恶意代码的未知检测技术研究

by nEINEI/2007-10

当下，主要的未知检测技术还以行为加权为主，该方法简洁有效。但带来的误报问题也比较明显，该方法最大的问题在于忽略了程序行为之间的逻辑关系，缺乏因果导向。我们可以在这个研究方向上做的更加深入，完成一个真正意义上具有较高检出率的恶意代码检测的专家系统。

未知检测产品的定位是辅助分析，让用户时刻感觉到有一个专家系统在自己身旁。市场上曾经有过像赛门铁克的 ANN 技术，东方微点、安铁诺等产品。以我个人的理解就检测而言，产品比拼的硬功夫的地方重点是，1 检测速度、2 灵活的引擎及其处置能力、3 脱壳能力、4 主动防御体系、5 未知检测技术。所以在该方向上的探索是值得考虑的。

一 技术因素分析

1 主流未知检测技术简要说明：

目前，主流未知检测技术还是基于木马病毒程序的行为特异性权值来判定，因该方法易于实现，且在一定的样本空间中具有较好的检出率。但问题是用户系统环境复杂，带来的多会是虚惊一场的报警。如终截者抗病毒软件会将 INSTDRV.EXE（安装驱动的一个工具）当做高危程序报警。而降低误报的方法只能靠白名单机制来处理。

2 主流未知检测技术分析：

(1) 启发式扫描：

绝大多数杀软采用的方法，NOD32 是该方法的典范，值得我们学习。但该方法仍然不能完全深入其里的分析程序特性，例如：Win32.TrojanDownloader.Small.DTT 木马，会去 http://rasearch.com/ucs5.dat 这个链接处下载文件，将其释放到"C:\WINDOWS\system32\ucsi.exe"，ucsi.exe 就是 usc5.dat 的重命名。

在没有病毒库的支持情况下NOD32是会放过该木马的。这并非NOD32的启发式方法不好，而在于该木马的行为与普通的下载程序无太大差别，不足以触发报警阈值。而我们应当以交互的方式给用户以提示，告知程序行为，如同专家会诊，询问患者病情一样，最终通过推理判断得出结论。

（2）BP 神经网络：

诺顿和安铁诺都采用了该方法，理论上认为神经网络方法要优于传统的启发式方法。诺顿并没有将该方法应用到整个未知检测当中，但很早以前就利用神经网络检测引导型病毒，各中原因猜测不定。

安铁诺是基于神经网络的未知检测技术，官方宣传识别率高达 90%以上，但我实际测试...，个人猜测是受加壳影响。

基于 BP 神经网络的未知检测方法在个别实验室的测试中取得了不错成绩，通过不断的强化训练使得其有了一定的辨认病毒的能力，是其优点。但个人觉得该方法仍然不能应用于实际产品当中，理由有 2：

A 样本空间因素：

BP 神经网络进行病毒检测的依据就是利用样本对其训练，可样本浩瀚，使得模型建立困难，大部分都采取了简化手段，致使部分功能丢失。同时，该方法本质上仍然是对数据做变换，大集合（样本）变换到小集合（病毒库），利用小集合去做检测。当一未知的病毒样本超过了大集合的某种变换时，较高的检出率也就无从谈起了。因为该方法是以样本数量、特性

作为学习基础的。

B 静态特性因素：

BP 神经网络提取的是病毒样本的静态特性，稍加变换的数据都会对结果产生很大影响，加壳的样本是该方法最大的挑战，因为训练加壳后的样本是无太大意义的，此时文件静态特性几乎全部丢失，用 UPX 加壳后的病毒样本与正常程序，在静态特性上无本质区别。

综上 BP 神经网络是不能作为专家系统的支撑技术的。

（3）基于数据挖掘技术：

国内和国外的课题小组都对该技术应用于未知检测领域进行过研究，暂无具体的商业应用。05 年时，金山和福州大学合作开展过该方向上的技术研究。但目前看，尚未知金山毒霸还没有使用该技术。

基于数据挖掘技术的一个比较好的应用是，对样本集的文件提取 API 函数序列作为文件特征，利用改进的 OOA 算法对样本集所产生的特征库进行规则提取，最后利用这些规则进行检测。但该应用，同样受到样本规模的困扰，API 调用序列也过于模式化和绝对。较好的检出率也只是针对特定样本空间的情况。迟迟没有商业应用也可能是上述两点原因造成的。

（4）基于动态仿真技术：

东方微点是使用该技术的典范，通过对程序动作监控，分析程序动作之间的逻辑关系，结合病毒识别规则进行分析，达到自动判定新病毒的目的。

该思路和我心目中的专家系统最为接近。但动态监控的最大问题是要让程序运行起来才能做出判断，有一定的风险，对于向 byshell 这样目的性极强的 Rootkit，一旦让它运行起来就为时已晚，它会用未公开的技术加载驱动，并摘掉所有的内核钩子，那时的监控也就成为了摆设。论坛里面大家也讨论过，可以过卡巴，诺顿，MCAFEE 等，微点也一样能过。因大部分的主动防御会去监控 loaddriver 这个 API，但如果通过其它方式加载驱动进入 RING0，病毒就与杀软又重新回到了同一个起跑线上，此时的优势也就没有了。但动态监控的思路仍有很大的参考价值。

3 小结：

未知检测技术除了上述分析的 4 种外，还有基于免疫原理，基于实例学习等方法，但都过于理论，难于商业应用。还有就是这些方法的出发点都是先学习一定量的样本特性，不可避免的进入到前面所提的问题当中。目前来看，除了启发式扫描与动态仿真有较高的商业价值外，其它方法都不能令人满意。如果专家系统能在技术上大胆创新，取长补短，应该可以开创未知检测领域一番新景象。

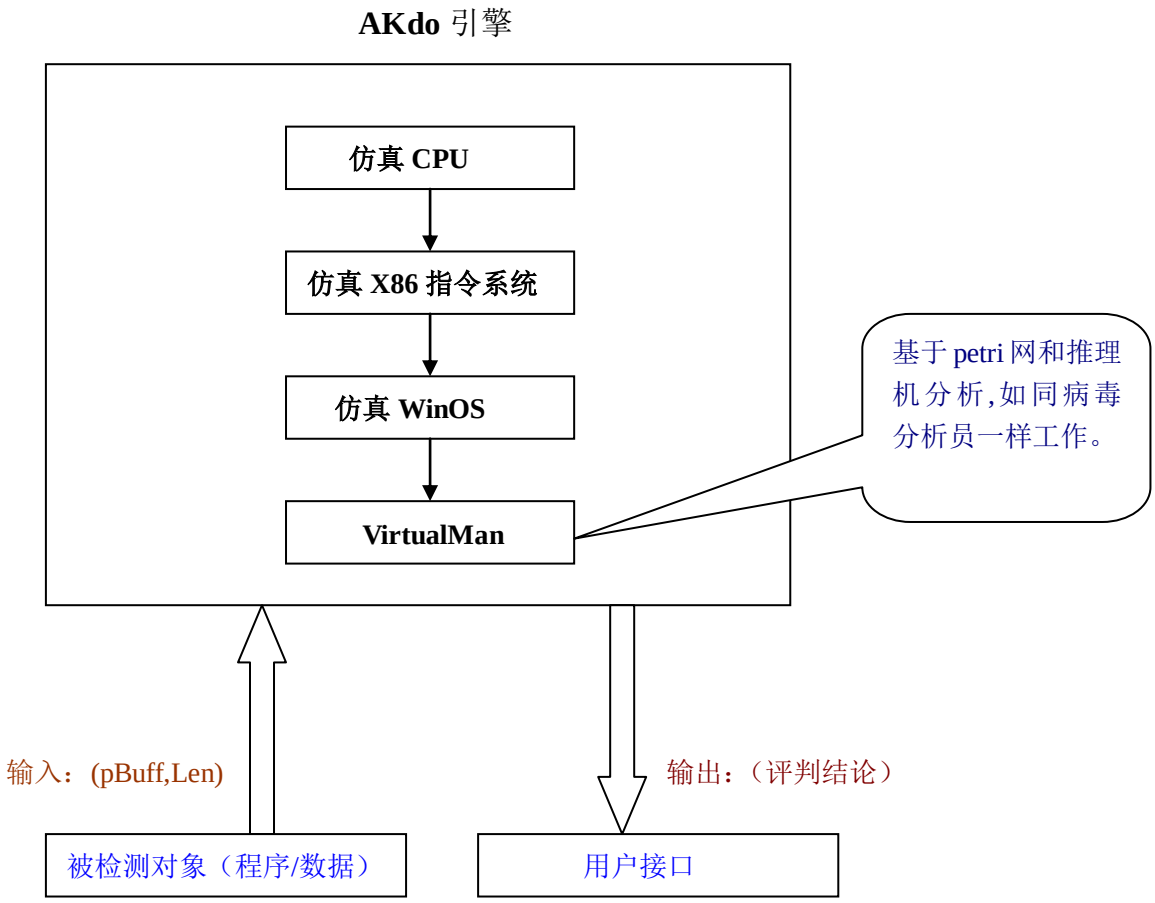
三 恶意代码检测的专家系统的设计与实现

专家系统是以程序的虚拟执行主，配合静态分析（启发式一类方法）为辅，利用进程的 petri 网模型和非确定推理机来完成恶意程序的未知检测。也就是说专家系统的根本出发点是移植病毒分析员的行业经验到软件中，而非针对样本训练检测能力，以往的经验告诉我们，依赖于样本必受制于样本。

在这里虚拟执行是为了深入获得程序的运行时可能出现的特性，为后面的分析提供最基本的数据资源保证。静态分析是对程序的初步判断，它可以最为最终判断提供一方面理论依据。进程 petir 网模型是我们需要建立的恶意程序特性的网状模型，后面会有详细说明，非确定推理机是为最终结论提供推理事实基础。

1 系统框架：

整个检测过程以用户输入 PE 文件为主，DLL 文件和 SYS 文件可作为扩展来考虑，输出为待检测程序的具体结论，结论并一定是病毒或不是病毒这样本明确的是非标准（因讨论具体“是或不是”注定是场口水战），可以有某一事实可能的倾向性。具体见（图一）。



（图一）

1. 仿真 CPU ：主要在仿真环境下模拟取指令，执行指令，及各种寄存器操作。
2. 仿真 X86 体系：主要体现在指令集上，它是真正的执行指令部分。
3. 仿真 WinOS ：仿真 Windows 特性，完成加载程序，API 仿真、SEH 等 0S 特性。
4. VirtualMan ：虚拟的病毒分析员，在指令级别上完成恶意程序的识别。

2 体系结构：

处理流程上主要仿照 AV 工程师处理样本的流程。

由 VirtualMan（以下简称 VMan）通知 WinOS 加载程序，WinOS 分配好了程序空间，通知 CPU 开始执行程序，同时 X86 指令系统开始执行。VMan 获得恶意程序启动初态列表，并下达命令给 VM，当待分析程序位于初态列表中时，进程 petri 网模型（以下简称 P）和推理机（以下简称 IM）开始工作。

此时由 VMan 利用 P 和 IM 分析当前程序逻辑。并通过 P 和 IM 的反馈去设置 VM 执行情况。VMan 是否随着程序执行开就分析取决于程序是否加壳，若加壳则不进行分析，直到脱壳后才去分析。

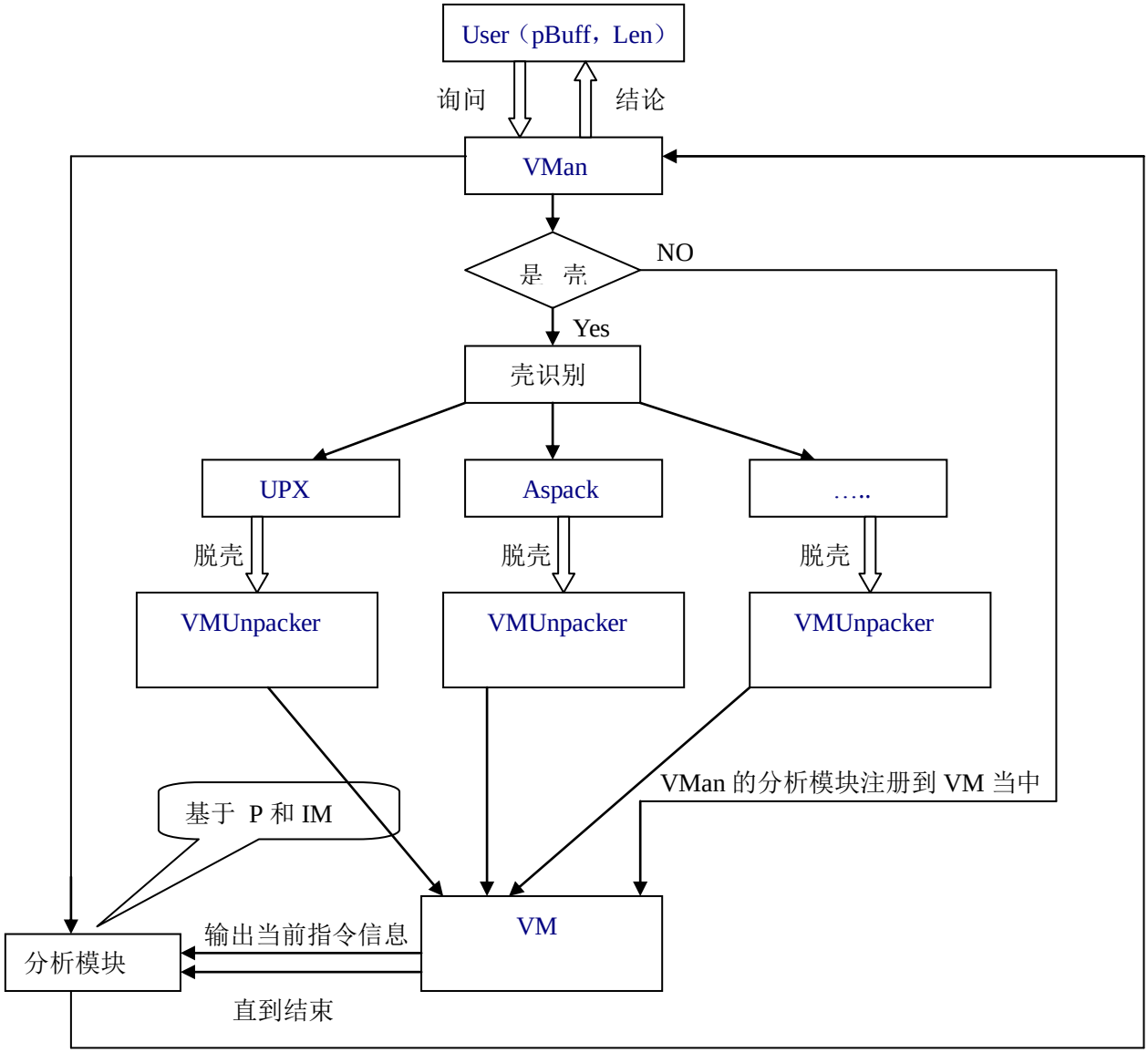
VMan 关心的是当前执行的一组指令序列是否存在恶意操作可能，也就是每一条指令表明一种逻辑关系，如同分析员在分析病毒样本时的逻辑一样，在运行了一定的指令执行集合后，判断是否有恶意的操作。当然在众多的指令逻辑条目集合中分析出恶意逻辑是不容易的，如同刚入行的病毒分析人员，要想完成一次样本分析工作要有一定的经验。VMan 的经验主要来源于 P，配合 IM 完成恶意代码检测工作。体系结构如（图二）所示。

1 用户 — User

2 VirtualMan（虚拟的病毒分析员） — VMan

3 VirtualMachine（虚拟执行环境） — VM 包含 Cpu、x86、winOS

4 壳模块 — UPX….等



（图二）

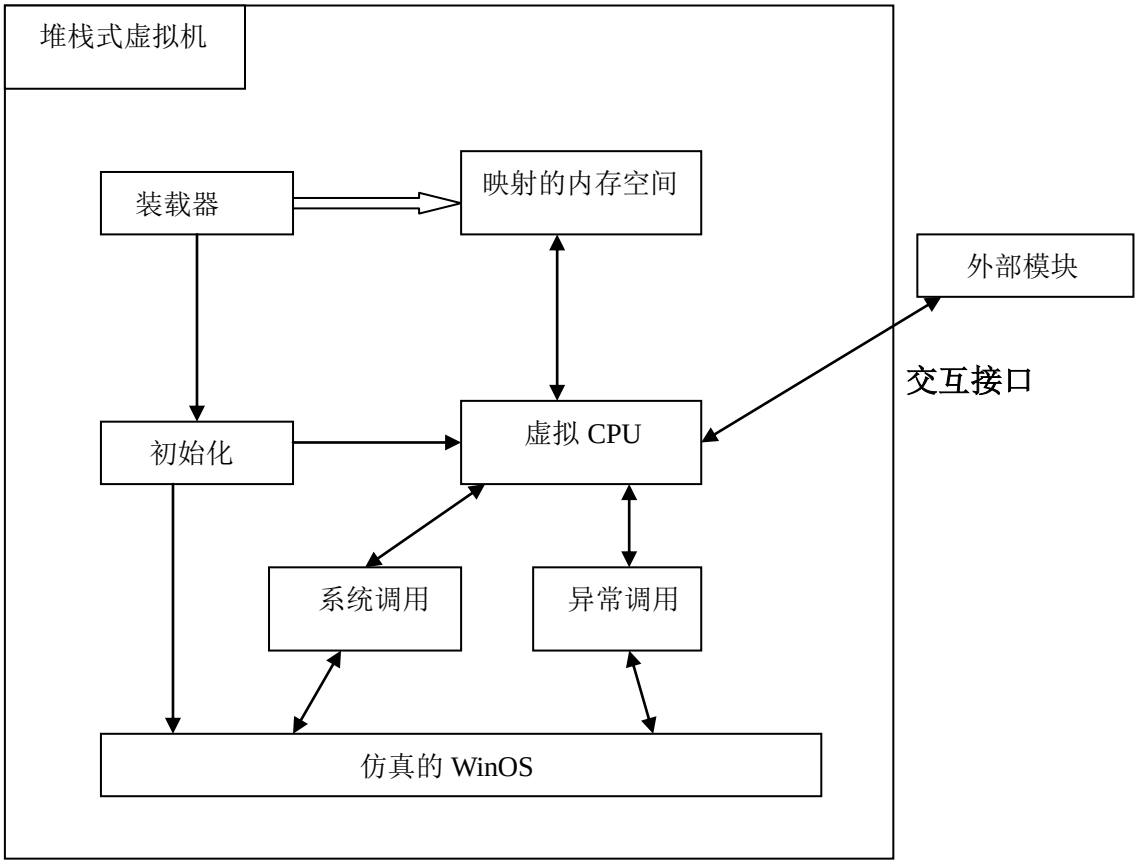
3 技术细节分析：

考虑到病毒程序与木马程序在模拟过程中有较大的差异性，petri 网模型分为木马 petri 模型和病毒 petri 模型。但虚拟执行部分仍提供所有的仿真功能。下面对组成专家系统的各部分做详细说明。

（1） 虚拟执行：

该部分主要是建立一个基于堆栈式的虚拟机，能不受阻断地执行出程序流程（异常中断除外），并记录下模拟运行过程中所出现的特性。

该虚拟机并不达到完全仿真的程度，会化简个别细节操作如程序调用 Sleep，WaitForSingleObject 等操作。体系结构如（图四）所示



1 虚拟机各项特性：

A CPU 模拟：

包括寄存器模拟和取址、译码、执行的模块,完成指令执行的基本功能。

B 系统调用模拟：

这部分主要指由用户态进入内核态的模拟过程，由于系统调用涉及 OS 底层数据结构，如文件系统，IO 操作等，如完全模拟势必要虚拟出内核这部分功能，这显然不是我们关心的。故可以由虚拟机去直接操作。以 win2000 为例，系统调用伪码如下：

```
Void  E_INT2E(REG &reg)
{
     _asm
     {
         Move origEsp, ESP
         Pushad
         Mov  EAX, reg.Eax
         Mov  EDX, reg.Edx
         Mov  ESI, reg.Esi
         Mov  EDI, reg.Edi
         Mov  EBX, reg.Ebx
         Mov  ECX, reg.Ecx
         INT 2E                // WinXP 则改为 ADD ESP ,4  call sysenter
         Mov  reg.Eax ,EAX
         Mov  reg.Edx ,EDX
         Mov  reg.Esi ,ESI
         Mov  reg.Edi ,EDI
         Mov  reg.Ebx ,EBX
         Mov  reg.Ecx ,ECX
         Popad
         Mov  ESP,orgiEsp
     }
}
```

C 异常处理模拟：

该部分模拟出 SHE 链，当异常产生是判断 pre 域是否为空，不为空则跳入自定义回调函数，否则虚拟机退出。

D API 及库函数模拟：

这部分是重点，对于函数的模拟我们坚持以下原则：不在执行过程中中断程序流程。

对向 PostMessage、SetWindowsPos 等一律跳过。仅象征性的调用一下然后恢复堆栈平衡。当然这一类的 API 也并不绝对，如 QQ 木马，会不断枚举当前窗口直到发现 QQ 后才继续执行。所以我们还要搜集这些 API，使这类 API 调用时始终返回真，保持程序继续执行，反调试的 API 除外。

对于文件操作，注册表操作的调用采用“真假调用结合”来处理 API 仿真。因仿真的目是为了取得调用 API 后寄存器，堆栈及内存等数据，使程序流程得以继续进行。对于以注册表某键值，文件中某标志做为发作条件的恶意程序。我们只能去真实的调用，当然我们可以虚拟出文件和注册表，“取“操作调用真实系统的数据，”写“操作则在虚拟的文件和注册表中完成。该功能还处于实验阶段，具体部分还有待于研究。

对库函数的模拟较为复杂，对于用 VC,VB,DELPHI 等工具编写的程序都大量地调用了相应的库函数。比如 MFC42.DLL 封装大量的系统调用操作,一个 CreateFile 也要绕上一大圈子才后去调用，如果不能跟进去模拟势必会丢失很多有用程序特性信息。解决这一问题还没有太好的方法，个人认为有两个方案可以考虑 ：

1 对库的导出函数整理出列表，如同 IDA 那样静态就可以识别出 CFile :: Open ，虚拟机可以直接当作 CreateFile 来处理。

2 是虚拟机载入同被分析程序同样的库（其好处是免去重定位的困扰），跳向该位置虚拟执行。

对于方案 1 可以做前期整理工作，形成 地址 ←→ 函数名 这样的列表。例如跟踪到某条语句时，

jmp dword ptr ds:[402024] ； // 402024 是引入表中的地址

00402020 ：11 7C D3 73 71 0D D4 73 78 B0 D4 73 18 9C DC 73

其中 73D40D71 就为 CFile ::Open 的函数入口，对于意义不很大的函数操作则直接跳过。

对于方案 2 则需要我们自己载入库函数，还以上面为例需要:

LoadLibrary("C:\\WINDOWS\\SYSTEM32\\mfc42.dll");

同时压入当前虚拟机的 ESP，跳入 0x73D40D71 继续模拟执行，直到 ret XXX 后出现原 ESP 值，证明该函数已经运行结束。

E 多线程模拟：

多线程的模拟是一个难点，但因我们并不是设计一个虚拟调试机制，而是一个虚拟执行机制，故可以简化处理。

当跟踪到 CreateThread 时，根据堆栈参数，先分析出 work 线程函数及 work 线程函数参数。然后虚拟机启动一个线程分析该 work 函数。Work 函数可能有监控功能未必立即执行其有效代码，所以虚拟机在模拟主线程与其它 work 函数时要设计好通信功能，使其共享的数据能有效交流。同时为防止带分析程序恶意开启多个线程，所以要设置一个线程上限开关，超过开关数则不予处理。

F 流程预判断：

这部分是为了防止陷入代码局部执行导致无法停机的情况。我们的目标是让程序按一切都为真的情况模拟执行好发现其特性。当某些事件不能模拟，或产生条件不够时，木马、病毒并不表现出明显的恶意行为。此时需及时判断出继续执行的流程，并触发恶意行为。故流程预判断很重要，当虚拟机陷入一系列的循环当中应及时跳出，或设置条件变量。

可以考虑逆编译部分代码片段，分析指令流程，跳向可继续执行的代码部分。

2 变形病毒的检测方法：

目前变形病毒一般在传染前先加密病毒体，然后才进行传染；而病毒在运行前也必须先进行解密器，还原病毒体。检测这类病毒的方式是：先判断是否加密，如果是加密则在虚拟机中进行解密，而后对解密后的病毒体结合病毒 petri 网模型进行分析。

A 解密子判断：

判断程序是否加密主要是在模拟过程中分析的。首先在虚拟机中运行 300 条指令，因为一般解密子不会超过 300 条指令，而且解密程序一般优先运行，解密子的判断可以考虑下面两种方法 1 循环判断、2 内存检测；

循环判断法的依据是病毒解密过程中反复调用解密指令对内存数据进行解密的循环过程，虚拟机先建立一个指令地址的数组（大小为 300），运行过程中保存每条指令的地址，并且与以保存进行比较，如果相同则说明发生过跳转，如果多次发生跳转，并且地址不变则可判断为存在循环。该方法可以判断出存在循环但不能真正判断出程序是否在解密。对于多层加密的病毒，如果加入垃圾指令并且包含循环，这样方法则会造成误判。例如：

```
Garbage：
        Push ecx
        …
        Mov ecx ,100
        Dec ecx
        Nop
        Jnz Garbage
Decrypt:
        Xor byte ptr[esi],123456    ; 123456 是密钥
        Inc esi
        Dec ecx
        Jnz Decrypt
```

内存检测法的依据是在内存空间对非连续性的地址进行解密的病毒非常少，大量病毒还是采用简单的加密手段,无论正向解密还是反向解密都采用地址空间递增或递减的方式进行。这样病毒解密过程中出现的内存区域是连续变化的。

内存检测就以此为依据，具体方法是：虚拟机在执行过程中观察数据是否有写入，如果有则记录下内存变化的地址和引起该变化的指令，当内存地址出现连续变化并且是由同一条指令引发的，则可以判断当前程序正在解密，该方法可以发现解密子的关键部分，如 Xor byte ptr[esi],123456 会引起 esi 指向地址的连续变化，123456 为密钥。

我们可以将这两种方法结合使用，发现循环且地址连续变化由同一指令引起，这两种情况同时发生时才认为程序是在解密。

B 解密过程结束判断：

如何判断解密过程结束并及时停止也是设计的关键，可以用下面两个方法。

一是地址连续变化结束。如果地址变化不在连续，并且引起变化的指令不再是前面保存的解密指令，那么就可以认为解密结束。

二是当 EIP 大于 0x10000000，这时程序可能是在调用 API，因为在解密的过程中极少调用系统函数，如果出现系统调用，则可能是解密结束。

多于采用多层加密的病毒，则需要解密完成后再运行 300 条指令看能否发现解密子。如发现则继续执行。

C 病毒未知检测技术：

这部分识别机制较为复杂，我先给出一个简化的方案。

对未知病毒的检测主要靠病毒运行过程中的特异性来判断，为此建立一个动态特性结构。

```
Typedef struct  _AntiUnkonwVirusSign
{
        Bool  Excep_EipSkip ;          // 指令运行异常跳转
        Bool  CodeReloc;               // 含有重定位
        DWORD Subtle_API               // 敏感 API 个数
} AntiUnkonwVirusSign, *PAntiUnkonwVirusSign;
```

PE 病毒运行过程中特异性有以下几个方面：

1 指令运行异常跳转：

对于 PE 程序，代码运行的正常范围一般位于 PE 文件的代码节内或地址大于 0x10000000

范围的动态链接库调用空间。如果代码运行过程中 EIP 不在该正常范围且位于 PE 程序被映射的地址空间内，则程序可能跳向了被感染的病毒体中运行了。

大部分 PE 病毒都是修改程序入口点，使其指向病毒体，病毒体与正常程序的代码节是分离的，类似 CIH 那样分段感染的也有，这样 EIP 可能位于正常的代码空间，但最终还是会跳向其它节运行的。

对于用 EPO 技术的病毒也一样，虽然初始运行在正常的代码节，但最终还是会跳向病毒体运行。虚拟机一旦发现 EIP 异常跳转，则立刻修改 AntiUnkownVirusSign 结构中的 Excep_EipSkip 字段为真。

2 重定位：

病毒要进行传染就必须对代码进行重定位，重定位代码一般如下：

```
Call  delta
Delta:
      Pop ebx
      Sub ebx,offset Delta
      mov dword ptr [ebx+appBase],ebx
```

对于正常的程序，call [xxx]指令调用紧随其后的地址是很罕见的（加壳程序除外），它可以作为病毒运行的一个标志。在虚拟机中保存 call 调用时堆栈中的返回地址，当调用后转向的地址与堆栈中的地址相同时就可以判断重定位事件产生,还可以更进一步判断 call 之后的指令是 pop 操作。这就更加证明当前在进行重定位操作，置 AntiUnkownVirusSign 结构中的 CodeReloc 字段为真。

3 API 地址搜索：

PE 病毒会主动搜索 API 函数，一般的静态扫描是无法发现这些信息的。病毒在搜索到相关 API 函数后会保存该函数地址，以便以后调用，虚拟机需保存这些地址供以后判断使用。病毒使用的 API 一般包括：FindFirstFile，FindNexeFile，FindClose，CreateFile，DeleteFile，WriteFile，ReadFile，LoadLibrary，GetModuleHandle，GetProcAddress 等。我们将这些统称为敏感 API。

虚拟机现保存这些敏感 API 的数组 Subtle_API[25],并对写如入操作进行监控，伪码如下：

```
Bool E_WriteSubtleAPI(DWORD addr,DWORD data)
{
    If(Search_SubtleAPI(data))  // 敏感 API 操作
    {
          nSubtleAPICount++;
          return true;
    }
    If((addr >= stack) && (addr <= stack + stacksize)) // 堆栈操作
    {
        Return false ;
    }
```

```
        Else if ((addr >= imgageaddr) && (addr <= imgageaddr + imagesize))
        {
            Return false;       // 内存操作
        }
    }
```

需要注意的一个问题是当没有发现敏感操作且没有前两个事件发生,虚拟机应及时停机，否则影响检测效率。一个方法就是在程序首次调用了除 LoadLibrary，GetModuleHandle，GetProcAddress 以外的其它 API 时停机。

我们可以上述 3 点做病毒未知检测的一个依据。

（2）静态特性分析：

这部分主要完成基于程序外部静态特性的检测。可以采用已有的检测方法，也可以重新设计。检测的方面可以包括：

1 是否加壳
2 是否盗用系统图标
3 是否在系统目录
4 是否有版本号信息
5 是否导入敏感 API
6 是否处于启动项中
7 是否捆绑 PE 程序
8 是否有非正常节属性
9 是否最近创建
10 是否有可疑文件名（如随机 8 位病毒或数字前缀）
…
…
…

更深一层次还可以分析，节与节之间是否有空隙，含有空隙的节中是否有数据，该部分可通过计算数据复杂度的来判断是代码或数据的可能（而不 0x00 或 0xFF）。

通过这部分检测会对程序有个最初的判定，为最后结论提供一方面证据。

（3）进程 petri 网络模型：

1 选择用 petri 网的原因：

专家系统作为人工智能领域的一个分支有着广泛的应用，如地质勘察专家系统，医疗专家系统，天气预报专家系统等。但反病毒领域和这些应用有截然不同的基础，故不能模仿那些应用来解决我们的问题，原因如下：

1 人工智能领域中大家普遍接受这样的观点，要让机器解决一个智力问题，人必须先知道如何解决。换句话说人必须有知识，在具体领域里知道如何做。而专家系统大部分都是基于知识规则的，在专家系统中知识的表示形式最常用的就是产生式和一阶谓词。而产生式和一阶谓词最大问题就是效率低下，设计不好的 100 条的规则足以产生组合爆炸，同时它们

也不能表达具有结构化的知识，如横向和纵向的知识，而程序逻辑恰好是横向和纵向都包含的。其次那些领域的专家系统运行在服务器上，而我们的专家系统是要跑在用户 PC 上，效率上的差异不可比拟，故我们只能寻找其它的知识表达形式。

2 基于规则的专家系统中知识存在优先级，规则冲突是最大的问题，为解决冲突还要寻找各种办法如引入元知识的概念，我们的精力不在于此。

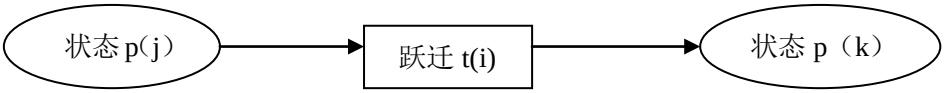
3 产生式的搜索策略低效，要反复推理知识库中的规则，这也是普通用户 PC 机所不能承受的。

基于上述三点原因，我们的专家系统不用产生式来表达知识，而用 petri 网来描述。

2 petri 网描述：

Petri 网是用来描述异步并发现象的动态系统模型，作为知识表示的方法之一，如今已发展成单独的一门学科了。当把一个进程看作一个活动的系统时就可用 petri 网来描述。

Petri 网是由状态和跃迁这些基本元素组成的。



对于产生式的表达 if d（j）then d（k）

P(j)代表第 j 个位置-------- 产生式的前提 d(j)

P(k)代表第 k 个位置-------- 产生是的结论 d(k)

t(i)是某个转换

当然对于复杂的系统或知识，petri 网可以用一个四元组来表示知识的因果关系，其形式为（P,T, F,A）

P 表示状态的有限集，记为 p = {p1,p2,p3,p4,…}

T 表示转换的有限集，极为 T = {T1,T2,T3,T4,…}

F 相关函数，表示强度，0 ~ 1 之间。

A 相关函数，表示位置对应命题的可信度，0 ~ 1 之间。

病毒木马程序与一般程序的区别在于执行一些特殊动作来破坏系统，这些动作就可以看作 P 集合，动作之间的关联因素就是 T 集合，状态跃迁中的强度就是 F 函数，程序执行到某状态后相对命题（是否病毒）的可信度就是 A 函数了。

对于木马程序一般的操作如下：对 Win.ini、system.ini 文件夹特定项的修改，对注册表特定键值的修改，如文件关联，端口异常与端口上的数据流量，硬盘数据共享，远程文件操作，抓屏操作，计算机关闭与启动，鼠标钩子，线程注入，文件复制，修改等。

我们可以根据这些特性行为，构建起 petri 网，这些行为都由 API 调用产生，我先整理一部分这样的 API，如（图五）

序号	程序行为	特定 API 调用	危险程度	所属链接库
1	堆操作	RtlFreeHeap,RtlAllocateHeap	2	Ntdll.dll
2	动态库加载	LoadLibrary,GetModuleHandle	1	Ntdll.dll
3	API 地址获取	GetProcAddress	1	Ntdll.dll
4	进程操作	OpenProcess,CreateThread	2	Ntdll.dll
5	内存读写	VirtualAlloc,GetProcessHeap OpenMutex,VirtualProtect	2	Ntdll.dll
6	读注册表	RegOpenKey,RegEnumkey…	1	ADVAPI32.DLL
7	写注册表	RegSetValue,RegQueryValue…	1	ADVAPI32.DLL
8	程序执行	WinExec,CreateProcess	2	KERNEL.dll
9	文件操作	CreateFile,ReadFile,WriteFile	1	KERNEL.dll
10	文件搜索	FindFirstFile,FindResource FindNexeFile	2	KERNEL.dll
11	目录搜索	GetWindowsDirectory, GetSystemDirectory, CreateDirectory	3	KERNEL.dll
12	目录删除	RemoveDirectory	3	KERNEL.dll
13	磁盘操作	GetDriveType,GetDiskFreeSpace	1	KERNEL.dll
14	时间操作	GetTickCount,GetSytemTime GetLocalTime	1	KERNEL.dll
15	系统重启	ExitWindows,AbortSystemShutdown InitialteSystemShutdown	2	KERNEL.dll
16	加密解密	CryptEncrypt,CryptDecrypt, CryptAcquireContext…	3	ADVAPI32.DLL
17	远程通信	Send,recv,bind,listen…	2	WSOCK32.dll
18	进程搜索	EnumProcess,EnumProcessModule, GetModuleBaseName	2	PSAPI.DLL
19	线程注入	CreateRemoteThread	4	KERNEL.dll

（图五）

这样构建起的进程 petri 网模型就有了动作之间的因果联系，可以通过不断的反馈去设置虚拟机监控的状态，在完成一部分进程 petri 模型的遍历后就可以通过推理得出一定结论了。

（4）非确定推理机：

这部分则依据数学定理完成一个非确定推理的过程,可以有两个办法一个是基于贝叶斯推理，一个是确定因子技术。

首先我们认为一个特定事件 E 的产生对结论 H 的影响是

$$P(H|E) = \frac{P(H \cap E)}{P(H)}$$

经过变换可以得到

$$P(H|E) = \frac{P(E|H) * P(H)}{P(E|H) * p(H) + p(E|-H) * p(-H)}$$

同时我们定义 $LS = \frac{P(E|H)}{P(E|-H)}$

LS 的值表示在证据 E 存在时，专家系统估计假设 H 的可信度，LS 值高（LS>>1）表示证据 E 存在时该规则强烈地支持假设（也就是我们认为是病毒的那一个结论）。

再定义 $LN = \frac{P(-E|H)}{P(-E|-H)}$

LN 的值表明证据 E 缺失的情况下，不信任 H 的度量，LN 的值低（0< LN <1）表明证据缺失时，规则强烈的反对假设。

通过这些公式我们就可以得出当进程处于某一状态时，对是恶意程序或不是恶意程序的这样结论的理论可能性，配合静态特性就可以给出最后的结论了。

当然上提到的仅是一个事件对结论的影响，可对上面公式进行变换使其支持多重证据。

因确定因子技术是贝叶斯推理的一个补充，与此类似不再赘述，我们的推理机可以考虑结合这两种方法进行。对于进程 petri 网模拟中状态的切换关系如何在推理机中不断得到证实，还需在实际开发深入讨论。

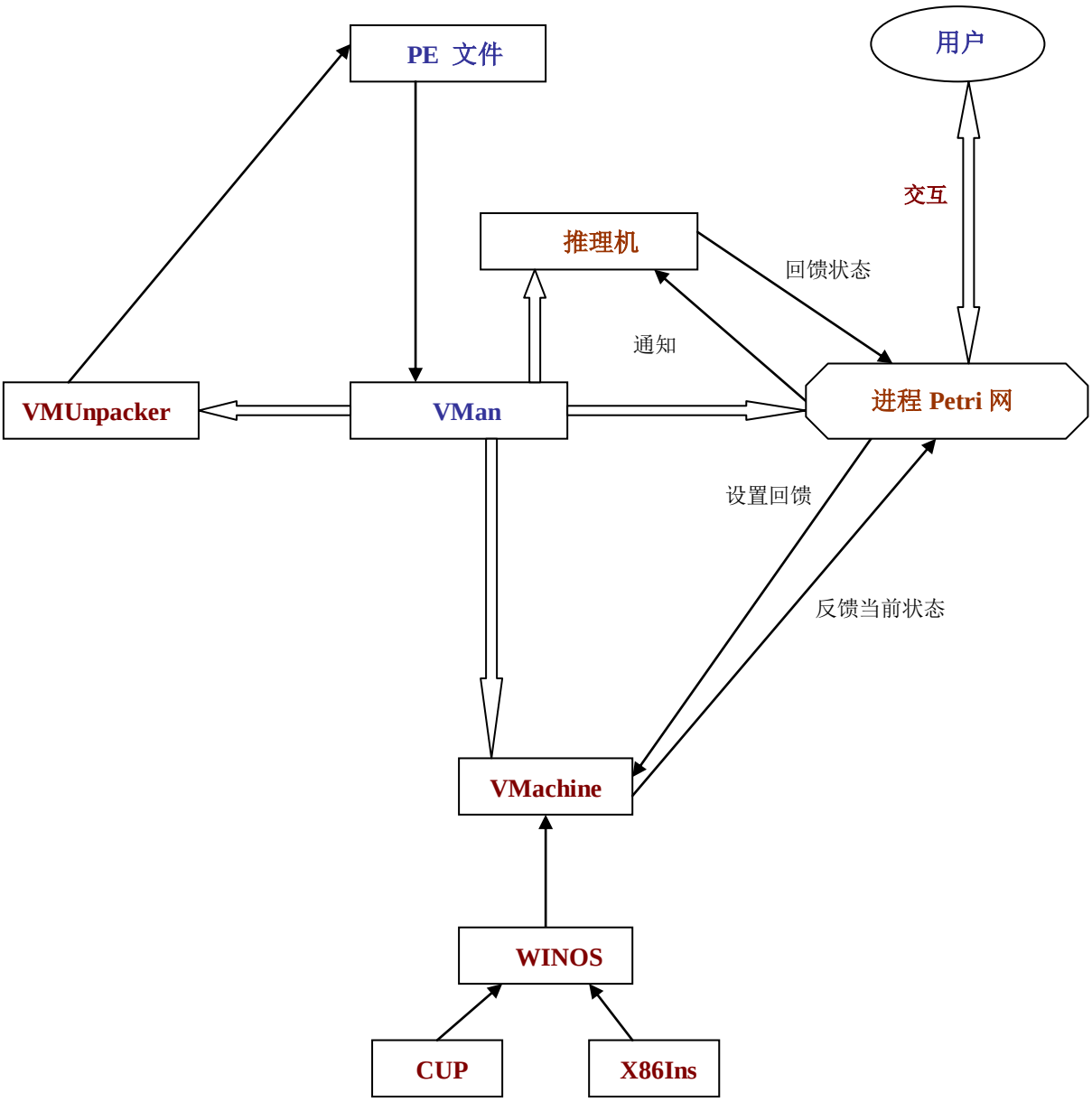
在 petri 网的每一个状态中，可由病毒分析员的经验给出 LS 与 LN。例如对于线程注入这一个环节可以描述为：

Rule 1 ：
If 远程线程注入产生 （LS 6 ,LN 0.4）
THEN 该程序是无进程木马 （prior 0.5）

这样经过不同的状态环节后，就可以由推理机给出一个理论上的结论了。

4 程序流程：

整个程序执行流程如下图六所示。



（图六）

这就是基于专家系统进行未知检测的整体设计思路。

关键技术：

1 CPU 模拟

2 WinOS 特性模拟

3 API 和库函数模拟

4 进程 petri 网模型建立

5 非确定推理机

目前来看，技术难点在于整体运行效率的把握及 winOS 特性模拟这两方面上。

[参考文献]

1) 《ring3 级 32 位 x86cpu 仿真》http://bbs.pediy.com/thread-42343.htm
2) 《堆栈式虚拟机设计》 电子科大的一篇硕士论文，记不得具体名字了
3) 《虚拟机 C++实现》介绍堆栈式虚拟机的书籍，实现了简单的指令集
还有不少网络上的分享的启发式检测和知识表示相关的技术资料

【后记】
这篇技术思路起于 2006 年在 AT 工作时的想法，最早写于 2007 年 8 月，当时辞职赋闲在家。想好了以后要开发的技术方向，遂找来 Linxer 一起讨论，因写仿真 X86 虚拟机技术他是最在行的，想着新思路马上就能实现甚是激动一番。数日后又同 Killer 聊起这个技术点，一番理想与现实挣扎后，来到新公司去搞防漏洞/挂马的的脚本引擎，也就是日后的畅游项目，之后再没做这方面工作。 2009 年时下定决心还是要出来继续搞一搞，同 Linxer，Xq 商量，选定个方向继续做了下去，三个不满 30 的毛头小伙子，一起朝着未知检测方向努力。记不得多少次通宵达旦，夜以继日，发布 1.0 版本的时候忐忑不安，有鼓励，有嘲讽，终于在 2011 收到 VB100 测试总监的来信“一个来自中国的小团队，用不到 2M 大小的引擎，不更新特征的情况下，有~60%的检出率”,那一刻我觉得我们接近偶像级技术 NOD32 了，因为把未知检测和误报率和引擎效率控制好平衡，太困难了，单纯做好其中一项是相对容易的，NOD32 是优秀的代表，Norman 被 AVG 收购了，AVG 被 AVAST 收购了，不知道这一类技术是否还会得以传承。

现在看来这篇技术写的非常粗略，很多地方考虑的不够周到，甚至有些臆想幼稚可笑。多线程技术在仿真 X86 虚拟机里面是可以实现的即便是中等强度的加密壳的反调试也是可以搞定的，静态是可以做到模拟流程，参数提取的，花指令是可以通过 2 遍交叉反编译过滤掉然后再静态分析。而在挖掘信息不够理想的情况下推理机却没有想象的那么强大，我们也没有使用进程 petri 网模型来完成交互，虽然至今我仍然认为其是一种比较理想的解决办法，但 API 序列识别针对恶意样本家族反而更简单易于实现，若知 AI 技术今日取得如此成功，为当初的放弃也略感可惜。我们妥协了很多技术理想，也实现了很多不曾想过的技术。

距离 07，刚好 10 年了，曾经一起奋斗过的兄弟们还记得当年的理想吗？AT 的兄弟们大多数还在冰城继续书写传奇，Linxer 当了老板但仍然天天写代码，Xq 去年开始了创业，大 S 做了产品经理，小 S 和 mtian 去了 tx, 十二羽翼去了上海，护驾游戏了，小倪开了公司专门对付勒索软件，Wr 去了数字身价高高，当然，还有我和 Robinh00d 继续在安全行业里干着。

终归，念念不忘，必有回响。
2017-07