二进制动态翻译

changkunli@vip.qq.com

背景

- 虚拟机分类：
  - 系统虚拟机，VMware
  - 语言虚拟机，JVM
  - 进程虚拟机
- 反病毒引擎模拟器： mach32
- 动态脱壳，通用脱壳，高级启发式，多态变形病毒，检测感染型

对比JVM与Mach32

	JVM	Mach32
输入	class file, byte code	X86 PE file, binary opcode
输出	real action	fake action
生命周期	long	very short
指令集	byte code, based stack	X86 Instruction, complex
限制性	none	any

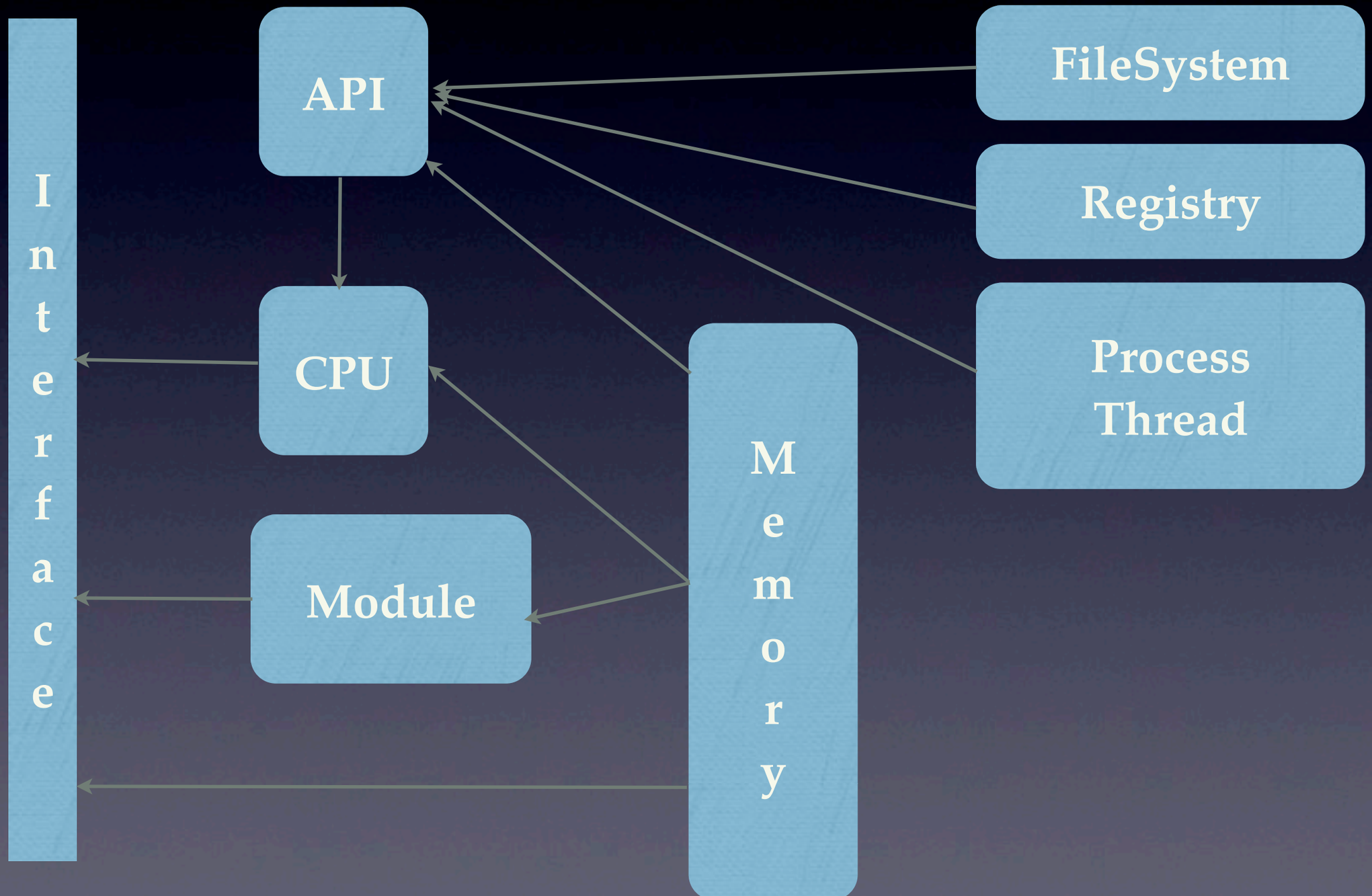
X86指令特点

- 指令集庞大
- 指令影响面广，大部分指令都会影响1-6个标志位
- 数据与代码区分不开，可以在栈，堆，代码区，数据区
- 代码可以自修改

组件

- 指令模拟
- 内存分页，地址空间管理，堆管理
- Modules(环境)
- 文件，注册表系统
- 进程列表，多线程，同步
- 窗口，消息循环

结构图


API
FileSystem
Registry
Process
Thread
CPU
Module
Memory
Interface

基本思想

- 把我们的内存（数组）当对方的寄存器
  - 通过操作内存来实现修改目标寄存器
- 把我们的『存储设备』当对方的内存
  - 通过函数调用显式解析页表来访问对方内存数据

模拟器初始化过程

- 类似于一个微型的Windows
- 初始化内存管理，分页，地址空间，堆
- 加载PE文件，注册对方PE空间到内存管理
- 加载进程相关信息，PEB，TEB，共享内存，模块链，环境变量
- 解析导入表加载依赖的动态库进入内存管理
- 初始化寄存器，栈数据为开始运行做准备

寄存器结构

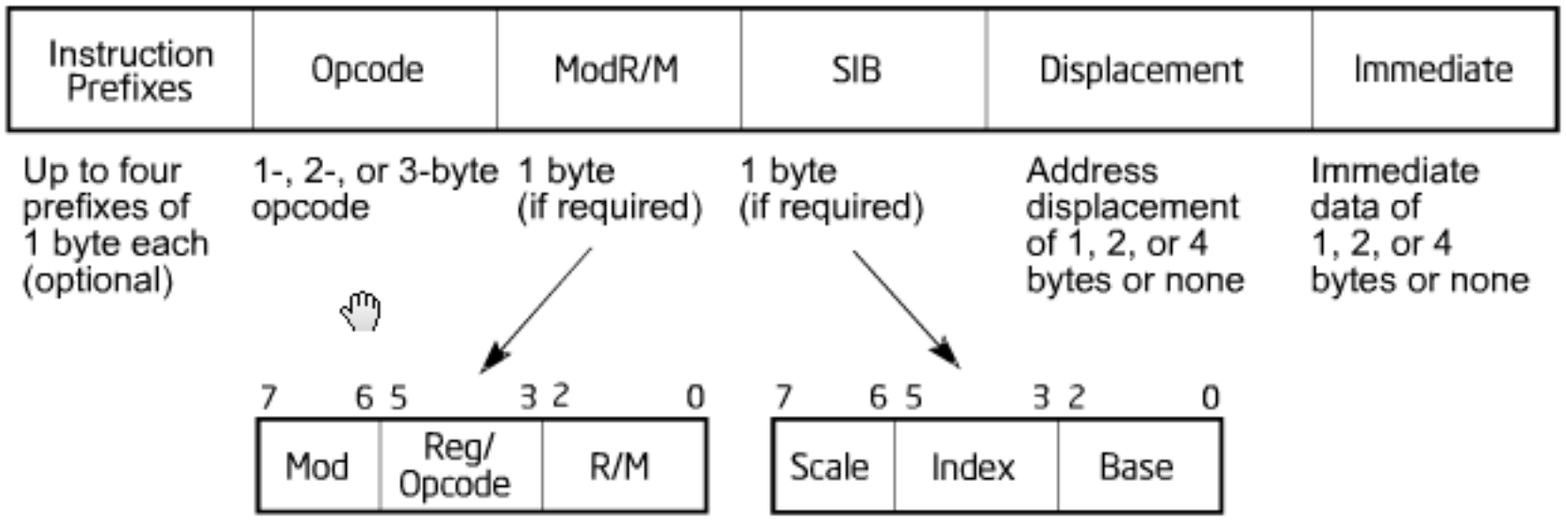
- 通用寄存器： t_uint32 m_VM_Reg[8]
- 标志寄存器： t_uint32 m_VM_EFlag
- 段寄存器： VM32_SEG m_VM_Seg[6]
- FPU, MMX

解释器工作流程

- 取指令（EIP地址是否可执行，调试寄存器）
- 解析指令（支持单，双字节指令）
- 根据解析的结果执行动作，比如操作虚拟寄存器。
- 循环取下一条指令.....

指令格式

复杂指令集


Instruction Prefixes
Opcode
ModR/M
SIB
Displacement
Immediate
Up to four prefixes of 1 byte each (optional)
1-, 2-, or 3-byte opcode
1 byte (if required)
1 byte (if required)
Address displacement of 1, 2, or 4 bytes or none
Immediate data of 1, 2, or 4 bytes or none
7 6 5 3 2 0
Mod
Reg/ Opcode
R/M
7 6 5 3 2 0
Scale
Index
Base

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

- 00405000 8BC3　mov　eax, ebx
  - GetModRmReg()=>m_VM_Reg[REGISTER_EBX]
  - SetRegDword(t_int32 nIndex, t_uint32 value) => m_VM_Reg[index] = value
- 00405002 8B18　mov ebx, dword ptr[eax]
  - GetModeRmReg()=>m_VM_Reg[REGISTER_EAX]
  - GetMemDataEx() => return GetMemData(GetDataBase(Seg_Ds)+addr)
  - SetRegDword(REGISTER_EBX, value)

解释器的优缺点

- 控制精度高，有利于外界做虚拟调试分析样本
- 性能差

二进制动态翻译

- 如何提升性能?
- 以块为单位将对方OPCODE翻译成可以直接执行的机器码块

- 动态翻译-即时翻译即时执行
- 怎么定义块?
- 将翻译后的机器码块缓存，索引，生成跳转直接链接起来

块的定义

- 下面是翻译停止条件（作为块结尾）
  - EIP转移指令(jxx, call, retn, int3...)
  - 暂时处理不了的指令
  - 翻译的指令太多，外界控制断点，调试寄存器，无效EIP

```
00405000   60              pushad
00405001   E8 00000000     call    00405006
00405006   5D              pop     ebp
00405007   81ED F31D4000   sub     ebp, 00401DF3
0040500D   B9 7B090000     mov     ecx, 97B
00405012   8DBD 3B1E4000   lea     edi, dword ptr [ebp+401E3B]
00405018   8BF7            mov     esi, edi
0040501A   61              popad
0040501B   60              pushad
0040501C   E8 00000000     call    00405021
00405021   5D              pop     ebp
00405022   55              push    ebp
00405023   810424 0A00000  add     dword ptr [esp], 0A
0040502A   C3              retn
0040502B   8BF5            mov     esi, ebp
0040502D   81C5 9A050000   add     ebp, 59A
00405033   896D 34         mov     dword ptr [ebp+34], ebp
00405036   8975 38         mov     dword ptr [ebp+38], esi
00405039   8B7D 38         mov     edi, dword ptr [ebp+38]
0040503C   81E7 00FFFFFF   and     edi, FFFFFF00
00405042   81C7 59000000   add     edi, 59
00405048   47              inc     edi
00405049   037D 5C         add     edi, dword ptr [ebp+5C]
0040504C   8B4D 58         mov     ecx, dword ptr [ebp+58]
0040504F   41              inc     ecx
```

00405000-00405001

00405006-0040501C

00405021-0040502A

基本思想-操作寄存器

- 将ebp指向m_VM_Reg[8]的首地址
- mov eax, ebx将被翻译成：
- mov eax, [ebp+3*4]
- mov [ebp], eax

基本思想-操作内存

- **mov [eax+4], ebx=>**
- mov eax, [ebp]
- mov ecx, [ebp+4*3]
- lea eax, [eax+4]
- mov [esp], ecx
- push eax
- mov eax, [ebp+SEG_DSOffset]
- add [esp], eax

push [ebp+MemMangerOffset]

push [esp+8]

call SetMemData32

cmp eax, 0

je xxxx //出现异常或代码自修改

leaveBlock...

return

xxxx:

下一条指令

链块

- 块执行完后，返回下一个要执行的虚拟地址
- 先查找缓存，是否翻译过，如果没翻译过，进行翻译然后执行，如果已经翻译过直接执行，并且把两个块用跳转连接起来。

- ....
- mov eax, ebx
- jmp 00401000

mov eax, [ebp+3*4]

mov [ebp], eax

jmp xxx:(E9 00 00 00 00)

xxxx:

LEAVE BLOCK

- ....
- add eax, ebx
- je 00401000

```
mov eax, [ebp+3*4]

add [ebp], eax

je xxxx(74 offset_x)

jmp yyyy(E9 00 00 00 00)

yyyy:

LEAVE BLOCK

xxxx:

LEAVE BLOCK
```

- ....

- add eax, ebx

```
mov eax, [ebp+3*4]

add [ebp], eax

je xxxx

jmp (e9 00000000)

LeaveBlock

xxxx:
```

- je ecx

```
mov eax, [ebp+4]

cmp eax, 0

je (74 0000000)

LEAVE BLOCK
```

提升性能的本质

- 循环里面，块被链起来，执行进去直到循环条件退出。
- 省去了，取指令，解析指令的过程，执行的效率高

翻译流程

- 翻译成中间代码（简单指令）
- 优化（特定的复写传播，标志寄存器，生成代码寄存器压力控制）
- 分配寄存器
- 生成可执行代码

微指令

- 由临时寄存器T组成(数量不限)，超简单的指令
- Mov_Imm_To_T(8,16,32)
- Mov_Reg_To_T/Mov_T_To_Reg (8,16,32)
- Mov_T_To_Mem/Mov_Mem_To_T (8,16,32)
- Calc_T_T (8,16,32)
- Jxx_Address/Jmp_T
- Save_EFlag/LoadEflag

- mov eax, ebx=>
- Mov_Reg_To_T32
- Mov_T_To_Reg32
- add eax, [ebx+4]=>
- Mov_Imm_To_T32
- Mov_Reg_To_T
- Add_T_T_NF(不影响标志寄存器)
- Mov_Mem_To_T
- Mov_Reg_To_T
- Add_T_T
- 可能会有Save_Eflag
- Mov_T_To_Reg

为什么拆的这么细?

- add eax, ebx => mov T, [ebp+4*3] add [ebp], T ?
- 所有的指令不管简单与否，运算都使用临时寄存器，都要有与临时寄存器传输的过程。
- 管理好数据流向以方便后面优化，中间代码越简单越不容易出错

复杂指令拆解

- push eax=>sub esp, 4 + mov [esp], eax
- call 00401000=> push NextAddr + jmp 00401000

拆不了的怎么办?

- 很多奇葩的指令DAL, DDA, BT....，专门为它们创建中间代码

中间代码优化

- 与传统复写传播的区别：
- Mov_Reg_To_T16/8优化为 Mov_Reg_To_T32
- EAX,AX,AL,AH都支持
- 临时变量没有太大意义，所以要控制寄存器压力
- SAVE_Eflag/LoadEflag的移除
- 窥孔优化，常量传播？暂时没有做

寄存器分配

- 有使用高低8位的，不能被分配给EDI，ESI
- 在翻译拆解指令的时候会记录每一个T的生命周期，如果生命周期结束，它占用的寄存器就可以重新被分配。
- 易失寄存器
- EAX， ECX， EDX， EBX， ESP（栈），EBP（基址），EDI， ESI
- EBP做基址的教训，本来调试就复杂！

代码生成

- 块头布置好栈数据，块尾返回给外界下一个EIP与辅助地址（链块用的）返回原因：正常，异常，断点，自修改，其它
- Gen_Mov_T_To_Reg...
- Gen_Calc_T_T
- 把T设置到对应的ModRm位即可
- 针对每一个中间代码，根据X86指令格式生成机器码。

TLB

- 内存读写太频繁了，越简单越好，内存操作成功后会将真实地址与虚拟地址写入TLB
- 缓存了6个页，读GetMemData8/16/32.写SetMemData8/16/32.
- 操作内存的时候仅简单对比一下指针即可。
- 改进：最多18个页，绑定寄存器，可将TLB失效降低到原来的60%
- 需要与代码自己修改配合
- 对内存保护操作的时候，调整TLB

TLB进一步优化

- CS, DS, ES,的基址为0，99%下不会执行POP CS/DS/ES，意味着不会被修改
- 在访问内存中对段基址的计算代码在生成的时候可以懒惰式的省掉，FS例外
- 在TLB地址匹配的代码生成上加入分支预测前缀（大部分的情况是TLB是命中的）

代码自修改

- 传统的解决方案：把已翻译的内存页做标记，如果发现写入数据的页被标记过了，把翻译块删除，清除对应TLB。默认代码自修改不常见。
- 在一堆有重叠区域的数据里这些块并且清除，有一定的计算量。
- 块地址范围有重叠冗余：do{} while();
- 在病毒与壳里有大量的代码自修改，传统的方法非常非常的慢。

启发式

- 代码自修改级别（与TLB配合）：
  - 在翻译块所占的页，但写的地址没有被翻译
  - 在翻译块所占的页，写的地址被翻译过了（如何快速知道）
  - 在翻译块所占的页，正在写自己当前的块
- 脱壳速度较传统方法提高了500倍

被污染的寄存器内存

- 发生代码自修改或内存访问异常，跳出重新翻译执行。
- 简单指令在翻译的时候可以避免，把Mov_T_To_Reg放到最后。
- 复杂指令，需要做影子操作，一直到最后才把内存写操作提交上去。指令不多比如：PUSAD， REP MOVS

支持X64

- 在X64上翻译X86指令
- 共用翻译中间代码，个别中间代码生成重写，寄存器分配与优化简单
- 优点：寄存器无压力，全部映射虚拟寄存器，每个寄存器都有低8位

```
//T is temp register
//S is Special register( in memory operation )
//+---------------v---+---+---+---+---+---+---+---+--+--+---+---+---+-----+---+---+
//|Real Registers v RAX|RCX|RDX|RBX|RSP|RBP|RSI|RDI|R8|R9|R10|R11|R12| R13 |R14|R15 |
//|_______________v____|___|___|___|___|___|___|___|__|__|___|___|___|_____|___|____|
//|VM   Registers v EAX|ECX|EDX|EBX|   |EBP|ESI|EDI|T |S |T  |T  |ESP| BASE|T  |T   |
//+---------------v---+---+---+---+---+---+---+---+--+--+---+---+---+-----+---+---+
```

- Mov_Reg_To_T/Mov_Reg_To_T大部分情况下可以不生成代码

X64的优势

- mov [eax*4+ebx+20], ecx =>
  - lea r15d, [eax*4+ebx+20]
  - mov r14d ecx
  - Mov_T_To_Mem
- add eax, ebx => add eax, ebx
- sub esp, ecx => sub r12d, ecx
- sub eax, 4 => mov r15d, 4 sub eax, r15d

X64的问题

- 指令集非常复杂
- 高8位寄存器ah, ch, dh, bh不能与R8-R15出现在同一指令中，需要技巧。
- 有些X86的指令在X64下没有，DDA
- 寄存器映射 xchag eax, ecx
- 块头块尾函数调用，加载与保存映射寄存器，出入块频繁影响性能。

动态翻译的优缺点

- 循环多的时候，性能高
- 在分支（小指令块）多，但够不成循环，加密壳中乱跳（三条指令一个跳转），性能非常差，在X64上尤为明显
- 严重的代码自修改，性能会非常差
- 可控不好，跑起来刹不住

热路径分析

- 在跳转的时候做目标地址的记录，超过三次（构成循环），进入二进制动态翻译。
- 大部分代码自修改不会在循环内，特别严重级别的代码自修改编写难度大，一般是手工编写

高精度控制

- 如何支持断点（执行断点，指令断点）
- 如何将病毒代码在翻译块内死循环后停下

效果

Windows XP, CPU: P8700 2.5G, Memory:2G
程序原大小52M;加壳后： 28M;壳名UPX1.01

(s)	解释器	翻译未优化	翻译优化	静态
1	44.172	4.688	2.891	1.360
2	51.453	4.687	2.859	1.359
3	49.453	4.656	2.844	1.359
平均	48.359	4.677	2.865	1.359

为什么不使用QEMU

- 进程里有多个虚拟机实例
- 兼容解释器的断点机制
- 运行在WINDOWS平台
- QEMU兼容太多平台在性能上会有取舍
- 在专有平台上，优化可以做的更好

X86下MMX映射

- MMX不支持16位与8位粒度的数据传输，需要技巧。
- 比不使用提高大概5%
- 难度大
- 放弃!

传说中的超块?

- 不支持，由于生命周期很短，业务上没有太大的提升