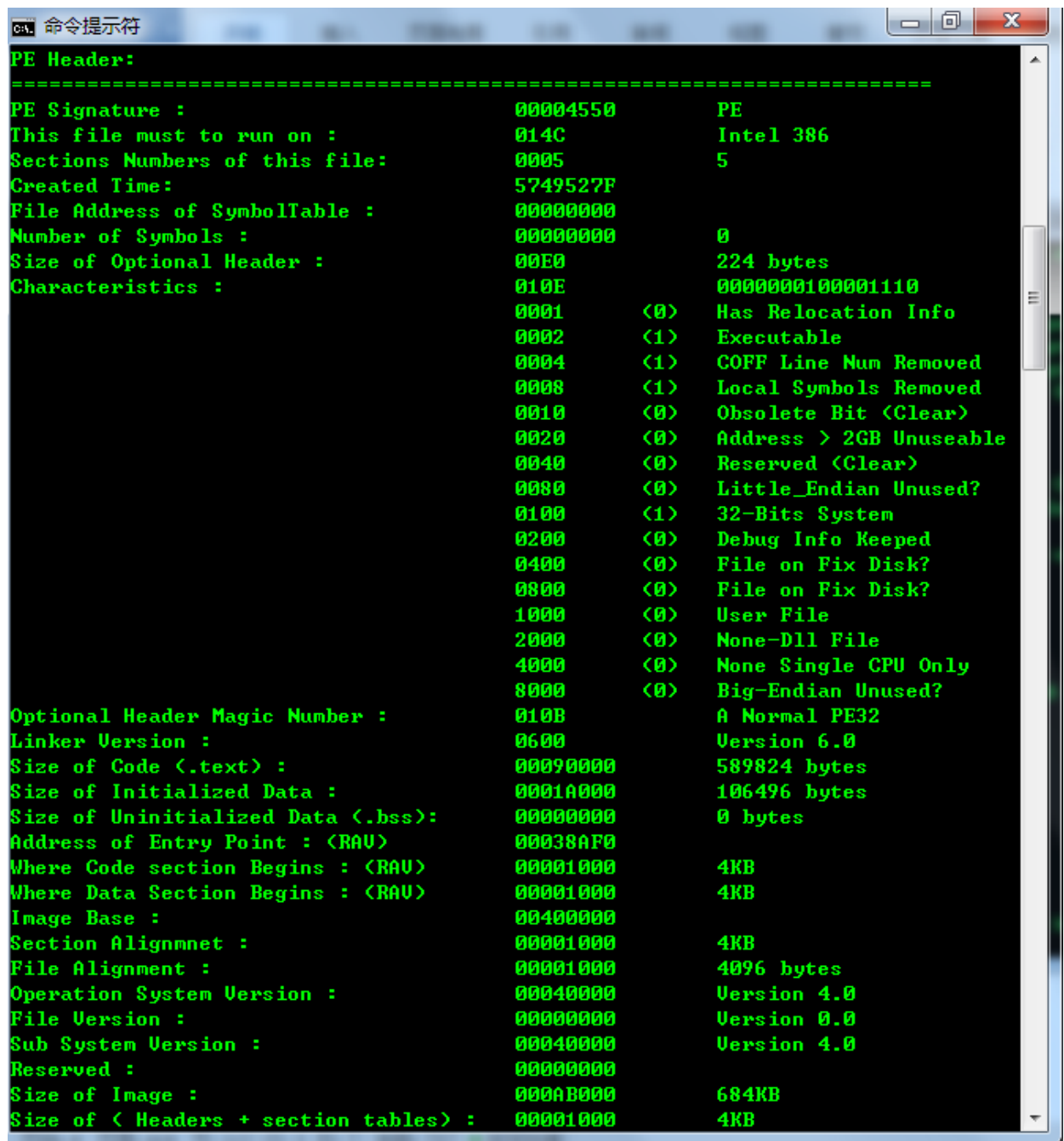


```

命令提示符
F:\MyProject\MalwareFighter\Debug>MalwareFighter.exe MalwareFighter.exe
DOS Header:
=====
DOS Header Magic Number:          5A4D          "MZ"
Bytes on last page of file:        0090
Pages in file:                     0003
Relocations:                       0000
Size of header in paragraphs:      0004
Minimum extra paragraphs needed:   0000
Maximum extra paragraphs needed:   FFFF
Initial (relative) SS:             0000
Initial SP:                        00B8
Checksum:                          0000
Initial IP value:                  0000
Initial (relative) CS value:        0000
File Address of relocation table:   0040
Overlay number:                    0000
Reserved words: (4 words)           0000
                                   0000
                                   0000
                                   0000
OEM identifier:                    0000
OEM information:                    0000
Reserved words: (10 words)          0000
                                   0000
                                   0000
                                   0000
                                   0000
                                   0000
                                   0000
                                   0000
                                   0000
Address of new exe header: (PE Header) 00E8
  
```

图 3-2 PE 文件 DOS 头

3.2.1.2 PE 头



```
命令提示符
PE Header:
=====
PE Signature :                00004550                PE
This file must to run on :    014C                    Intel 386
Sections Numbers of this file: 0005                    5
Created Time:                  5749527F
File Address of SymbolTable : 00000000
Number of Symbols :            00000000                0
Size of Optional Header :      00E0                    224 bytes
Characteristics :              010E                    0000000100001110
                                0001                <0>    Has Relocation Info
                                0002                <1>    Executable
                                0004                <1>    COFF Line Num Removed
                                0008                <1>    Local Symbols Removed
                                0010                <0>    Obsolete Bit <Clear>
                                0020                <0>    Address > 2GB Unuseable
                                0040                <0>    Reserved <Clear>
                                0080                <0>    Little_Endian Unused?
                                0100                <1>    32-Bits System
                                0200                <0>    Debug Info Kepted
                                0400                <0>    File on Fix Disk?
                                0800                <0>    File on Fix Disk?
                                1000                <0>    User File
                                2000                <0>    None-Dll File
                                4000                <0>    None Single CPU Only
                                8000                <0>    Big-Endian Unused?
Optional Header Magic Number : 010B                    A Normal PE32
Linker Version :               0600                    Version 6.0
Size of Code <.text> :         00090000                589824 bytes
Size of Initialized Data :     0001A000                106496 bytes
Size of Uninitialized Data <.bss>: 00000000                0 bytes
Address of Entry Point : <RAU> 00038AF0
Where Code section Begins : <RAU> 00001000                4KB
Where Data Section Begins : <RAU> 00001000                4KB
Image Base :                   00400000
Section Alignmnet :            00001000                4KB
File Alignment :               00001000                4096 bytes
Operation System Version :     00040000                Version 4.0
File Version :                 00000000                Version 0.0
Sub System Version :           00040000                Version 4.0
Reserved :                     00000000
Size of Image :                000AB000                684KB
Size of < Headers + section tables> : 00001000                4KB
```

图 3-3 PE 文件可选头部

3.2.1.3 数据目录

Discription	RVA	Size(bytes)
00: Export Table	00000000	0
01: Import Table	000A5000	40
02: Resource Table	00000000	0
03: Exception Table	00000000	0
04: Certificate Table(File Ptr)	00000000	0
05: Base Relocation Table	000A6000	14736
06: Debug	00091000	28
07: Architecture	00000000	0
08: Global Ptr	00000000	0
09: TLS Table	00000000	0
10: Load Config Table	00000000	0
11: Bound Import	00000000	0
12: Import Address Table(IAT)	000A5194	364
13: Delay Import Descriptor	00000000	0
14: CLR Runtime Header	00000000	0
15: Reserved	00000000	0

图 3-4 PE 文件数据目录

3.2.1.4 区段表

No	Name	USize	UAddr	RawSize	RawOff	Reloc	LineNO	NR	NL	Charact
01	.text	0008F760	00001000	00090000	00001000	00000000	00000000	0000	0000	6000
02	.rdata	0000A670	00091000	0000B000	00091000	00000000	00000000	0000	0000	4000
03	.data	00008490	0009C000	00007000	0009C000	00000000	00000000	0000	0000	C000
04	.idata	00000946	000A5000	00001000	000A3000	00000000	00000000	0000	0000	C000
05	.reloc	00004AA6	000A6000	00005000	000A4000	00000000	00000000	0000	0000	4200

图 3-5 PE 文件区段表

3.2.1.5 导入函数表

命令提示符

Imported Moudles & Functions :

Name	Binded Time	OrigistTk	ForwdChain	FirstThunk
KERNEL32.dll	00000000	000A5028	00000000	000A5194

	Name	Ordinal	RVA
- 01	WideCharToMultiByte	722	000A5300
- 02	MultiByteToWideChar	484	000A5316
- 03	ExitProcess	125	000A532C
- 04	TerminateProcess	670	000A533A
- 05	GetCurrentProcess	247	000A534E
- 06	RtlUnwind	559	000A5362
- 07	RaiseException	523	000A536E
- 08	IsBadWritePtr	440	000A5380
- 09	IsBadReadPtr	437	000A5390
- 10	HeapValidate	423	000A53A0
- 11	GetCommandLineA	202	000A53B0
- 12	GetVersion	372	000A53C2
- 13	LCMapStringA	447	000A53D0
- 14	LCMapStringW	448	000A53E0
- 15	DebugBreak	81	000A53F0
- 16	GetStdHandle	338	000A53FE
- 17	WriteFile	735	000A540E
- 18	InterlockedDecrement	429	000A541A
- 19	OutputDebugStringA	501	000A5432
- 20	GetProcAddress	318	000A5448
- 21	LoadLibraryA	450	000A545A
- 22	InterlockedIncrement	432	000A546A
- 23	GetModuleFileNameA	292	000A5482
- 24	GetCPIInfo	191	000A5498
- 25	CompareStringA	33	000A54A4
- 26	CompareStringW	34	000A54B6
- 27	GetLastError	282	000A54C8
- 28	CloseHandle	27	000A54D8
- 29	SetFilePointer	618	000A54E6
- 30	FlushFileBuffers	170	000A54F8
- 31	HeapFree	415	000A550C
- 32	SetUnhandledExceptionFilter	651	000A5518
- 33	HeapAlloc	409	000A5536
- 34	HeapReAlloc	418	000A5542

图 3-6 PE 文件导入函数表

3.2.1.6 程序入口地址（OEP）

在 PE 文件中，PE 头部结构中包含一个 OptionalHeader，其中就有一个字节代表程序入口的相对地址（AddressOfEntryPoint），并且 OptionalHeader 中还有另外一个字节代表程序加载的镜像基址（ImageBase），所以最终程序的入口地址就应该为 AddressOfEntryPoint + ImageBase，程序运行结果如下。

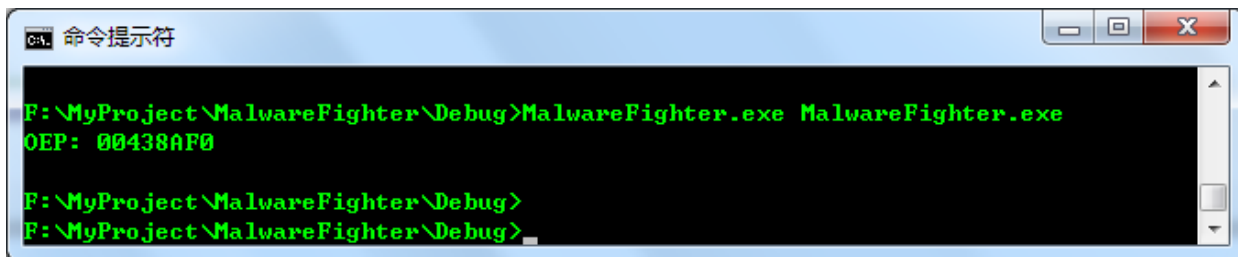


图 3-7 PE 文件入口函数地址

我们在 PEiD 中查看相关信息结果如下，



图 3-8 PEiD 中 PE 文件信息

对比之后发现结果相同，说明解析 PE 文件头部信息部分功能运行正常。

3.2.1.7 反汇编

.text:004020C0			
.text:004020C0	\$E28	proc near	; I
.text:004020C0			
.text:004020C0	var_40	= byte ptr -40h	
.text:004020C0	55	push	ebp
.text:004020C1	8B EC	mov	ebp, esp
.text:004020C3	83 EC 40	sub	esp, 40h
.text:004020C6	53	push	ebx
.text:004020C7	56	push	esi
.text:004020C8	57	push	edi
.text:004020C9	8D 7D C0	lea	edi, [ebp+var_40]
.text:004020CC	B9 10 00 00 00	mov	ecx, 10h
.text:004020D1	B8 CC CC CC CC	mov	eax, 0CCCCCCCCh
.text:004020D6	F3 AB	rep stosd	
.text:004020D8	E8 23 00 00 00	call	\$E25
.text:004020DD	E8 CE 02 00 00	call	\$E27
.text:004020E2	5F	pop	edi
.text:004020E3	5E	pop	esi
.text:004020E4	5B	pop	ebx
.text:004020E5	83 C4 40	add	esp, 40h
000020DD 004020DD: \$E28+1D			

图 3-9 IDA 反汇编结果

4020C0	push	ebp
4020C1	mov	ebp, esp
4020C3	sub	esp, 040
4020C6	push	ebx
4020C7	push	esi
4020C8	push	edi
4020C9	lea	edi, dword ptr [ebp - 40]
4020CC	mov	ecx, 10
4020D1	mov	eax, CCCCCCCC
4020D6	rep	stosd
4020D8	call	402100
4020DD	call	4023B0
4020E2	pop	edi
4020E3	pop	esi
4020E4	pop	ebx
4020E5	add	esp, 040
4020E8	cmp	ebp, esp
4020EA	call	433BD0
4020EF	mov	esp, ebp
4020F1	pop	ebp
4020F2	retn	
4020F3	int3	
4020F4	int3	
4020F5	int3	
4020F6	int3	
4020F7	int3	
4020F8	int3	
4020F9	int3	
4020FA	int3	
4020FB	int3	

图 3-10 程序生成反汇编文件结果

通过对比 IDA 生成的反汇编代码和我们系统生成的反汇编代码，可以验证我们反汇编部分功能基本正常。

3.2.2 去混淆

由于去混淆部分是在汇编代码的跳转逻辑上进行处理，并且唯一直观的结果就是代码大小明显减小了，因为我们不仅去掉了代码生存期中永久不会执行的部分，还去掉了例如 `int3`, `NOP` 等没有任何功能的语句，关于混淆以及去混淆部分的详细原理请参考作品报告。

另外我们对运行结果采取的验证方法就是人工检测，根据代码跳转逻辑人工跟踪，结果发现人工结果与实验运行结果相符合。

3.2.3 汇编代码转换为图像

对于这部分功能，检测的方法很直接，就是观察能否成功的根据汇编代码转化出正常可显示的图像。

为测试这部分功能，我们选择了震网病毒（stuxnet）的某一个变种，并将去混淆前和去混淆后的代码都进行转化，结果如下所示，



图 3-11 震网病毒直接反汇编后图像



图 3-12 震网病毒反汇编并去混淆之后图像

3.2.4 图像匹配

在上一小节我们测试并且成功的将汇编代码转化为相应的图像，通过示例图像我们可以看出代码转化而来的图像并没有什么明显的特征，并且与一般的自然环境或者人类环境下的图像匹配有显著区别，所以适用于一般图像匹配的算法对这种特殊图像可能并不能取得很好的检测效果。我们通过选择几种常用的图像匹配算法进行比较，最终选择了 Surf 算法来对我们处理后得出的图像进行匹配。

由于这部分的功能测试同时还可以检测我们之前提出并且实现的去除混淆代码算法的实际运行效果，所以测试将分为两个部分来进行。首先，我们测试去混淆之前，也就是直接将目标程序进行反汇编之后的结果转化为图像，和我们从 win7 系统中随机抽取四个可执行程序然后进行反汇编并转化为图像进行对比。对比的结果如表 3-2

所示。

表 3-2 尚未去混淆的变种病毒特征匹配

匹配率	DCM 病毒变种 1	DCM 病毒变种 2	DCM 病毒变种 3	测试样本 1	测试样本 2	测试样本 3	测试样本 4
DCM 病毒变种 1	100%	16.95%	19.43%	1.62%	2.32%	0.92%	1.08%
DCM 病毒变种 2	18.89%	100%	36.62%	1.91%	3.02%	1.48%	1.54%
DCM 病毒变种 3	22.50%	37.50%	100%	1.69%	2.69%	0.69%	1.50%

然后,我们对比去混淆之后的图像与测试样本图像之间的匹配程度,结果如表 3-3 所示。

表 3-3 去混淆之后的变种病毒特征匹配

匹配率	DCM 病毒变种 1	DCM 病毒变种 2	DCM 病毒变种 3	测试样本 1	测试样本 2	测试样本 3	测试样本 4
DCM 病毒变种 1	100%	44.04%	44.95%	3.21%	2.29%	1.38%	1.38%
DCM 病毒变种 2	53.89%	100%	58.33%	1.91%	1.11%	0.00%	2.78%
DCM 病毒变种 3	55.29%	57.06%	100%	0.59%	1.76%	1.76%	1.76%

通过表 3-2 和表 3-3 之间的对比,我们可以发现,在去除混淆之前变种之间的匹配率基本维持在 20%左右,需要解释的一点是,例如变种 1 和变种 2 的结果与变种 2 和变种 1 比较的结果并不相同,这是算法本身选取特征点的问题而不是我们实验结果的问题;而病毒变种和测试样本(即无关程序)之间的匹配率基本都在 2%左右,两者之间的区别还是比较明显。

而在去除混淆之后,我们可以发现,病毒变种之间的匹配率达到了 50%左右,并且病毒样本和测试样本之间的匹配率基本维持不变,实验结果证明我们的方法在提高

恶意代码识别率方面的有效性和可行性。

第四章 创新性

4.1 对病毒程序的去混淆预处理

4.1.1 混淆代码的干扰问题

代码混淆俗称花指令，是将计算机程序的代码，转换成一种功能上等价，但是难于阅读和理解的形式行为。代码混淆可以用于程序源代码，也可以用于程序编译而成的中间代码。目前已经存在许多种功能各异的代码混淆器，所以进行代码混淆非常容易。

一般来说，经过混淆的代码更加难于理解。混淆就是对发布出去的程序进行重新组织和处理，使得处理后的代码与处理前代码完成相同的功能，而混淆后的代码很难被反编译，即使反编译成功也很难得出程序的真正语义。被混淆过的程序代码，仍然遵照原来的档案格式和指令集，执行结果也与混淆前一样，只是混淆器将代码中的所有变量、函数、类的名称变为简短的英文字母代号，在缺乏相应的函数名和程序注释的情况下，即使被反编译，也将难以阅读。同时混淆是不可逆的，在混淆的过程中一些不影响正常运行的信息将永久丢失，这些信息的丢失使程序变得更加难以理解。

对于病毒程序来说，代码混淆技术最辉煌的成就却是在黑客们手中实现的——用于保护病毒、蠕虫、木马和rootkit免遭查杀。病毒的作者和杀毒软件厂商已经陷入了一场猫捉老鼠的游戏中：每当杀毒软件厂商开发出一种新的病毒检测技术时，病毒的作者们就会使用一种更为巧妙的代码混淆技术来抵御这种检测技术，而这又促使杀毒软件厂商研发更为强大的病毒检测技术……

4.1.2 去混淆的作用以及对项目的影响

以上说明了混淆代码能使病毒逃过杀毒软件的查杀，也更难让计算机发现或者检测出来。在病毒变种如此之多的时代，为了能更加精确地检测出恶意代码，保护计算机的安全，保障国家、社会以及个人信息的安全。在本次项目中，采用去混淆技术对恶意代码进行去除混淆预处理。

4.2 基于深度优先搜索的去混淆技术

4.2.1 深度优先搜索算法

深度优先搜索所遵循的搜索策略是尽可能“深”地搜索图。在深度优先搜索中，对于最新发现的顶点，如果它还有以此为起点而未探测到的边，就沿此边继续汉下去。当结点v的所有边都已被探寻过，搜索将回溯到发现结点v有那条边的始结点。这一过程一直进行到已发现从源结点可达的所有结点为止。如果还存在未被发现的结点，则选择其中一个作为源结点并重复以上过程，整个进程反复进行直到所有结点都被发现为止。

和广度优先搜索类似，每当扫描已发现结点u的邻接表从而发现新结点v时，深度优先搜索将置v的先辈域 $\pi[v]$ 为u。和宽度优先搜索不同的是，前者的先辈子图形成一棵树，而后者产生的先辈子图可以由几棵树组成，因为搜索可能由多个源顶点开始重复进行。因此深度优先搜索的先辈子图的定义也和宽度优先搜索稍有不同：

$$G_{\pi} = (V, E_{\pi}), E_{\pi} = \{(\pi[v], v) \in E : v \in V \wedge \pi[v] \neq \text{NIL}\}$$

深度优先搜索的先辈子图形成一个由数个深度优先树组成的深度优先森林。 E_{π} 中的边称为树枝。和宽度优先搜索类似，深度优先在搜索过程中也为结点着色以表示结点的状态。每个顶点开始均为白色，搜索中被发现时置为灰色，结束时又被置成黑色(即当其邻接表被完全检索之后)。这一技巧可以保证每一顶点搜索结束时只存在于于一棵深度优先树上，因此这些树都是分离的。

除了创建一个深度优先森林外，深度优先搜索同时为每个结点加盖时间戳。每个结点v有两个时间戳：当结点v第一次被发现(并置成灰色)时记录下第一个时间戳d[v]，当结束检查v的邻接表时(并置v为黑色)记录下第二个时间戳f[v]。许多图的算法中都用到时间戳，他们对推算深度优先搜索进行情况是很有帮助的。

4.2.2 几种混淆类型的去除

针对几种常用混淆手段，我们分别对这几种类型进行去除混淆：无效的代码，变换跳转顺序，从而使经过混淆处理的汇编代码“现出原形”。

4.3 基于二进制灰度图像纹理的恶意代码聚类

恶意代码图像这个概念最早是2011年由加利福尼亚大学的Nataraj和Karthikeyan在他们的论文 *Malware Images: Visualization and Automatic Classification* 中提出来的，我们小组讨论觉得这个想法的思路非常新颖，任何程

序文件都可以以二进制灰度图的形式展现出来,利用图像中的纹理特征对恶意代码进行聚类。本项目中运用了最简单的一种恶意代码图像绘制方法。对一个二进制文件,每个字节范围在00~FF之间,刚好对应灰度图0~255(0为黑色,255为白色)。将一个二进制文件转换为一个矩阵(矩阵元素对应文件中的每一个字节,矩阵的大小可根据实际情况进行调整),该矩阵又可以非常方便的转换为一张灰度图。

据我们小组调查所知,国内这方面的研究较少,所以采取将程序转换成二进制灰度图之后对比纹理特征的方法有可能会开启恶意代码检测的一个全新的时代。

4.4 基于 surf 特征点匹配算法的图像特征分析

图像匹配是计算机视觉和图像处理中的一个重要技术。其方法思想就是根据已知的图像在其他图像中查找出含有已知图像的过程。该技术的研究涉及到许多相关的知识领域,如图像预处理、图像采样、特征提取等,同时将计算机视觉、多维信号处理和数值计算等紧密结合在一起。图像匹配技术还与图像融合、图像匹配等研究方向息息相关,为图像理解和图像复原等相关领域的研究提供基础。

本项目对于上一步骤所得的二进制灰度图像,在目前所有的特征匹配算法中,经过小组讨论,并进行第一阶段的初步测试之后,在gist特征与sift算法还有freak算法中,最终选取了surf算法。SURF算法的全称是Speed-up robust features(加速健壮特征),SURF算法是SIFT(Scale-invariant feature transformation,尺度不变特征变换)算法的加速版,SURF算法可以在适中的条件下完成两幅图像中物体的匹配基本实现了实时处理。Surf算法的步骤可分为特征点检测、特征描述、特征匹配三个步骤。在特征描述子生成后,首先通过Hessian矩阵来进行初始判断,加快匹配的速度,然后采用欧式距离来度量两个特征向量的匹配。欧式距离的相似性度量:对于待匹配图上的特征点,计算它到参考图像上所有特征点的欧式距离,得到一个距离集合。通过对距离集合进行比较运算,得到最小欧式距离和次小欧式距离。设定一个阈值,当最小欧式距离和次小欧式距离的比值小于该阈值时,认为特征点与对应最小欧式距离的特征点是匹配的。我们在项目中选用阈值定为0.8,能够较为准确地找到二进制灰度图像间的纹理相似区域。从而分辨出DCM变种病毒与其他病毒的相似程度以及和样例的匹配。

第五章 总结

5.1 本系统的设计与开发

基于图像处理的恶意代码检测系统是我们Malware fighter团队历时两个月，在目前恶意代码每天呈指数增长的背景下，针对现有检测软件存在的不足而开发的高效、高准确性、变种高识别率的系统。

我们搜集了大量相关系统，找出了它们各自的优势与不足。针对这些不足结合相关的文献制定出优化方案，最终我们确定以图像处理这个新的角度切入。

为解决上述两点不足，本系统在现存恶意代码检测系统基础上主要涉及以下几个新知识点：

- 1、基于同族恶意代码二进制灰度图像纹理相似的特性进行恶意代码检测。
- 2、还原经过混淆处理的汇编代码，为接下来的二进制代码可视化做准备，并提高了最后一步图像匹配的准确度。其中运用到了深度优先算法。主要针对不执行代码和跳转指令两种类型的混淆。
- 3、运用surf图像匹配算法将目标代码的二进制灰度图像与病毒库中的二进制灰度图像进行匹配。

在开发过程中，我们在去除混淆和图像匹配上遇到过困难，为此我们查阅了大量文献，无数次的沟通讨论，最终在团队协作下找到了解决办法，并正逐步优化我们的系统。

5.2 作品展望

本系统的亮点在于将图像处理的技术运用于恶意代码检测领域，并解决了恶意代码检测软件现存的问题：静态检测的变种检测低识别率和动态检测的高系统运行负担。我们也对最新的几大流行病毒进行了测试，震网（stuxnet）、火焰（Flame）、鬼影、黑暗幽灵（DCM）等的变种的检测率得到了很大的提升，且准确度优于传统的静态检测方法。

这应该是在提出恶意代码可视化分析的思想之后的首次将其运用于实际恶意代码检测的大胆尝试。希望能够引发对这一新思路的思考与讨论。我们也会进一步健全和完善我们的系统。

参考文献

- [1] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, Giorgio Giacinto, Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification. CODASPY ' 16, March 09–11, 2016, New Orleans, LA, USA.
- [2] 韩晓光. 恶意代码检测关键技术研究
- [3] Malwise ——An Effective and Efficient classification System for packed and polymorphic Malware. IEEE TRANSACTIONS ON COMPUTERS, Vol. 62, No. 6, June 2013.
- [4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In Proceedings of the 5th ACM Symposium on Principles of Programming Languages (POPL' 78), pages 84 - 96. ACM Press, January 1978.
- [5] Mihai Christodorescu Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns.
- [6] C. A. R. HOARE, I. J. HAYES, HE JIFENG, C. C. MORGAN, A. W. ROSCOE, J. W. SANDERS, I. H. SORENSEN, J. M. SPIVEY, and B. A. SUFRIN. LAWS OF PROGRAMMING. Communications' of the ACM, August 1987 Volume 30 Number 8.
- [7] L. Nataraj, S. Karthikeyan, G. Jacob, B. S. Manjunath, Malware Images: Visualization and Automatic Classification.
- [8] George E. Dahl, Jack W. Stokes, Li Deng, Dong Yu. LARGE-SCALE MALWARE CLASSIFICATION USING RANDOM PROJECTIONS AND NEURAL NETWORKS.