



VMProtect的一次奇妙之旅

何潇潇@永信至诚





CONTENTS

VMProtect是什么

VMProtect背后的原理

还原VMProtect的方法

演示还原

总结





VMProtect是什么?

VMProtect是世界上最强大的二进制代码保护软件之一，是由俄罗斯人开发的，至今没有人公开声称对其完全破解。



PC保护壳现状：
难度低：压缩壳 UPX,ASPACK...
难度中：保护壳 ASProtect,EXECryptor...
难度高：虚拟机 VMProtect,Themida



VMProtect是什么?

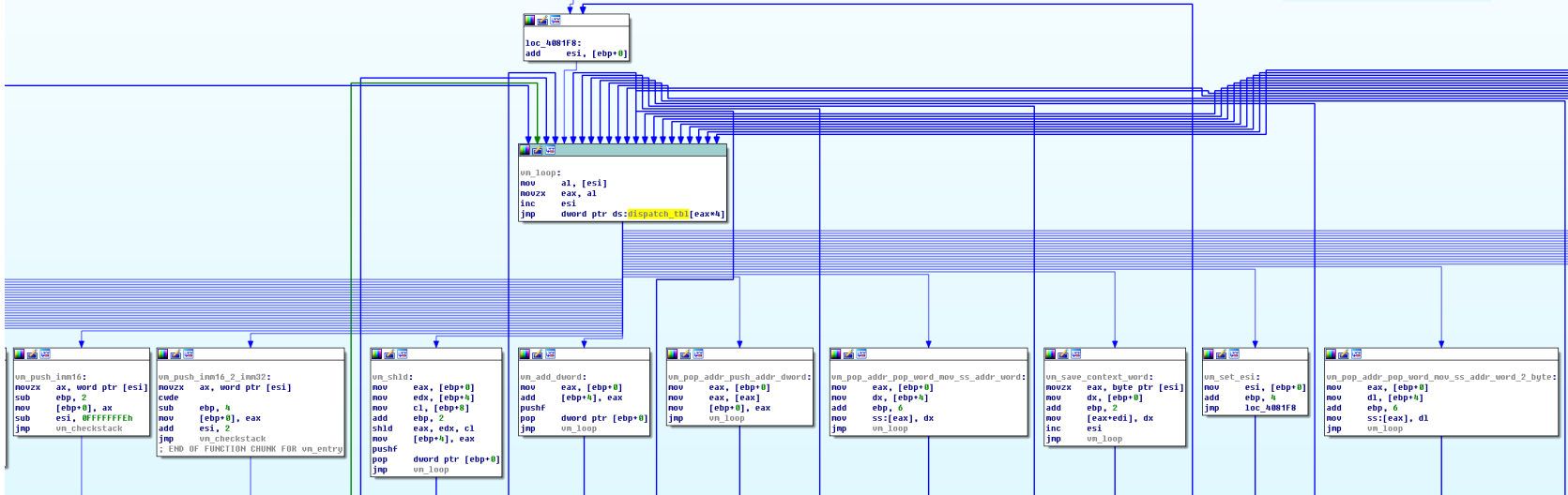


从本质来讲, VMProtect是



```
va_entry proc near
arg_0= dword ptr 4
; FUNCTION CHUNK AT 00401005 SIZE 00000026 BYTES
; FUNCTION CHUNK AT 00400100 SIZE 00000006 BYTES
push ebx
pushf
push esi
push ebx
push ebp
push edx
push edi
push eax
push ecx
push dword ptr ds:[_MemoryProtect]
push 0 ; delta
mov esi, [esp+30h] ; vndata
mov ebp, esp
sub esp, 0C0h
mov edi, esp ; va_context 16个格子的开始地址
```


```
vm_exit:
mov esp, ebp
pop ebx
pop ebx
pop ecx
pop eax
pop edi
pop edx
pop ebp
pop ebx
pop esi
popf
pop ebx
retn
```





I 相关术语

➤ Intel指令寄存器(Register) 和 VM指令寄存器(VMReg)



Intel	VMProtect
Eax	vEax
Ecx	vEcX
Edx	vEdx
Ebx	vEbx
Esp	vEsp
Ebp	vEbp
Esi	vEsi
Edi	vEdi
Eflags	vEfx
	vEix
	vErX
	vEtX
	vEmX
	vEex

VMProtect一共有14个寄存器，但是用16个格子(slot)存放它们，有多的2个格子可以理解成自由寄存器，最终扩展成16个寄存器。



I 相关术语

➤ Intel指令(Intel Instruction) 和 VM指令(VMRecord)

Intel指令就是Intel的汇编指令，
比如

```
add eax,ecx  
xor  eax,eax ...
```



在VMProtect的世界里面，指令是由
VMRecord组成的，比如

```
vm_push_imm32          0x1  
vm_get_context_dword  slot_offset
```

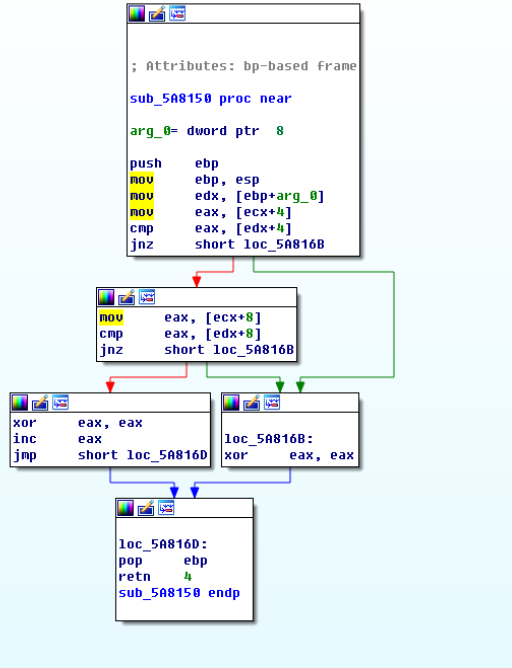
Intel指令编译生成对应的一组**VMRecord**，
比如 `mov ecx,eax;`

```
vm_get_context_dword    vEax  
vm_save_context_dword  vEcx
```



相关术语

Intel函数基本块(BasicBlock) 和 VM基本块(VMBlock)



BasicBlock是组成Intel函数控制流图的基本单位，在VMPProtect里面,VMBlock和它一一对应的，只不过VMBlock是VMRecord的载体。而且基本块与基本块之间的关系(也就是控制流图CFG),也在VMBlock之间一一对应。

```

1078DC1F:4572: vm_save_context_deord 2C
1078DC1A:4576: vm_push_lambda 60E48FDC
1078DC19:4576: vm_add_deord 4
1078DC18:4572: vm_save_context_deord 14
1078DC17:4580: vm_save_context_deord 24
1078DC16:4564: vm_save_context_deord 08
1078DC15:4560: vm_save_context_deord 14
1078DC14:4556: vm_save_context_deord 10
1078DC13:4560: vm_push_esp
1078DC12:4560: vm_prop_addr_push_esp_addr_deord
1078DC11:4556: vm_save_context_deord 08
1078DC10:4560: vm_push_esp
1078DC0F:4560: vm_prop_addr_push_esp_addr_deord
1078DC0E:4560: vm_not_not_and_deord 4
1078DC0D:4564: vm_save_context_deord 34
1078DC0C:4560: vm_push_lambda 5C7F5A8A
1078DC07:4560: vm_not_not_and_deord
1078DC06:4564: vm_save_context_deord 30
1078DC05:4560: vm_push_lambda 4380A7F9
1078DC00:4564: vm_get_context_deord 00
1078DBFF:4564: vm_not_not_and_deord
1078DBFE:4560: vm_save_context_deord 1C
1078DBFD:4560: vm_not_not_and_deord
1078DBFC:4556: vm_save_context_deord 0C
1078DBFB:4552: vm_save_context_deord 00
1078DBFA:4548: vm_save_context_deord 0C
1078DBF9:4544: vm_save_context_deord 28
1078DBF8:4540: vm_save_context_deord 30
1078DBF7:4536: vm_save_context_deord 1C
1078DBF6:4532: vm_save_context_deord 34
1078DBF5:4528: vm_save_context_deord 04
1078DBF4:4524: vm_save_context_deord 3C
-----
1078DBEF:4528: vm_push_lambda 2F4DF0FF
1078DBEA:4532: vm_push_lambda 20890802
1078DBE5:4536: vm_push_lambda 17011404
1078DBE4:4536: vm_add_deord
1078DBE3:4532: vm_save_context_deord 20
1078DBE2:4536: vm_not_not_and_deord
1078DBE1:4536: vm_add_deord
1078DBE0:4532: vm_save_context_deord 18
1078DBDF:4532: vm_push_lambda 8ED002
1078DBDE:4530: vm_save_context_deord 20
1078DBDD:4524: vm_save_context_deord 04
1078DBDC:4528: vm_get_context_deord 04
1078DBDB:4532: vm_push_esp
1078DBDA:4532: vm_prop_addr_push_esp_addr_deord
1078DBD9:4532: vm_not_not_and_deord
1078DBD8:4528: vm_save_context_deord 20
1078DBD7:4532: vm_push_lambda 83001770
1078DBD6:4536: vm_get_context_deord 04
1078DBD5:4536: vm_not_not_and_deord
1078DBD4:4532: vm_save_context_deord 20
1078DBD3:4532: vm_not_not_and_deord
1078DBD2:4528: vm_save_context_deord 20
1078DBD1:4524: vm_save_context_deord 20
-----
1078DBBD:4528: vm_get_context_deord 34
1078DBBC:4532: vm_get_context_deord 1C
1078DBBB:4526: vm_get_context_deord 08
1078DBBA:4540: vm_get_context_deord 30
1078DBB9:4544: vm_get_context_deord 14
1078DBB8:4548: vm_get_context_deord 10
1078DBB7:4552: vm_get_context_deord 0C
1078DBB6:4556: vm_get_context_deord 14
1078DBB5:4560: vm_get_context_deord 34
1078DBB4:4564: vm_get_context_deord 30
1078DBB3:4568: vm_get_context_deord 20
1078DBB2:4572: vm_get_context_deord 24
1078DBAD:4576: vm_push_lambda 7187033
1078DBAC:4576: vm_add_deord
1078DBAB:4572: vm_save_context_deord 18
1078DBAA:4576: vm_get_context_deord 2C
1078DBA9:4580: vm_get_context_deord 20
1078DBA8:4576: vm_jmp

```




VMProtect背后的原理



8/27/16 2:30PM



VMProtect背后的原理

逻辑门运算

VMProtect是通过NOR(或非门)和ADD(加法门)来实现intel指令的等价运算。

$NOR(a,b) = NOT(OR(a,b)) = AND(NOT(a),NOT(b))$

有了NOR的操作，就很容易表示其他的运算,例如:

$NOT(a) = NOR(a,a)$

$AND(a,b) = NOR(NOT(a),NOT(b)) = NOR(NOR(a,a),NOR(b,b))$

$OR(a,b) = NOR(NOR(a,b),NOR(a,b))$

$XOR(a,b) = NOR(NOR(a,b),NOR(NOR(a,a),NOR(b,b)))$

$SUB(a,b) = NOR(ADD(NOR(a,a),b),ADD(NOR(a,a),b))$

比如VMRecord来表示and eax,ecx

```
vm_get_context_dword    vEcx
vm_get_context_dword    vEcx
vm_not_not_and_dword    vEcx
vm_save_context_dword   vEax
vm_get_context_dword    vEax
vm_get_context_dword    vEax
vm_not_not_and_dword    vEcx
vm_save_context_dword   vEcx
vm_not_not_and_dword    vEfx
vm_save_context_dword   vEfx
vm_save_context_dword   vEax
```



VMProtect背后的原理

映射

指令集可以理解成线性空间，寄存器就是空间的基，寄存器个数也就是空间的维数。指令集中的指令，可以理解成算子，比如intel里面的add,xor,or等

在这里,intel指令集空间维度是9，VMProtect的是16，所以注定这2个空间不同构。

从intel指令到VMProtect指令的变换 f ，是同态变换，也就是这种变换没有逆变换，从理论上面证明了不存在完全还原方法。



VMProtect背后的原理

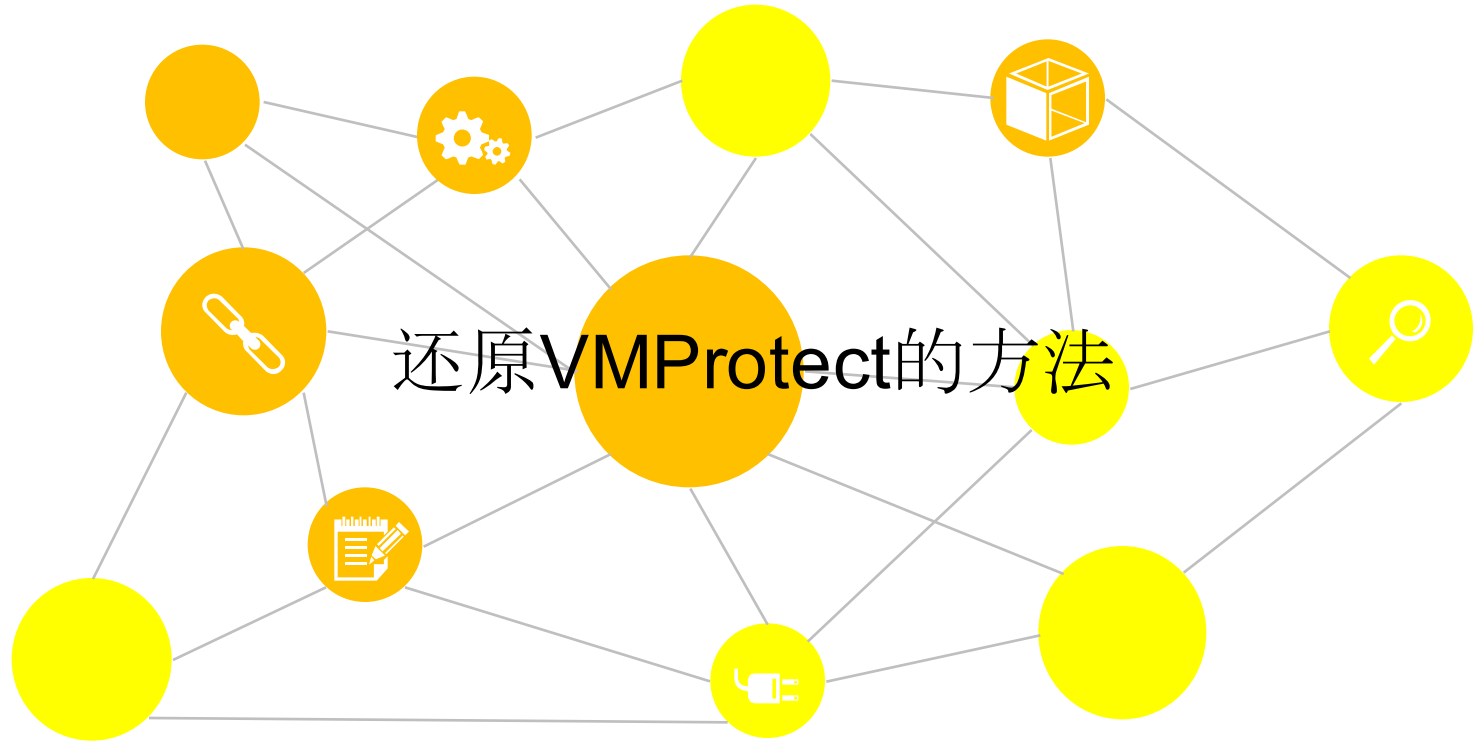
具体原因是，VMProtect的寄存器16个格子里面，任何时刻都有2个格子是自由的(其实是3个,因为vEex是垃圾寄存器,也是自由的)。VMProtect当一个寄存器发生变化的时候，它不会把新的值保存在原来的那个格子里面，它会从空闲格子里面取一个出来保存新值。比如vEax开始存放在1号格子里面，经过一系列运算vEax值发生变化，需要更新，这时会从空闲格子里面随机取出一个比如(2号格子)用来存放vEax,原来的1号格子就会进入空闲池子里了，正是这种特性，造成了Intel指令到VM指令之间不可能是一一对应的关系，因为操作数也就是VMProtect用到的格子时刻都在随机变化，只有VMProtect自己知道对应关系，除了它本身，第三者很难知道。



VMProtect背后的原理

```
vm_get_context_dword    vEcx  
vm_get_context_dword    vEcx  
vm_not_not_and_dword  
vm_save_context_dword   vEex  
vm_get_context_dword    vEax  
vm_get_context_dword    vEax  
vm_not_not_and_dword  
vm_save_context_dword   vEex  
vm_not_not_and_dword  
vm_save_context_dword   vEfx  
vm_save_context_dword   vEax
```

还是and eax,ecx，通过变换f，生成了上面的VMRecords
比如vEax开始保存在1号格子里面，经过and运算以后，保存最后结果的时候，从空闲格子中随机取一个出来(比如2号)，最终vEax从1号格子转移到2号格子里面，而1号格子变成了空闲格子。





还原VMProtect的方法



基于数据流的还原方法

是通过动态监控堆栈机的执行,获取其每一步的执行的指令,和操作数.然后根据最终的结果进行溯源,找到其指令的内在联系.



优点

比较简单,而且效率比较高.



缺点

并没有跑遍所有的指令(non-all-path),相当粗糙,不精确.



还原VMProtect的方法



基于控制流的静态还原方法

1.Control Flow的还原

2.Intel指令还原

Intel指令与VMRecord的对应

操作符(opcode)还原

操作数(operand)还原



Control Flow的还原

```
List    recoverControlFlow(ulong uVMEntryCode)
{
    List    AllBlocks;
    List    Workspace;

    VMBlock*    pBlock = createVMBlock(uVMEntryCode);

    AllBlocks.push_back(pBlock);
    Workspace.push_back(pBlock);

    while(!Workspace.empty())                //当Workspace不空
    {
        VMBlock* pWork = Workspace.pop_front();

        for(int i=0;i<pWork->succnum;i++)
        {
            if(pWork->succ[i] not in AllBlocks)    //如果后继Block还没处理
            {
                VMBlock*    pSuccBlock = createVMBlock(pWork->succ[i]);

                AllBlocks.push_back(pBlock);
                Workspace.push_back(pBlock);
            }
        }
    }

    //返回最终的所有VMBlocks
    return AllBlocks;
}
```



Intel指令还原

Intel指令与VMRecord的对应

在一个VMBlock里面，哪些VMRecords对应原始的Intel指令，这是需要首先解决，因为虚拟机的本质是堆栈机，也就是说，当执行完一条Intel指令对应的VMRecords后，堆栈机的堆栈应该是平衡的。为此，这里给VMRecord加上一个字段，表示执行完后，相对于VMBlock入口出的堆栈偏移。通过观察这个堆栈的偏移的变化来确定。

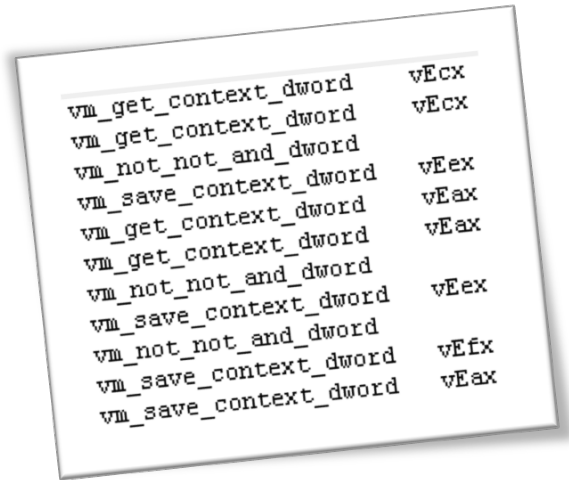
```
0040B244:-4: vm_get_context_dword 0C
0040B249:-8: vm_push_Imm32 DC0F9048
0040B24E:-12: vm_push_Imm32 FFBF4435
0040B24F:-16: vm_get_context_dword 04
0040B250:-20: vm_push_esp
0040B251:-18: vm_pop_addr_push_ss_addr_word
0040B256:-22: vm_push_Imm32 79014035
0040B258:-26: vm_push_Imm8_2_Imm32 0000000A
0040B259:-30: vm_push_esp
0040B25A:-30: vm_add_dword
0040B25B:-26: vm_save_context_dword 08
0040B25C:-18: vm_pop_addr_pop_dword_mov_ss_addr_dword
0040B25E:-16: vm_save_context_word 38
0040B25F:-16: vm_not_not_and_dword
0040B260:-12: vm_save_context_dword 00
0040B261:-16: vm_get_context_dword 20
0040B262:-16: vm_add_dword
0040B263:-12: vm_save_context_dword 10
0040B264:-12: vm_pop_addr_push_addr_dword
0040B265:-12: vm_add_dword
0040B266:-8: vm_save_context_dword 10 //Ch
0040B267:-12: vm_push_esp //esp
0040B26C:-16: vm_push_Imm32 D9DD674D
0040B271:-20: vm_push_Imm32 A9CBE502
0040B272:-24: vm_get_context_dword 34
0040B273:-28: vm_push_esp
0040B274:-26: vm_pop_addr_push_ss_addr_byte_2_word
0040B276:-28: vm_push_Imm8_2_Imm16 0039
0040B27B:-32: vm_push_Imm32 5674CB26
0040B27C:-36: vm_push_esp
0040B27E:-40: vm_push_Imm8_2_Imm32 00000008
0040B27F:-40: vm_add_dword
0040B280:-36: vm_save_context_dword 10
0040B281:-28: vm_pop_addr_pop_dword_mov_ss_addr_dword
0040B283:-26: vm_save_context_word 38
0040B285:-24: vm_save_context_word 10
0040B286:-24: vm_add_dword
0040B287:-20: vm_save_context_dword 38
0040B288:-24: vm_get_context_dword 20
0040B289:-24: vm_add_dword
0040B28A:-20: vm_save_context_dword 38
0040B28B:-20: vm_pop_addr_push_addr_dword
0040B28C:-20: vm_add_dword
0040B28D:-16: vm_save_context_dword 00 //4
0040B28E:-16: vm_add_dword
0040B28F:-12: vm_save_context_dword 00
0040B290:-12: vm_add_dword
0040B291:-8: vm_save_context_dword 38
0040B292:-8: vm_pop_addr_push_ss_addr_dword //[[esp+0Ch]
0040B293:-4: vm_save_context_dword 08
```

```
List divideVMRecords (VMBlock* pVMBlock)
{
    List IntelRecords;
    removeBlockEntryAndExit (pVMBlock);
    //删除VMBlock的入口和出口VMRecords 因为这个是VMProtect自己加上去的，不属于任何原始的Intel指令
    //从后往前枚举VMBlock中所有的VMRecords
    if (pVMBlock->VMRecords.size())
    {
        int CurIdx = pVMBlock->VMRecords.size() - 1; //指向最后的VMRecord
        while (CurIdx >= 0)
        {
            int stackOffset = pVMBlock->VMRecords[CurIdx].offset;
            int FirstIdx = FindFirstVMRecord (CurIdx, stackOffset);
            if (FirstIdx == -1) //表示没找到，也就是说该VMRecord对应Intel
            {
                IntelRecord* pNewIntel = createIntelRecord();
                pNewIntel->AddVMRecord (pVMBlock, CurIdx);
                CurIdx--; //
                IntelRecords.push_front (pNewIntel);
            }
            else
            {
                IntelRecord* pNewIntel = createIntelRecord();
                pNewIntel->AddVMRecords (pVMBlock, FirstIdx, CurIdx);
                CurIdx = FirstIdx - 1; //
                IntelRecords.push_front (pNewIntel);
            }
        }
    }
    return IntelRecords;
}
```



Intel指令还原

➤ 操作符(opcode)还原



操作符的还原，就是**模式识别**。首先是要建立识别库，也就是规则，这个需要相当的积累。

还是用前面的例子**and eax,ecx**，看到左边的**VMRecords**，通过模式识别，很容易就能分析出，这是一个**and**或者是**test**操作，操作数是**寄存器**，操作数大小是**dword**，通过具体后面分析，因为结果不是垃圾数据，确定是**and**操作。



Intel指令还原

➤ 操作数(operand)还原

只讨论寄存器操作数的还原，这是VMProtect里面最难的部分，之前的介绍了解到，VMProtect有**2个自由寄存器**和**1个vEex垃圾寄存器**，导致再重新写入1个寄存器的时候，不是写在先前位置，而是从空闲里面找一个出来写入，这种情况在很多时候会带来很大麻烦。

对于二元操作，比如add,xor,and,or等，可以表示成 $result = lhs \text{ op } rhs$ 。这是一个典型的**三地址模式**，因为Intel的格式，这种指令在Intel下面其实是**两地址模式**，**result** 和 **lhs** 重合了。对于二元操作的情况，只要分析出源操作符，就能对应出目的操作数是Intel下面的哪个寄存器。

```
vm_get_context_dword    vEcx
vm_get_context_dword    vEcx
vm_not_not_and_dword
vm_save_context_dword   vEex
vm_get_context_dword    vEax
vm_get_context_dword    vEax
vm_not_not_and_dword
vm_save_context_dword   vEex
vm_not_not_and_dword
vm_save_context_dword   vEfx
vm_save_context_dword   vEax
```



Intel指令还原



但是情况在`mov eax,ecx`这种时候，却变得很棘手。它对应的VMRecords是：

`vm_get_context_dword` `vEcx`

`vm_save_context_dwordslot_id`

此时我们并不知道save的slot_id对应的是哪个寄存器。



Intel指令还原

解决办法

待定寄存器法

通过vm_exit指令和vm_jmp指令，对应到真实寄存器环境，或者已知的VMBlock入口上面，从而找到所需的数据

空闲寄存器队列法

一个寄存器被重新写入后，原来的是被放入空闲寄存器里面，而且空闲队列就3个，再加上指令执行过程中，会经常从空闲中选择出格子写入，所以原来的位置有很大的可能(1/3)会从空闲队列中选择出来，从而捕捉到机会。

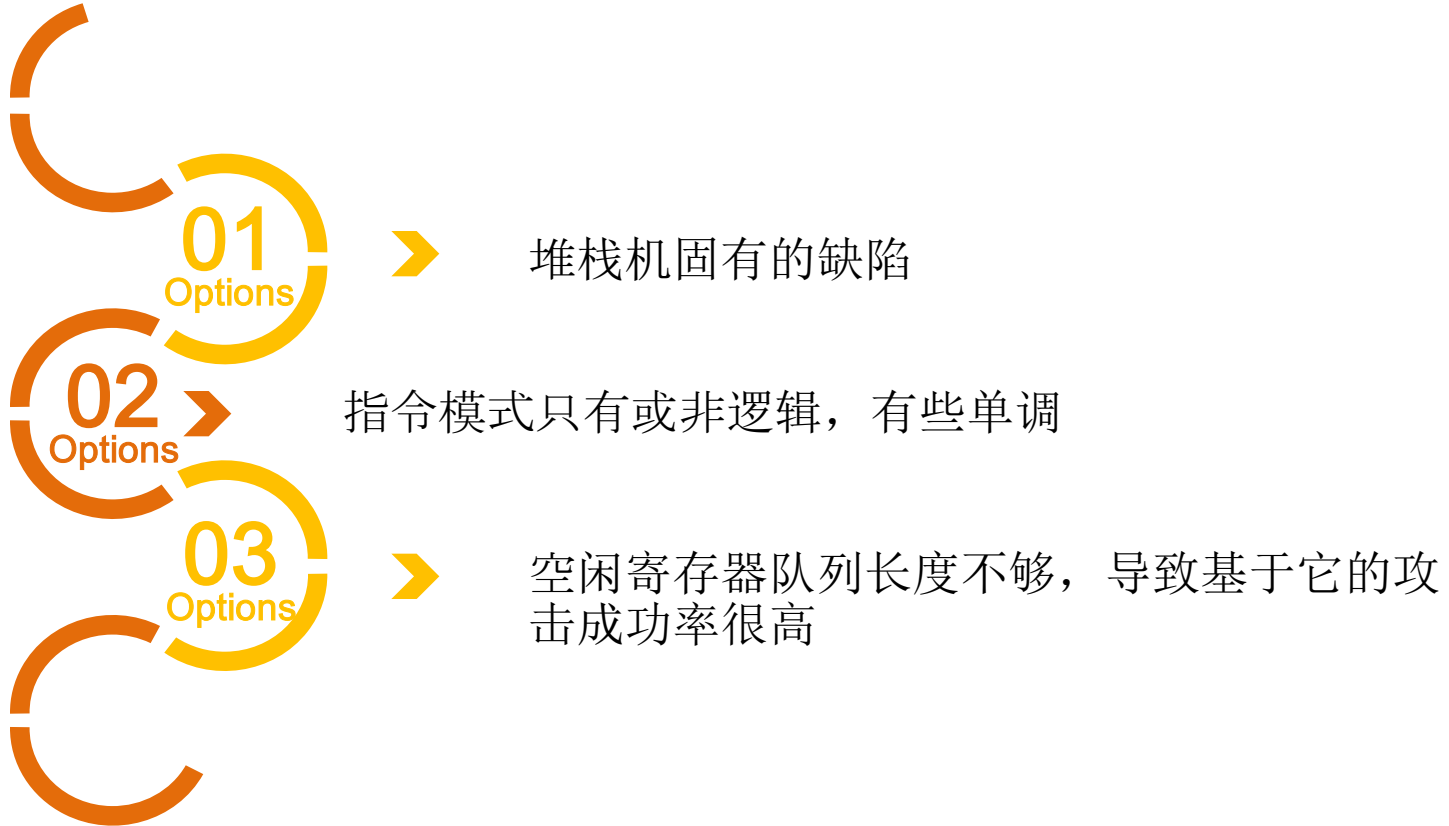
猜测法



演示还原，是基于WeChat的某个版本的添加联系人功能做演示的。



I 总结——VMProtect的不足





总结——我的工作

01

Treadstone保护引擎，杜绝上面的大部分缺点，还引入了寄存器置换引擎

02

目前正在基于LLVM框架，实现一个arm指令集类似虚拟保护引擎



THANKS