

# VMP 介绍 & 一个小例子

VMP (Virtual Machine Protection, 虚拟机加固) 是一种非常高级的混淆加固技术, 广泛应用于 Android 应用的 SO 层 (Native 层) 保护中。VMP 的核心思想是将原始的机器码翻译成一套自定义的“字节码”指令, 然后在运行时通过虚拟机解释执行这些字节码, 极大增加逆向分析难度。

## 一、如何应对 VMP, 还原程序原始逻辑?

要应对 VMP, 目标是逆推出字节码的执行逻辑和语义, 最终还原出原始程序的行为。

关键应对思路:

识别虚拟机入口点 (Dispatcher)

找到字节码执行的调度器, 即虚拟机解释器的主循环逻辑;

通常表现为大量的 switch 或 if-else 结构、jmp 表等。

识别字节码指令集和对应语义

每个虚拟指令在解释器中对应一个处理分支, 需要逐一分析它们实现的行为 (如加法、跳转、调用等);

可使用调试手段记录字节码和执行行为, 构建对应关系表。

追踪字节码流 (控制流还原)

跟踪程序运行过程中的字节码流, 并通过分析跳转逻辑重建控制流图 (CFG);

对于条件跳转指令, 需要观察其影响条件。

还原原始逻辑 (行为还原)

通过字节码的动态执行行为, 模拟或记录其逻辑;

可以写工具或脚本对虚拟机执行流进行“脱虚拟化”。

## 二、应对 VMP 的关键点

关键点 说明

Dispatcher 识别 是进入虚拟机的核心入口点, 定位后才能跟踪执行流程

指令语义分析 识别虚拟指令的操作逻辑, 是还原原始程序逻辑的关键

控制流重构 理解程序的逻辑流程结构

动态调试能力 需要配合调试器进行动态分析, 分析加密解密、虚拟执行流程

自动化辅助工具 编写辅助脚本/插件提高分析效率，如指令跟踪、模拟器

### 三、常见工具链

#### 1. 调试器/反调试绕过

GDB / LLDB：底层调试器

Xposed + Frida：进行动态 hook 与 bypass

IDA Pro / Hopper / Ghidra + Debugger Plugin：静态+动态结合

脱壳工具：可选地绕过加壳阶段，进入 VMP 主逻辑

#### 2. 跟踪分析工具

Frida + 自定义脚本：动态 hook 虚拟机执行函数，记录字节码和执行路径

Qiling / Unicorn Engine：模拟执行字节码流，进行动态行为分析

PIN Tool / DynamoRIO：插桩工具，可以收集执行路径信息

#### 3. 辅助工具

angr / Binary Ninja 插件：符号执行分析复杂流程

VMPTrace / UnVMP：专门用于对抗虚拟机加壳（部分开源或自制工具）

### 四、学习 VMP 逆向的路径

#### 1. 掌握基础逆向技能

熟悉 ARM 指令集和汇编

熟练使用 IDA Pro/Ghidra 进行静态分析

掌握 LLDB/GDB 调试技巧

学习脱壳、动态调试、反调试绕过

#### 2. 分析已有样本

找一些典型的 VMP 保护 SO 样本（可以找 Unity、腾讯、字节加固的样本）

自己写一个简易的 VM 保护器，用于理解 VMP 构造原理

#### 3. 阅读相关论文与技术博客

《Virtualization Obfuscators: A Threat for Reverse Engineering》论文

Github 上搜索关键词 “unvm”，“vmp trace”，研究现有工具源码

研究国内外 CTF 题目中涉及虚拟机加固的题解

#### 4. 进阶实践：脱虚拟化

编写指令跟踪脚本（如 Frida 脚本记录执行指令）

重建指令语义并写反汇编器/解释器

结合符号执行、模拟执行还原原始逻辑

#### 五、建议的练习方法

分析一个实际的 VMP 加固 SO（推荐使用旧版本样本）

手写一个最小 VM 保护器（比如 10 条虚拟指令）

用 Frida/Unicorn 模拟执行自己 VM 代码

写脚本提取并还原 VMP 字节码流

尝试自动重建控制流图（CFG）

=====sample=====

## 一个简单的 C 函数逻辑

编写一个自定义虚拟机（VMP）将其转换为虚拟字节码执行；

并提供对应的逆向分析思路，供你从虚拟机入口到字节码语义还原练习。

▣ Part 1: 最小 VMP 样本构建（源码）

#### 1. 原始 C 逻辑（原函数）：

C

```
int logic(int x, int y) {  
    int z = x * 2 + y;  
    if (z == 10) {  
        return 1;  
    } else {  
        return 0;  
    }  
}
```

```
}  
}
```

我们将这个函数转换成自定义字节码，并在运行时通过“虚拟机”解释执行。

## 2. 虚拟指令集设计（最简）

Opcode 指令名 含义

0x01LOAD 加载立即数到寄存器

0x02ADD 加法

0x03MUL乘法

0x04CMP比较两个寄存器

0x05JE 相等则跳转

0x06RET 返回

我们定义了寄存器：r0, r1, r2, r3（虚拟机内部数组实现）

## 3. 虚拟字节码样例

下面是将原始逻辑翻译成的字节码：

makefile

```
LOAD r0, x      ; 输入 x  
LOAD r1, y      ; 输入 y  
MUL  r0, 2      ; r0 = x * 2  
ADD  r0, r1     ; r0 = r0 + y  
CMP  r0, 10  
JE   LABEL_TRUE  
LOAD r2, 0  
RET  r2  
LABEL_TRUE:  
LOAD r2, 1  
RET  r2
```

转成机器可读的简易格式（以字节表示）：

c

```
unsigned char bytecode[] = {  
    0x01, 0x00, 0xFF, // LOAD r0, arg0  
    0x01, 0x01, 0xFE, // LOAD r1, arg1  
    0x03, 0x00, 0x02, // MUL r0, 2  
    0x02, 0x00, 0x01, // ADD r0, r1  
    0x04, 0x00, 0x0A, // CMP r0, 10  
    0x05, 0x0D,      // JE offset=0x0D  
    0x01, 0x02, 0x00, // LOAD r2, 0  
    0x06, 0x02,      // RET r2  
};
```

```

    0x01, 0x02, 0x01,    // LOAD r2, 1
    0x06, 0x02          // RET r2
};
(注: 0xFF 表示第一个参数, 0xFE 表示第二个参数)

```

#### 4. 虚拟机解释器代码 (C 实现):

```

#include <stdio.h>

int run_vm(unsigned char *code, int arg0, int arg1) {
    int r[4] = {0}; // 虚拟寄存器
    int ip = 0;     // 指令指针
    while (1) {
        unsigned char op = code[ip++];
        switch (op) {
            case 0x01: { // LOAD
                int reg = code[ip++];
                int val = code[ip++];
                if (val == 0xFF) val = arg0;
                if (val == 0xFE) val = arg1;
                r[reg] = val;
                break;
            }
            case 0x02: { // ADD
                int dst = code[ip++];
                int src = code[ip++];
                r[dst] += r[src];
                break;
            }
            case 0x03: { // MUL
                int dst = code[ip++];
                int imm = code[ip++];
                r[dst] *= imm;
                break;
            }
            case 0x04: { // CMP
                int reg = code[ip++];
                int imm = code[ip++];
                r[3] = (r[reg] == imm); // r3 做 flag
                break;
            }
            case 0x05: { // JE
                int offset = code[ip++];
                if (r[3]) ip = offset;
            }
        }
    }
}

```

```

        break;
    }
    case 0x06: { // RET
        int reg = code[ip++];
        return r[reg];
    }
    default:
        return -1;
}
}
}

```

## 📌 Part 2: 逆向分析教学步骤

🎯 目标：从 binary 中恢复出原始逻辑

📌 第一步：识别 Dispatcher (VM 执行入口)

run\_vm 函数就是 dispatcher，实际工程中你需要用 IDA 或动态调试找到它；

判断特征是：一个不断循环 + switch-case 类结构。

📌 第二步：动态调试执行过程

使用 GDB/Frida 动态 attach，打断点在 dispatcher 内部；

每次执行前打印当前字节码指令 (op) 和寄存器值；

记录这些值形成执行 trace。

📌 第三步：还原指令语义

逐个理解 opcode 的含义 (LOAD、ADD、MUL、CMP、RET)；

写一个映射表，比如：

python

```

INSTR_TABLE = {
    0x01: "LOAD",
    0x02: "ADD",
    0x03: "MUL",
    0x04: "CMP",
    0x05: "JE",
    0x06: "RET",
}

```

📌 第四步：还原字节码逻辑

拿到 bytecode 后，用 Python 写个脚本反汇编：

python

```
def disasm(bytecode):
    i = 0
    while i < len(bytecode):
        op = bytecode[i]
        if op == 0x01:
            print(f"{i:02X}: LOAD r{bytecode[i+1]}, {bytecode[i+2]}")
            i += 3
        elif op == 0x02:
            print(f"{i:02X}: ADD r{bytecode[i+1]}, r{bytecode[i+2]}")
            i += 3
        elif op == 0x03:
            print(f"{i:02X}: MUL r{bytecode[i+1]}, {bytecode[i+2]}")
            i += 3
        elif op == 0x04:
            print(f"{i:02X}: CMP r{bytecode[i+1]}, {bytecode[i+2]}")
            i += 3
        elif op == 0x05:
            print(f"{i:02X}: JE {bytecode[i+1]}")
            i += 2
        elif op == 0x06:
            print(f"{i:02X}: RET r{bytecode[i+1]}")
            i += 2
```

执行后，你将能看到对应的伪汇编，还原程序控制流。

## ▣ 第五步：重建逻辑流程图

根据跳转和条件指令（如 JE），构建基本块；

最终还原出原函数逻辑：

```
z = x*2 + y; if (z == 10) return 1; else return 0;
```

## ▣ Bonus：进阶挑战方向

增加堆栈指令（PUSH/POP）、CALL/RET；

隐藏指令调度结构（如 thread 上下文切换、代码拼接）；

使用 opaque predicate、指令混淆、垃圾字节等高级 VMP 策略。

=====

## 📁 项目结构说明:

bash

mini\_vmp\_project/

```
|—— src/
|   |—— main.c      # 主函数，调用虚拟机执行
|   |—— vm.c        # 虚拟机解释器实现
|—— disasm.py       # Python 脚本，反汇编 bytecode
```

💡 编译运行方式 (Linux)

bash

```
cd mini_vmp_project/src
```

```
gcc main.c vm.c -o vmtest
```

```
./vmtest
```

你会看到结果输出，如：

ini

Result = 0

你可尝试修改 `x, y` 的值，观察不同逻辑的返回变化，进一步用于逆向练习。