



VMProtect的逆向分析和静态还原

Bughoho



2009中国软件安全峰会



目录

- 一. **VMProtect逆向分析**
 - (一) **VMProtect**简单介绍
 - (二) **VMProtect**逆向分析
 - 1. 执行流程图全貌
 - 2. **VMProtect**的Handler
 - 3. **VMProtect**指令分类
 - 4. 逻辑运算指令
 - 5. 寄存器轮转
 - 6. 字节码加密和随机效验
 - 7. 阶段总结
- 二. **VMProtect静态跟踪**
 - (一) 虚拟执行特点
 - (二) 执行引擎的虚拟执行
 - (三) 分析条件跳转的两条出边
- 三. 字节码反编译
 - (一) 中间表示语言
 - (二) 指令化简和优化
 - (三) 转换汇编指令——树模式匹配
 - (四) 归类映射寄存器
 - (五) 转换汇编指令——动态规划
 - (六) 寄存器染色
 - 1. 基本块内的寄存器轮转
 - 2. 基本块间的寄存器轮转
 - 3. 寄存器的二义性问题
 - 4. 识别寄存器的二义性的步骤



VMProtect的逆向分析和静态还原

一. VMProtect逆向分析

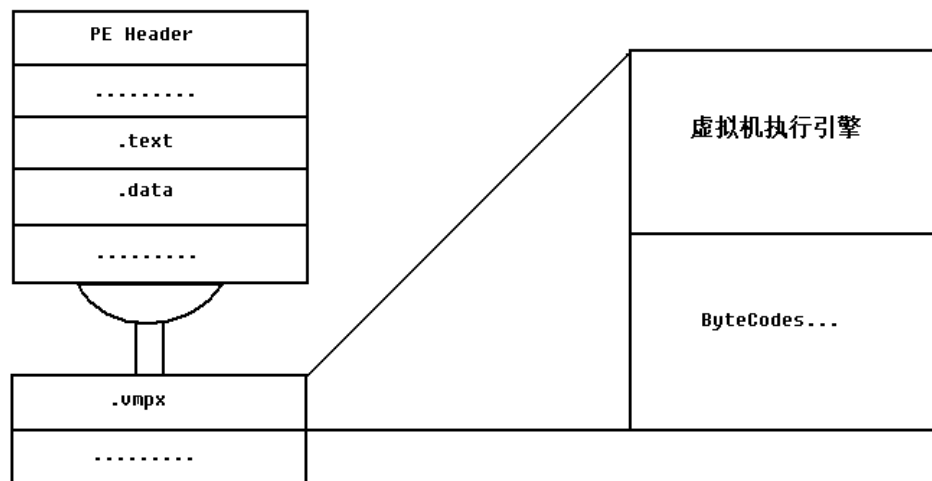
一. VMProtect逆向分析

(一) VMP简单介绍

VMProtect是一款虚拟机保护软件，是目前最为流行的保护壳之一，与其他类型保护软件不同的是，它使用的是虚拟机保护技术，侧重点在于保护所指定的函数，增加逆向分析的复杂度

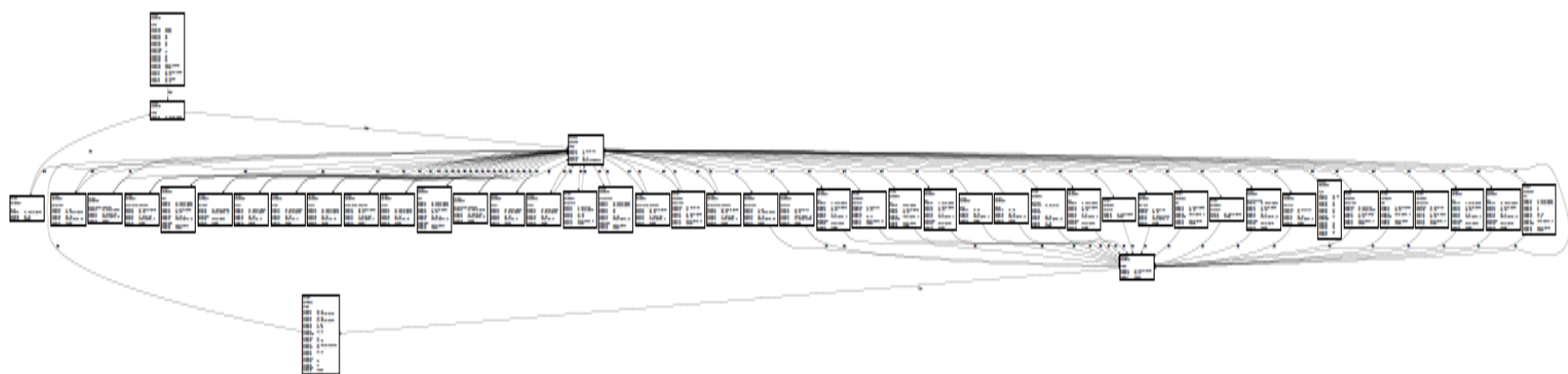
虚拟机保护特征

1. 将由编译器生成的本机代码 (Native Code) 转换成字节码 (Bytecode)
2. 将控制权交由虚拟机，由虚拟机来控制执行
3. 转换后的字节码非常难以阅读，增加了破解的复杂性



一. VMP逆向分析

(A) VMProtect Demo版本 流程图



graph.demo.pdf

虚拟机其实就是一个字节码解释器，它循环的读取指令并执行，并且它只有一个入口和一个出口(vm_exit)。通过静态分析，我们可以分析出整个执行引擎的完整代码

一. VMP逆向分析

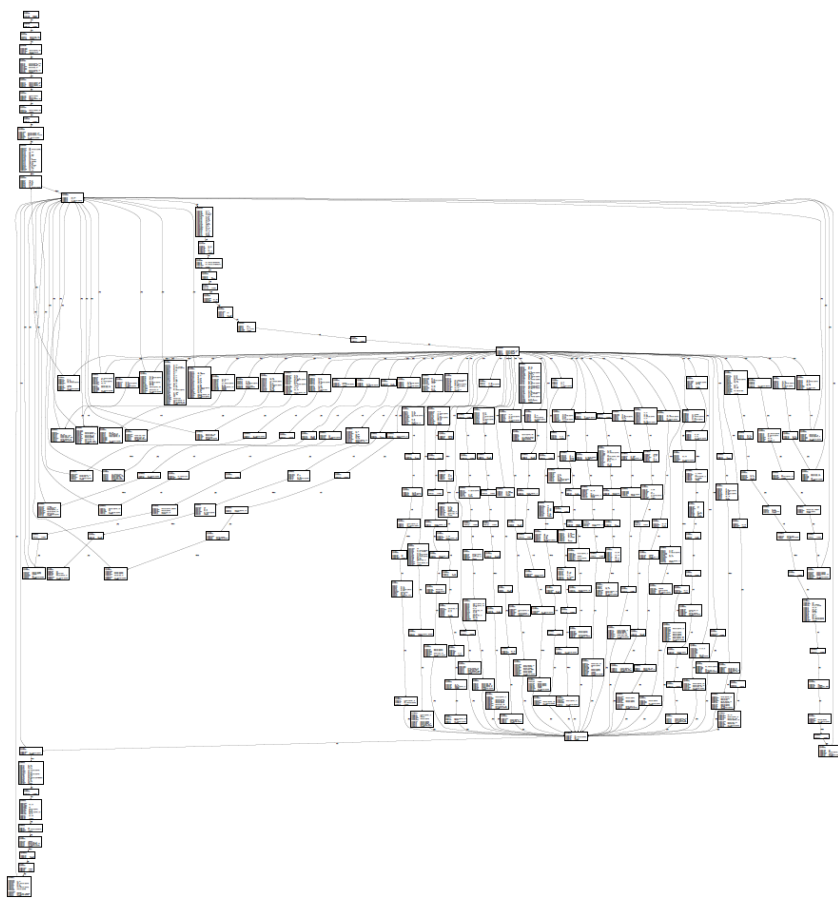
(B) Vmprotect Professional 版本 流程图

所有选项全部开启后的结果

✓ 虚假跳转

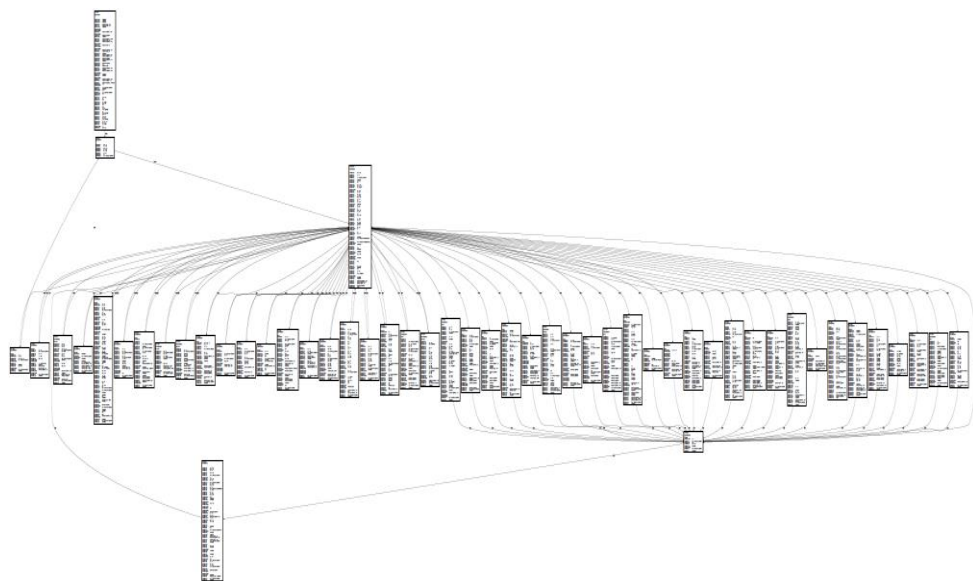
✓ 垃圾指令

大量的fake jcc（虚假跳转）和垃圾指令使原来十分简单的代码变得非常复杂



一. VMP逆向分析

(C) 清理了垃圾分支并做了伸直化处理后的结果



设定一些规则，将虚
假分支清除后，流程
图就跟原来一样清晰
了。如果再清除掉垃
圾指令，几乎就跟
demo版本的代码一
样😊

经过清理之后，新的
流程图分析起来难度
将会降低很多。



一. VMP逆向分析

2.VMP的Handler

VMP是基于堆栈的虚拟机（Stack-Based VirtualMachine）

虚拟机指令并不显式的使用某个参数，而是先将参数压入堆栈，然后直接从堆栈中读取

表达式：

Add eax,ecx

可以翻译为：

Push ecx

Push eax

Add

Pop eax

0040D043	>	8B45 00	mov	eax, dword ptr [ebp]
0040D046	.	0145 04	add	dword ptr [ebp+4], eax
0040D049	.	9C	pushfd	
0040D04A	.	8F45 00	pop	dword ptr [ebp]
0040D04D	..	E9 51000000	jmp	0040D0A3

无论push进来的是谁，Add指令总是读取并弹出堆栈中存放的值，然后Add算出结果再压入堆栈。





一. VMP逆向分析

3.VMP指令分类

汇编指令在转换到虚拟机的指令体系的过程中，被最大限度的化简和归类了，VMP中的指令大体分五类：

1. 算术运算和移位运算
2. 堆栈操作
3. 内存操作
4. 系统相关（无法模拟指令）
5. 逻辑运算

其中最复杂的是逻辑运算指令



一. VMP逆向分析

4.逻辑运算指令

Vmp中的逻辑运算只有一条指令:nor。这个指令在电路门中叫NOR门，它由三条指令组成，即not not and，与NAND门一样，用它可以模拟not and xor or这四条逻辑运算指令

转换公式：

$P(a,b) = \sim a \ \& \ \sim b$

$\text{not}(a) = P(a,a)$

$\text{and}(a,b) = P(P(a,a),P(b,b))$

$\text{or}(a,b) = P(P(a,b),P(a,b))$

$\text{xor}(a,b) = P(P(P(a,a),P(b,b)),P(a,b))$

0040D743	>	8B45 00	mov	eax, dword ptr [ebp]
0040D746	.	8B55 04	mov	edx, dword ptr [ebp+4]
0040D749	.	F7D0	not	eax
0040D74B	.	F7D2	not	edx
0040D74D	.	21D0	and	eax, edx
0040D74F	.	8945 04	mov	dword ptr [ebp+4], eax
0040D752	.	9C	pushfd	
0040D753	.	8F45 00	pop	dword ptr [ebp]
0040D756	^	E9 48F9FFFF	jmp	0040D0A3

NOR门

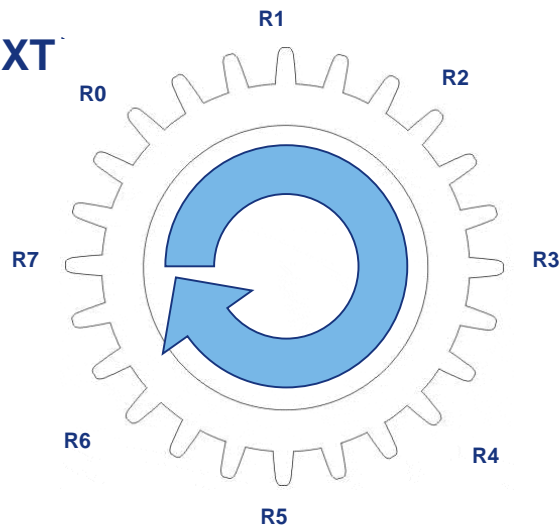
一.VMP逆向分析

5.寄存器轮转

VMP将所有寄存器都存放在了堆栈的结构中（**VM_CONTEXT**结构中的每一项代表一个寄存器或者临时变量。

但在运行过程中，其中的项所映射的真实寄存器都是不固定的，可以把它比作一个齿轮，每做完一个动作，部分项的映射就互换了一下位置，或者执行完一段指令，齿轮就按不固定的方向和度数转动一下，然后全部的项映射就改变了。

VMP在生成字节码的过程中，维护了一份结构中每一项所映射的真实寄存器，但这只存在于编译过程，而在运行时是没有明确的信息的。这直接导致了分析和识别的难度。

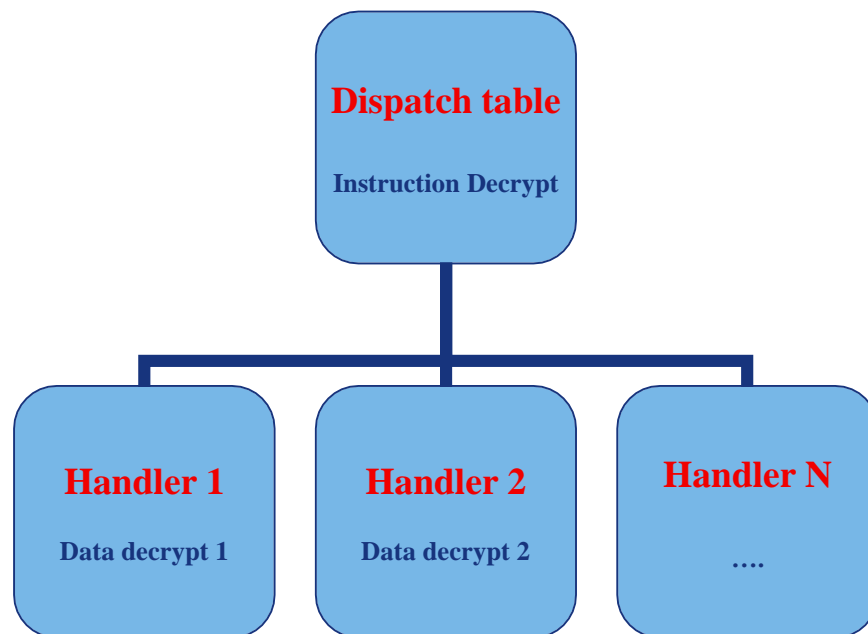


一.VMP逆向分析

6.字节码加密和随机效验

VMP把解码算法分布到了**Dispatch**和每个**Handler**中，只有在取指令和取数据时才会解密，而每个解码的算法也都是不同的，并且它的**Seed**每次解密都会变化的。

要写出字节码的逆算法不是不可以，但是复杂度太高，有些得不偿失。所以如果想要修改数据，还是使用**HOOK**的方式比较轻松。



一.VMP逆向分析

但是HOOK的方式得解决代码检测的问题，VMP注册版除了会加密字节码以外，还会随机对一段代码做检测，如果有错将无法运行。

VMP注册版中有一条叫指令（calchash），就是用来做检测的。VMP会在编译好的字节码中加一些自己的指令，每次执行都会随机对一段代码生成一个Hash结果，然后与另一个随机的数相加，结果必须为0，否则就会出错。如果要爆破或者修改VMP的代码，还需要处理这个过程。

```
0040D660 add     ch, bh
0040D662 bswap   ecx
0040D665 sub     eax, eax
0040D667 sub     esp, FCh
0040D66A pushfd
0040D66B pushfd
0040D66C movzx   ecx, byte ptr [edx]
0040D66F pushfd
0040D670 stc
0040D671 jmp     0040D28Ch
0040D67C cmp     ch, cl
0040D68E shl     eax, 04h
0040D691 cmp     si, cx
0040D694 call    0040F7B3h
0040F7B3 test     si, bx
0040F7B6 add     eax, ecx
0040F7B8 pushad
0040F7B9 mov     ecx, eax
0040F7BB jmp     0040E269h
0040E269 bt      dx, cx
0040E26D push    D368eF8Fh
0040E272 and     ecx, F0000000h
0040E278 call    0040E566h
0040E27B push    dword ptr [esp]
0040E27C lea     esp, dword ptr [esp+3Ch]
0040E27D je      0040D5ABh
0040E27F jmp     0040D57Dh
0040D57D push    edx
0040D57E setnle  dh
0040D581 mov     edx, F88CB978h
0040D586 shld    dx, ax, 00000006h
0040D58B mov     edx, ecx
0040D58D stc
0040D58E shr     edx, 18h
0040D591 cmp     cl, ah
0040D593 push    5A0B7EC1h
0040D596 xor     eax, edx
0040D59A cll
0040D59B add     dl, bh
0040D59D dec     edx
0040D59E xor     eax, ecx
0040D5A0 setp    dh
0040D5A3 mov     edx, dword ptr [esp+04h]
0040D5A7 lea     esp, dword ptr [esp+08h]
0040D5AB pushad
0040D5AC call    0040F11Dh
0040F11D jmp     0040D263h
0040D263 inc     edx
0040D264 push    799FC3DBh
0040D269 mov     word ptr [esp], 9DD3h
0040D26F jmp     0040F564h
0040F564 pushfd
0040F565 dec     dword ptr [ebp+00h]
0040F568 pushfd
0040F569 lea     esp, dword ptr [esp+30h]
0040F56D jnz     0040D66Ah
0040F573 jmp     0040D2E7h
0040D2E7 push    edx
0040D2E8 push    edx
0040D2E9 mov     dword ptr [ebp+00h], eax
0040D2EC pushad
0040D2ED push    31C89B79h
0040D2F2 lea     esp, dword ptr [esp+2Ch]
0040D2F6 jmp     0040EACCh
```



一. VMP逆向分析

7.阶段总结:

VMP作者的原则

最简单的正向设计

导致最困难的逆向分析





VMProtect的逆向分析和静态还原

二.VMProtect静态跟踪

二. VMProtect静态跟踪

(一) 虚拟执行特点

1. 虚拟执行是静态分析与动态执行的一个折中办法
2. 虚拟执行时对内存访问做了一定的控制以防止出现异常
 - a. 允许读写静态内存与堆栈内存
 - b. 忽略其他内存访问与修改

解决了异常问题后，就可以从入口点一直虚拟执行到出口了

00401000	\$	B8 11110000	mov	eax, 1111
00401005	.	3D 11110000	cmp	eax, 1111
0040100A	~	75 05	jnz	short 00401011
0040100C	.	B8 22220000	mov	eax, 2222
00401011	>	C3	retn	

记录得到的字节码日志



二. VMProtect静态跟踪

(二) 执行引擎的虚拟执行

分析虚拟机的一般传统方法

- 找到关键位置
- 动态执行并使用记录断点记录数据
- 输出记录日志

优点:

- 寻找关键位置时间相对较短

缺点:

- 多路径时只能走其中一条路径
- 分析多个虚拟机时要重复做相同的工作

虚拟执行方法

- 虚拟执行代码
- 根据已分析字节码灵活控制代码流程
- 输出记录日志

优点:

- 虚拟执行不会对系统造成任何伤害
- 完整的字节码流程

缺点:

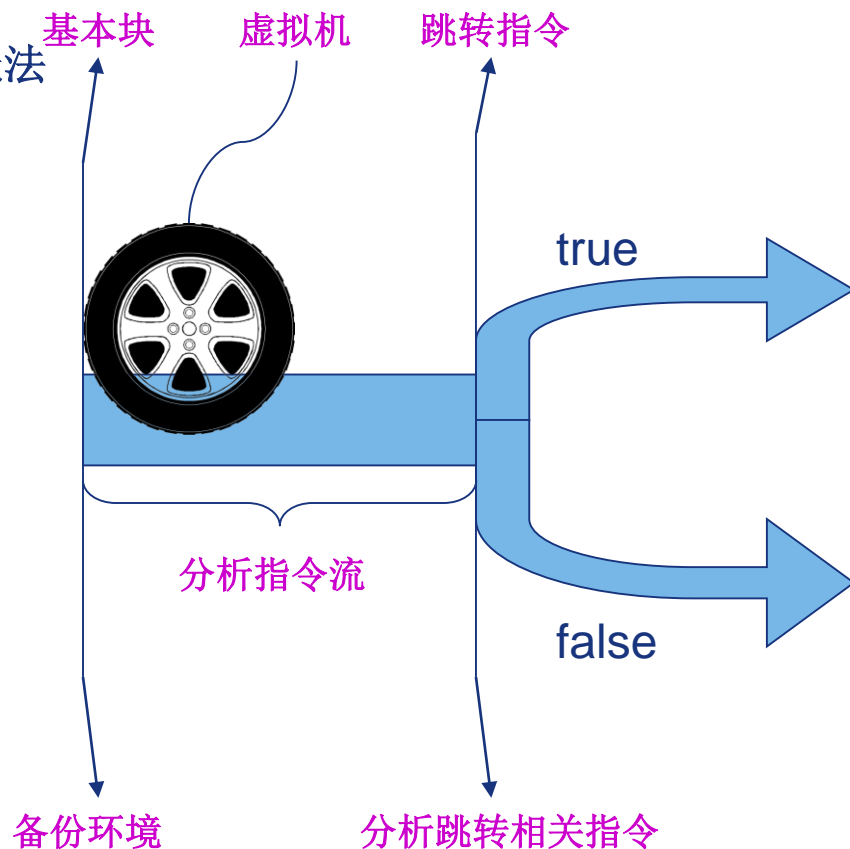
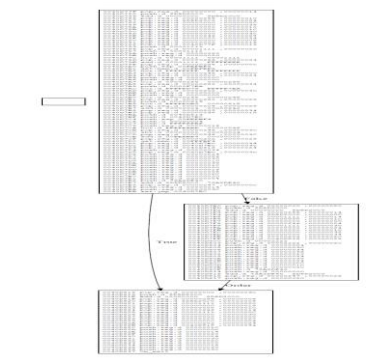
- 指令正确但操作数的值不可靠
- 复杂度较高, 开发时间较久

二. VMProtect静态跟踪

(三) 分析条件跳转的两条出边

因为虚拟执行是不依赖运行时信息的，所以它无法判断应该走哪一条，必须把两条边都走过一遍。

- ✓ 在基本块（**BasicBlock**）执行前备份运行时环境
- ✓ 执行到跳转处分析指令流，获得修改路径关键点
- ✓ 退回基本块起始位置
- ✓ 重新执行并控制路径





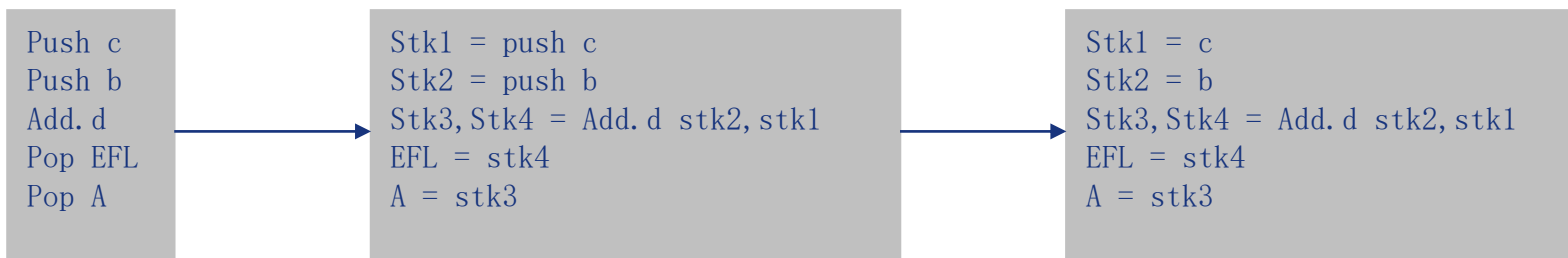
VMProtect的逆向分析和静态还原

三.字节码反编译

三.字节码反编译

(一) 中间表示语言

VMP的**Handler**只能算是低级中间语言，缺少一些例如数据依赖、流程走向等信息，还不满足反编译的条件。需要将其转换为包含更多信息的高级中间语言形式。



- ✓ 去掉了对堆栈的依赖，转为直接关联变量
- ✓ 表达式被转换成了**SSA**（静态单赋值）形式，方便对指令做优化处理

三.字节码反编译

(二) 指令化简和优化

✓ 常数收缩

```
Push 1A2FBCA0  
Push F80499D8  
Add. d  
Pop EFL
```

```
Push 12345678
```

✓ 活跃变量分析

```
Stk0 = Ctx24  
Stk1 = Ctx10  
Stk2, EFL1 = Add. d Stk0, Stk1  
Ctx28 = EFL1  
Ctx30 = Stk2  
Stk3 = Ctx30  
Stk4 = 1111  
Stk5, EFL2 = Add. d Stk3, Stk4  
Ctx28 = EFL2  
Ctx34 = Stk5
```

去除中间变量后

```
Ctx30, Ctx28 = Add. d Ctx24, Ctx10  
Ctx34, Ctx28 = Add. d Ctx30, 1111
```

去除无用变量后

```
Ctx30 = Add. d Ctx24, Ctx10  
Ctx34, Ctx28 = Add. d Ctx30, 1111
```

✓ 删除无关代码

VMP在生成的字节码中夹杂了一些自己的指令流，这些指令与原汇编代码没有任何关系，且对还原分析没有任何好处，只会起到干扰的作用。需要根据特征制定一些规则来识别这些垃圾指令。

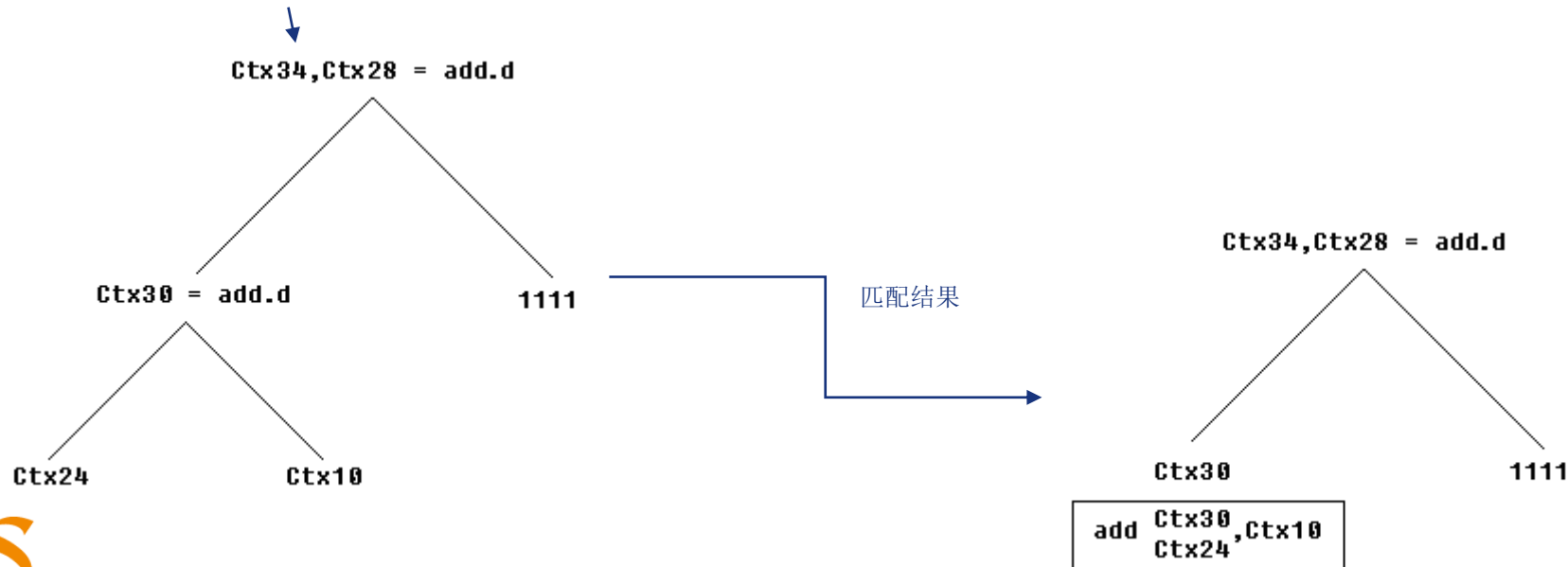


三.字节码反编译

(三) 转换汇编指令——树模式匹配

- ✓ 文本表达转换为树形表达
- ✓ 收集转换规则 这是最麻烦的一个过程，需要分析VMP将汇编生成字节码的特征来收集将字节码逆向转换回去的规则，这是一个不得不做的体力活。
- ✓ 使用匹配规则迭代匹配汇编指令

Ctx30 = Add.d Ctx24,Ctx10
Ctx34,Ctx28 = Add Ctx30,1111



三.字节码反编译

(四) 归类映射寄存器

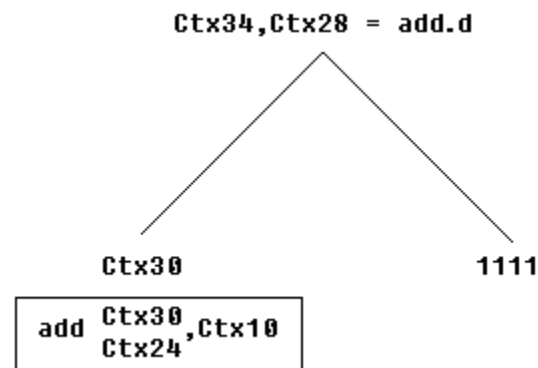
经过迭代后的最终结果是这样：

```
Add Ctx30(Ctx24),Ctx10
```

```
Add Ctx34(Ctx30),1111
```

虽然已经转换为汇编指令，但是还无法确定寄存器到底是哪一个，以目前所知的信息也的确无法判断。不过，我们可以尽可能的确定一些信息，以供后面的分析参考。

在转换规则中，预先明确定义了Add指令的第一个参数与结果是同一个寄存器（其他指令也差不多，类似xchg的指令除外），所以可以推理得到，在指定的区间内，Ctx34、Ctx30、Ctx24是同一个寄存器。这样后面在做专门针对寄存器识别的分析时，就可以一下确定这四个寄存器所映射的寄存器了。



匹配结果



字节码反编译

(五) 转换汇编指令——动态规划

考虑右边两段指令

所谓动态规划，通俗的讲就是制定一些规则，根据实际情况来选择最终匹配结果。

这里的意思是对每一个匹配规则设一个权值，使用计算后值最大的那个匹配规则来进行转换。

第二段的情况有些特殊，其中两条指令唯一的不同只有目标操作数。Add指令认为目标操作数与源操作数1相同，而lea指令则无此限制。当出现权值一样的情况时，可以同时作为结果，在识别出寄存器后，再根据实际情况来匹配规则，在这两个指令中选出更像的那一个。

```
mov eax,dword ptr [edi+0x100]
add edi,100
```

其中第一条里面包含了第二条的指令
第一条的权值应该设得更高

```
add edi,100
Lea ebx,[edi+100]
```

两条指令仅仅是目标寄存器不同
两条指令的权值应该相等



（六）寄存器染色

要识别前面所代表的寄存器，要从以下几个方面进行分析：

1. 初始化虚拟机时各项所映射的寄存器
2. 根据汇编转换规则映射或者结束映射某项到某寄存器
3. 退出虚拟机时通过弹出各项时确定各项最终映射的寄存器

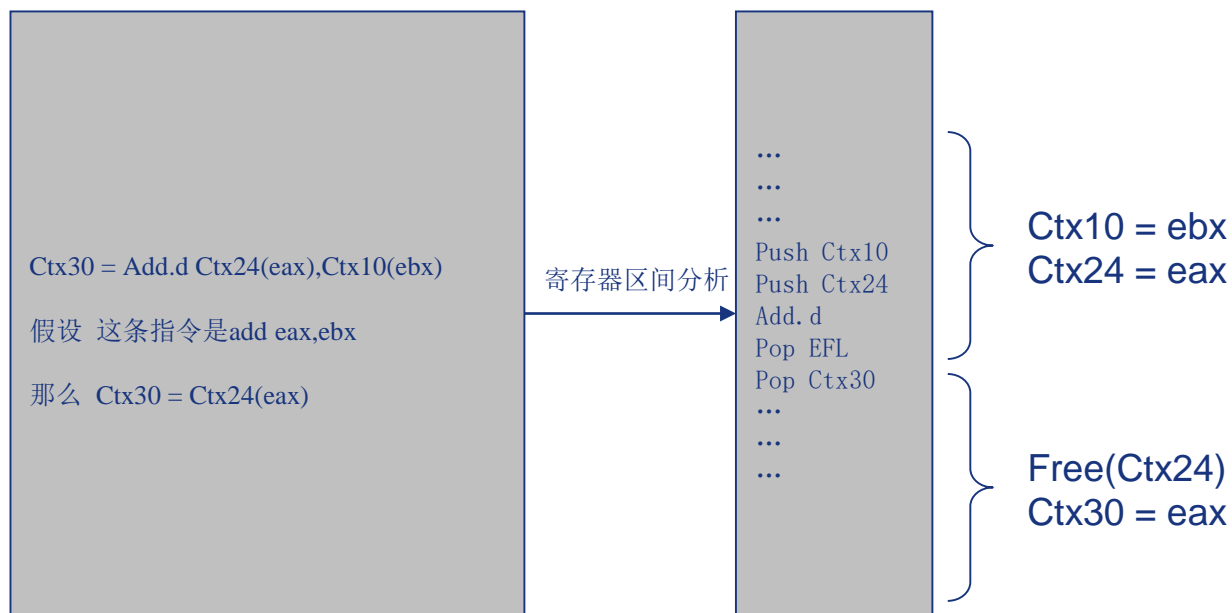
从这三方面可以大体推理出各项所映射的寄存器

但仅仅是这样的话，只有在没有跳转指令的字节码中，成功率才最高。
因为还得考虑寄存器轮转

(六) 寄存器染色

1. 基本块内的寄存器轮转

基本块内的寄存器轮转比较容易简单，只要转换规则正确，就可以识别出寄存器





（六）寄存器染色

2.基本块间的寄存器轮转

在执行**set.jmp**之前，将**Context**中所有位置的值都临时存放到了堆栈中，跳向目标地址后又再全部把它弹出到不同位置中去，这样就完成了一次轮转。

它比基本块内的寄存器轮转更麻烦，因为其中涉及到了二义性的问题

```
push. reg. d 00000024
...
push. reg. d 00000020
push. reg. d 00000004
push. reg. d 00000014
push. reg. d 0000003C
set. jmp
pop. reg. d 00000020
...
pop. reg. d 0000002C
pop. reg. d 00000028
```



(六) 寄存器染色

3.寄存器的二义性问题

寄存器的二义性问题是一个很严肃的问题，因为如果不能正确的分析和处理，将会造成一子放错，满盘皆错的局面。
寄存器的二义性由指令的二义性衍生出，要解决指令的二义性，需要先解决寄存器的二义性问题。



Push\Pop的二义性

Push\Pop在VMP中存在一种二义性，即传值与传引用。



传值

当pop指令的作用是传值时，表示将源项中的值放到目标项中去，所映射寄存器不变



传引用

当pop指令的作用是传引用时，不但将值从源项放到目标项中去，且目标项所映射的寄存器也将被覆盖。

传值是一般情况，即汇编指令的push\pop指令对，传引用是特殊情况，如寄存器轮换等。



Add与Lea等其他指令的二义性

这两条指令广义的讲也是Push\Pop的二义性。



为Add指令时，pop指令的含义为传引用



为Lea指令时，pop指令的含义为传值

普通Push\Pop指令对：

Push Ctx20(edx)

Pop Ctx24(esi)

寄存器轮换：

Push Ctx20(edx)

Pop Ctx24(edx)





(六) 寄存器染色

4. 识别寄存器的二义性的步骤

1. 根据转换规则尽可能确定一些特性，缩小可能的寄存器范围
2. 无法判断的寄存器将其加入此项的可能性列表中，并建立起传递链表
3. 退出虚拟机时可知各项真实的寄存器，排除其他可能的寄存器
4. 确定寄存器后，再重新排除有二义性的指令





VMProtect的逆向分析和静态还原

参考资料:

《高级编译器设计与实现》（Steven S.Muchnick）

《编译原理及实践》（Louden, K.C）

《decompilers and beyond white paper》（Ilfak Guilfanov）

[破解vmp程序的关键点（海风月影）](#)

以及其他众多互联网资料



感谢

- ❖ 看雪论坛
- ❖ 组委会
- ❖ 参考资料
- ❖ 以及各位观众

www.phei.com.cn

Thank You !

