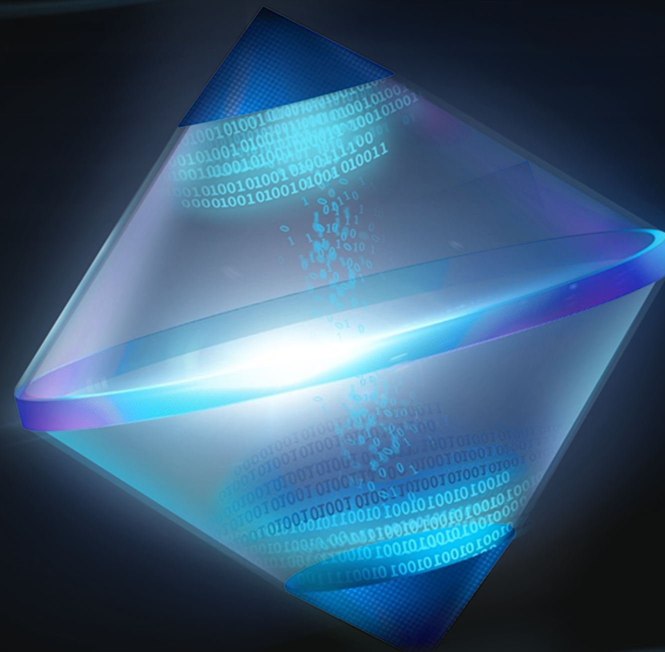
KCon

归源·智变
Returning to the source · Intelligent transformation

内窥镜-反混淆虚拟化加固的安卓程序



演讲人：余帘

目录 / CONTENTS

01 虚拟化加固
简介及研究背景

02 Rhino逆向
RHINO字节码的特殊案例

03 加固后的程序结构
更常见的虚拟化加固程序模式

04 恢复原始字节码
基于已知模式进行逆向

05 观点和总结
安卓程序加固趋势

虚拟化加固简介

虚拟化加固是指将原始程序中包含的代码指令"编译"成由一组特定的自定义指令组成的字节码，并在自定义的虚拟机上执行该字节码。

它是混淆手段中强度较高的一种，常用于：

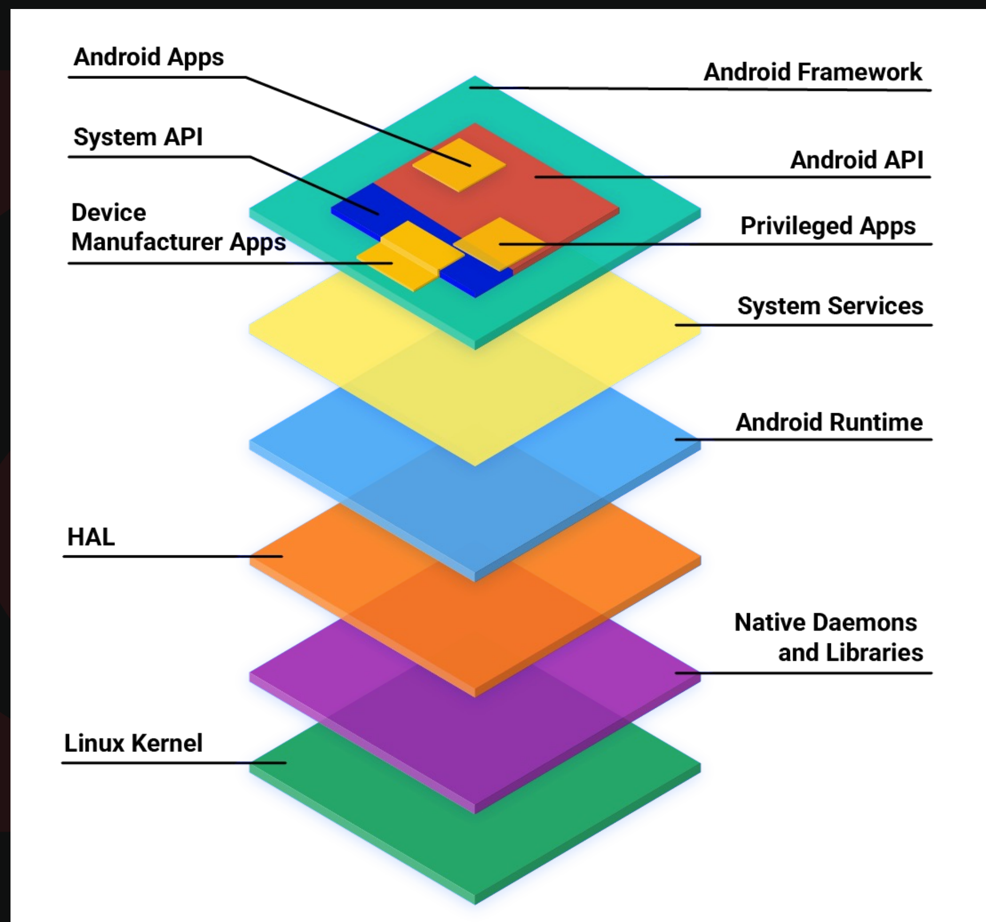
合法用途：反剽窃、防抄袭，保护软件知识产权等

非法用途：隐藏恶意行为，提高分析和执法难度

虚拟化加固安卓应用

近年来，安卓系统上使用虚拟化加固技术对应用进行混淆的案例持续增加。

虚拟化加固安卓应用与加固PC程序的区别在于，安卓存在多层次的架构，加固器普遍会利用Java Native Interface(JNI)进行跨层函数调用，而PC程序不存在这种特征。



研究背景

数月前，我们发现一个安卓恶意样本的主要逻辑被编译成字节码，在Rhino JS引擎上执行，且源码字段被故意置空了。这种类型的混淆不是严格意义上的vmp但有相似之处。我们通过分析Rhino引擎的解释执行过程，能够逆向还原出部分原始语义。

后续，我们又遭遇了多款在野的安卓恶意程序，它们使用了更为常见的虚拟化加固模式，并且加固器是闭源的。前述的逆向方法对这些程序不再适用，因此我们采用了一种更加通用的方法进行逆向，还原出混淆前的字节码

Interpreter

Custom Virtual Machine

KCon

归源 · 智变
Returning to the source · Intelligent transformation

目录 / CONTENTS

01 虚拟化加固
简介及研究背景

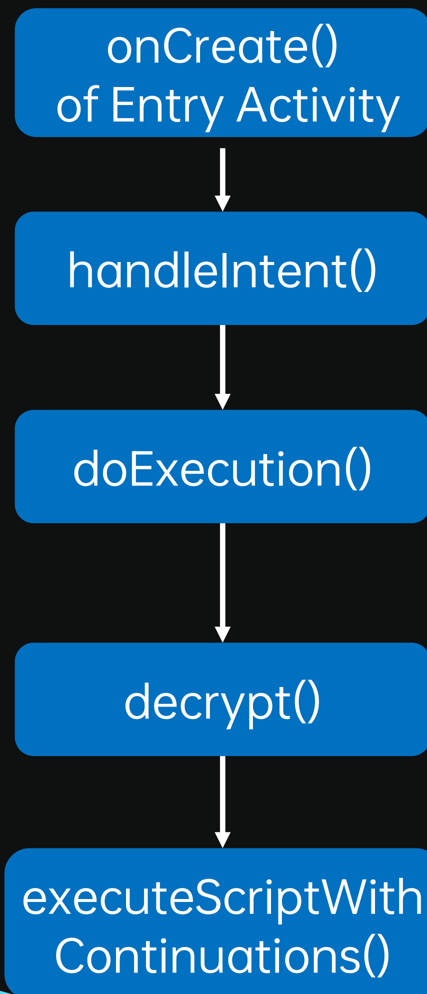
02 Rhino逆向
RHINO字节码的特殊案例

03 加固后的程序结构
更常见的虚拟化加固程序模式

04 恢复原始字节码
基于已知模式进行逆向

05 观点和总结
安卓程序加固趋势

Rhino字节码案例--JS on Android



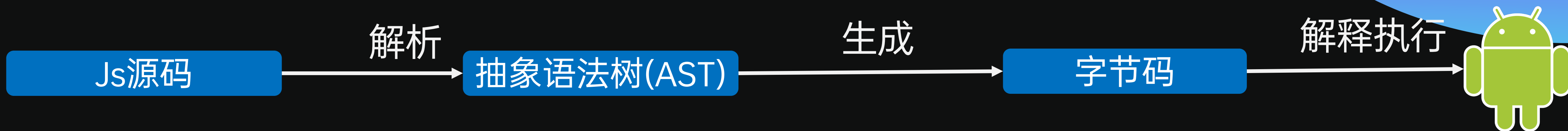

```java
class InterpretedFunction {
    InterpreterData idata;
    Interpreter.interpret(this);
    ...
}
class InterpreterData {
    byte[] itsICode;
    String[] itsStringTable;
    BigInteger[] itsBigIntTable;
    String encodedSource;
    ...
}
```

```java
public final class Interpreter{
    private static Object interpretLoop(...) {
        Object[] stack = frame.stack;
        int op = iCode[frame.pc++];
        switch (op) {
            case Icode_GENERATOR:
                ...
            case Token.YIELD:
                ...
        }
    }
}
```

这个恶意程序主Activity的onCreate()加载并执行一个InterpretedFunction对象，该对象只保存字节码而非源码，使用interpretLoop()中一个有大量分支的switch-case语句解释执行。

不像Dalvik字节码已有多款工具能够把它反编译成人类易读的java， 目前没有现成工具可以逆向Rhino字节码

Rhino字节码的生成




```
"foobar".split("").reverse().join("");
```

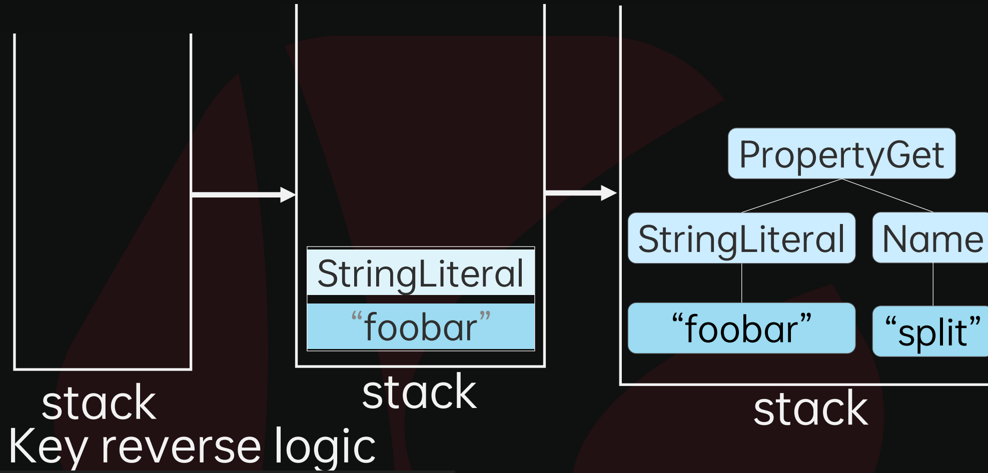
root
ExpressionStatement
FunctionCall
PropertyGet
StringLiteral
FunctionCall
Name
“”
PropertyGet
“join”
FunctionCall
Name
PropertyGet
StringLiteral
“reverse”
StringLiteral
Name
“”
“foobar”
“split”

Bytecode
-42
-16
-43
41
-33
38
-44
-16
-32
38
-45
-16
-45
41
-33

Icode_LINE
Icode_REG_STR_C0
STRING
Icode_REG_STR_C1
Icode_PROP_AND_THIS
Icode_REG_STR_C2
STRING
Icode_REG_IND_C1
CALL
Icode_REG_STR_C3
Icode_PROP_AND_THIS
Icode_REG_IND_C0
CALL
Icode_REG_STR1
Icode_PROP_AND_THIS
Icode_REG_STR1
STRING
Icode_REG_IND_C1
CALL
Icode_POP_RESULT
RETURN_RESULT

逆向-从字节码重建AST

-42	Icode_LINE
-16	Icode_REG_STR_C0
-43	STRING
-42	Icode_REG_STR_C1
-16	Icode_PROP_AND_THIS
-43	Icode_REG_STR_C2
41	STRING
-33	Icode_REG_IND_C1
38	CALL
-44	Icode_REG_STR_C3
-16	Icode_PROP_AND_THIS
-32	Icode_REG_IND_C0
38	CALL
-45	Icode_REG_STR1
-16	Icode_PROP_AND_THIS
-45	Icode_REG_STR1
41	STRING
-33	Icode_REG_IND_C1
-42	CALL
-16	Icode_POP_RESULT
-43	RETURN_RESULT

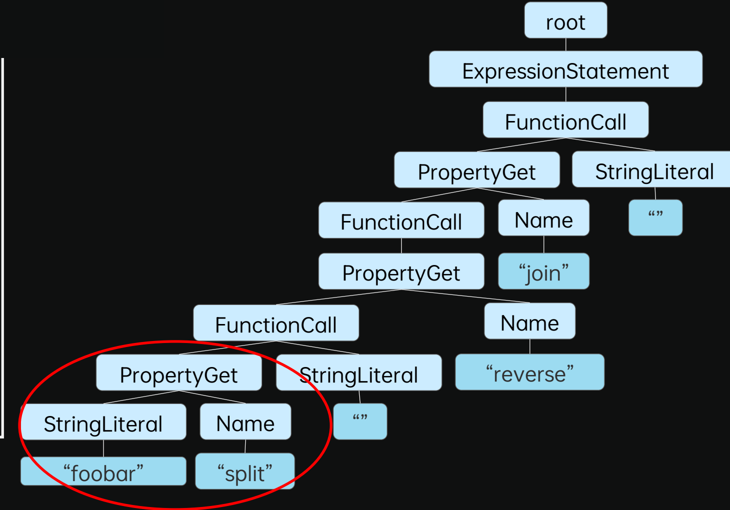


Key reverse logic

```
Token.STRING -> {
    val sl = StringLiteral(0)
    sl.value=stringReg
    sl.quoteCharacter='"'
    stack.push(sl)
}
```

itsStringTable

0	"foobar"
1	"split"
2	""



Rhino provides a toSource() method that transform an AST back into source code recursively ☺

"foobar".split("").reverse().join("");

```
Icode.Icode_PROP_AND_THIS -> {
    val func = FunctionCall()
    func.target = PropertyGet(stack.poll(), Name(0, stringReg))
    stack.push(func)
}
```

目录 / CONTENTS

01 虚拟化加固
简介及研究背景

02 Rhino逆向
RHINO字节码的特殊案例

03 加固后的程序结构
更常见的虚拟化加固程序模式

04 恢复原始字节码
基于已知模式进行逆向

05 观点和总结
安卓程序加固趋势

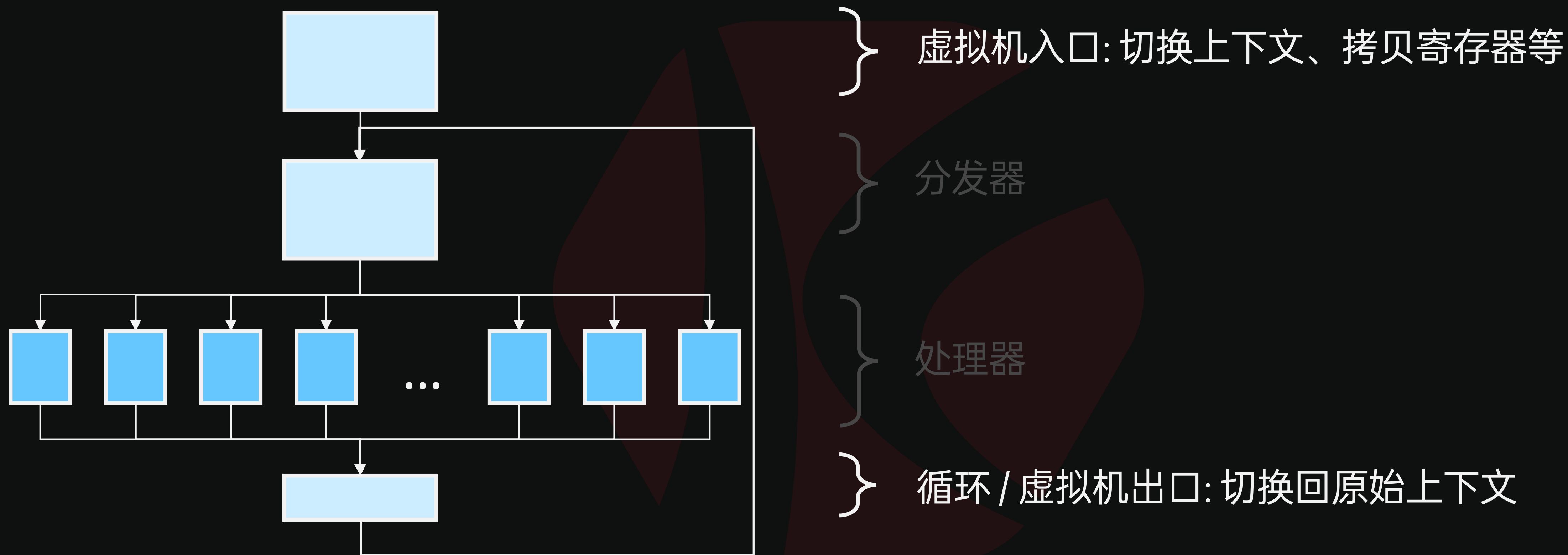
更常见的虚拟化加固场景

上述Case的加固方式属于特例。更常见的虚拟化加固场景存在如下特征：

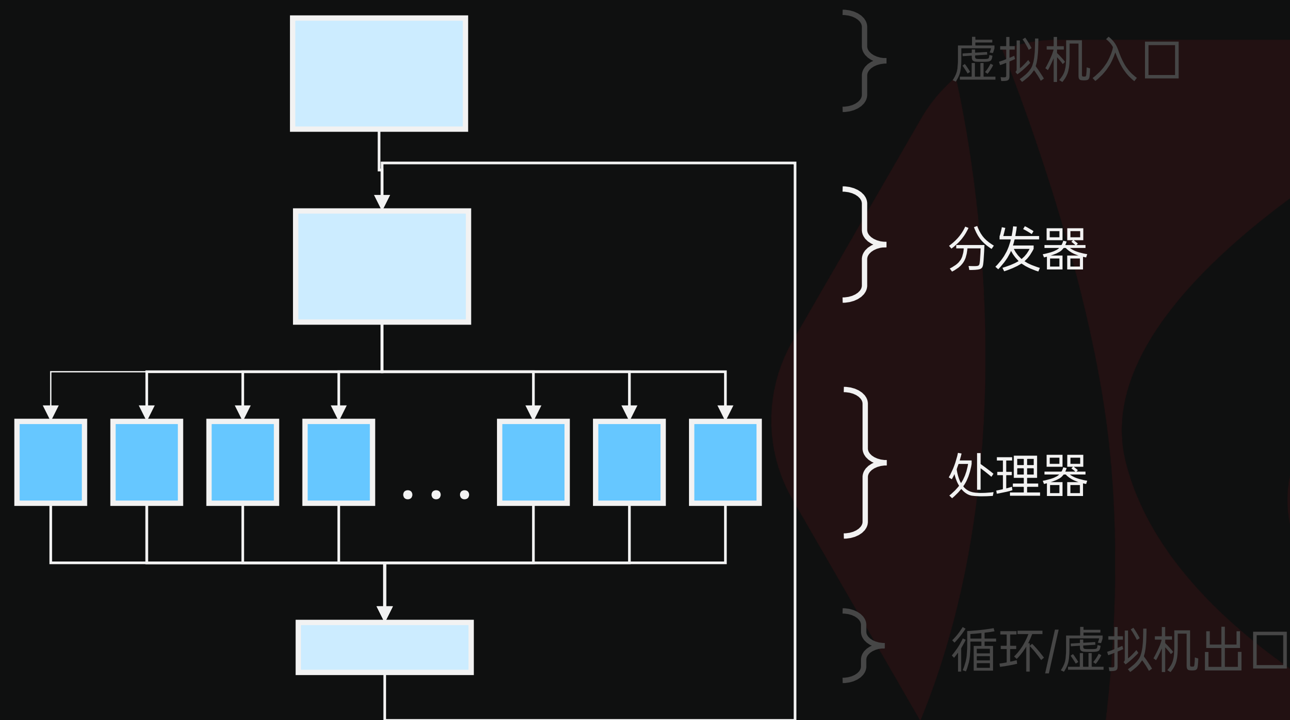
- 加固器闭源
- 虚拟机在native库中实现，并以JNI形式调用
- 进行针对特定应用的随机化

上述特征导致了更高的逆向难度

加固程序结构: 虚拟机入口/出口

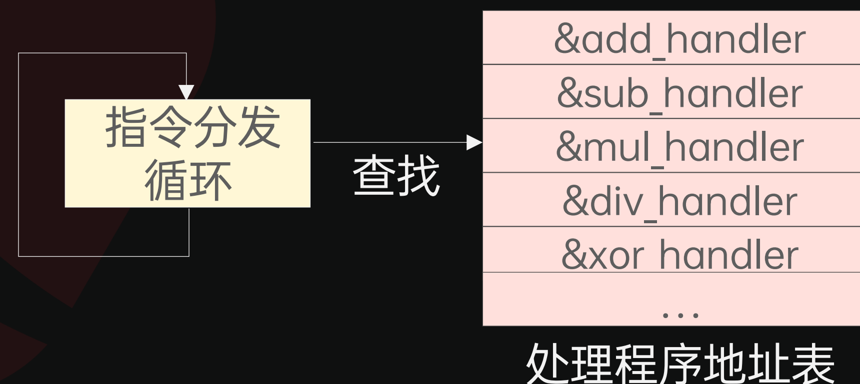


加固程序结构:分发器和处理程序



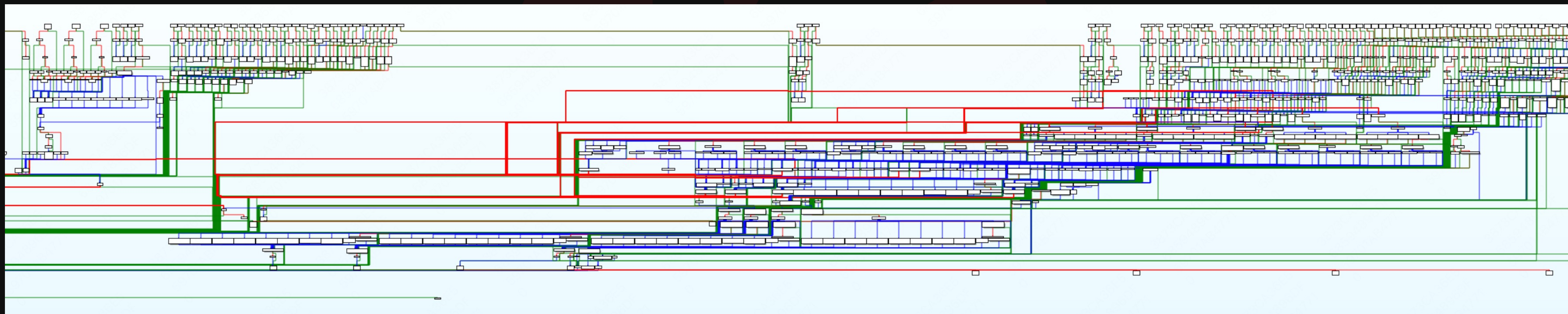
指令分发循环

- 获取一条虚拟指令，并对其进行解码
- 在处理程序地址表中进行查找
- 调用处理程序



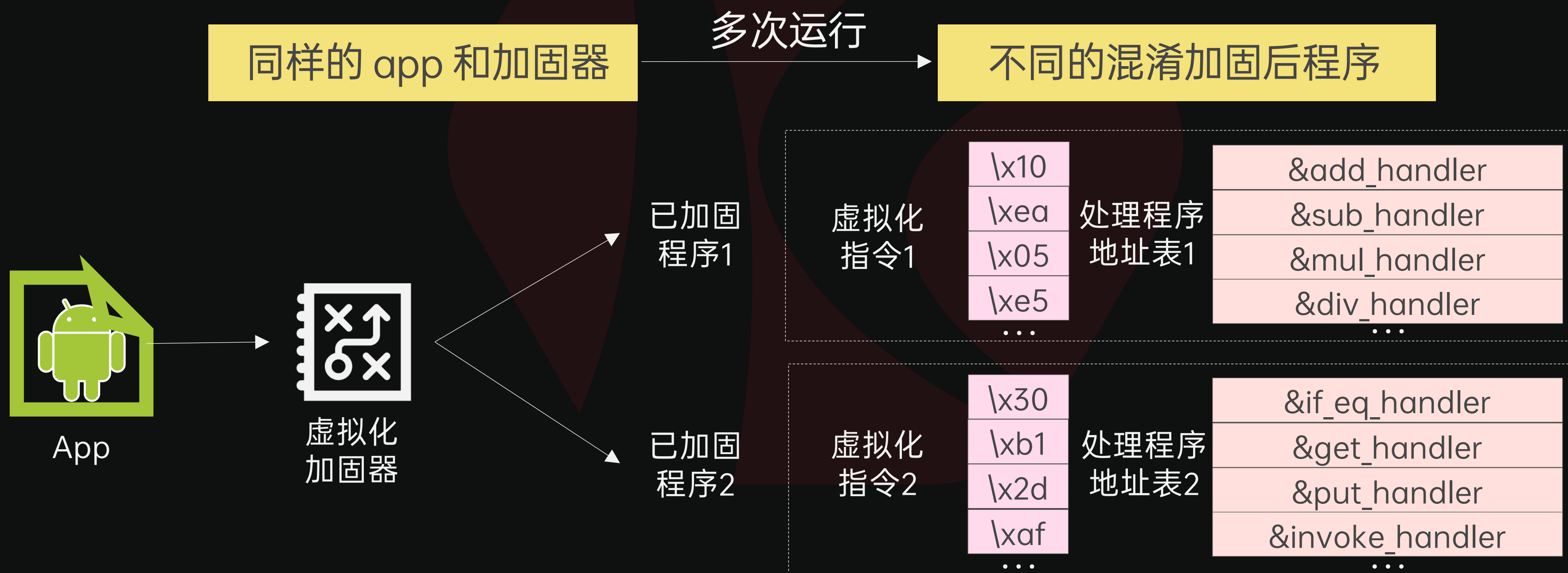
逆向的挑战和难点

- 对于在野恶意程序，仅能获取二进制文件，无法获取程序的源码/原始Dalvik字节码
- 此外，由于无法获取闭源加固器的源码，缺乏有关虚拟指令和虚拟机机制的信息
- 控制流非常复杂，因此静态和动态分析都很耗时



又一个挑战

- 许多受虚拟化加固的程序使用特定于应用程序的加密参数生成虚拟化指令，并且处理程序表中的地址顺序是随机的
- 因此，人工分析虚拟化加固的程序A的结果不能重复使用于经过相同加固器混淆的程序B



KCon

归源 · 智变
Returning to the source · Intelligent transformation

目录 / CONTENTS

01 虚拟化加固
简介及研究背景

02 Rhino逆向
RHINO字节码的特殊案例

03 加固后的程序结构
更常见的虚拟化加固程序模式

04 恢复原始字节码
基于已知模式进行逆向

05 观点和总结
安卓程序加固趋势

假设和前提条件

我们的逆向方法仅适用于以下假设和前提成立的情况:

- 被逆向的加固器是公开可访问的（无论是通过本地还是云端的形式），我们可以使用它来加固任何自定义的应用程序
- 加固器遵循“将 Dalvik 字节码转换为Native函数”的典型模式
- 在这种情况下，鉴于 Dalvik 字节码可以使用现有工具轻松转换为 Java，我们的目标是从混淆后的程序恢复出 Dalvik 字节码

加固后的程序

原始Dalvik
字节码

\x50
\xb1
\x08
\x8e
\x01
\x53
\x00
\xc0

...

加固

虚拟化
指令

\x10
\xea
\x05
\xe5
\x00
\x04
\xa9
\x7f

...

处理程序
地址表

&nop_handler
&mov_handler
&return_handler
&const_handler
&add_handler
&sub_handler
&mul_handler
&div_handler

...

处理程序对应
的指令序列

return handler

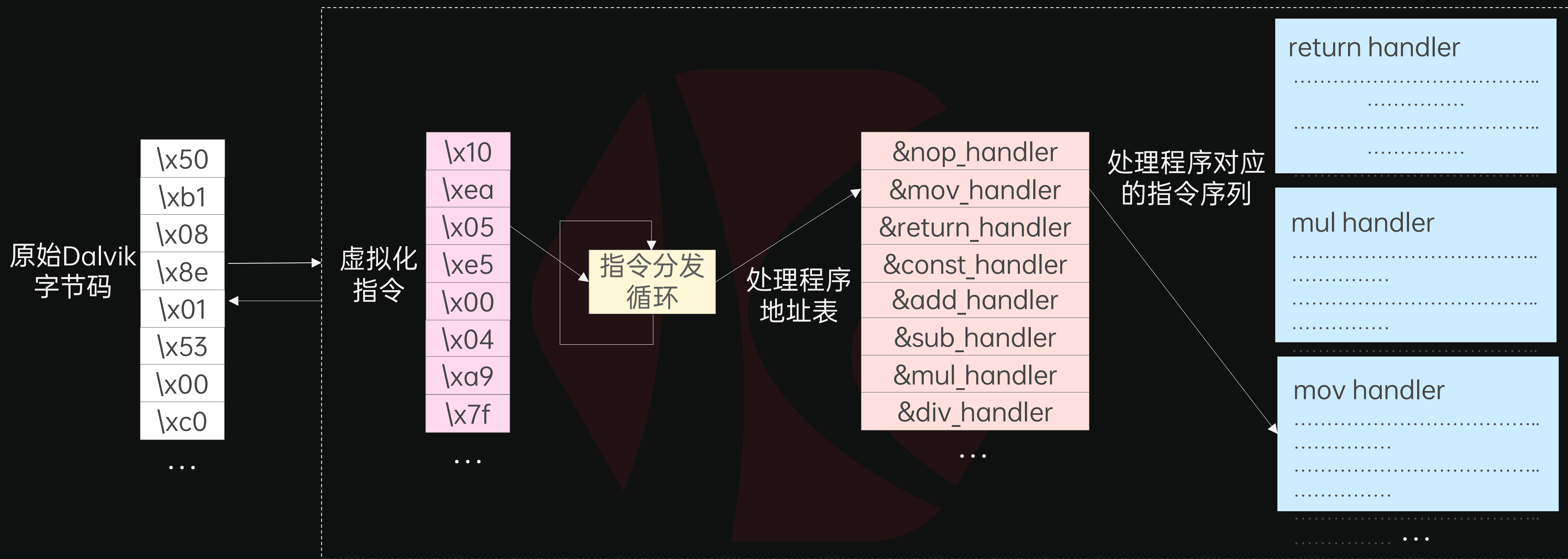
mul handler

mov handler

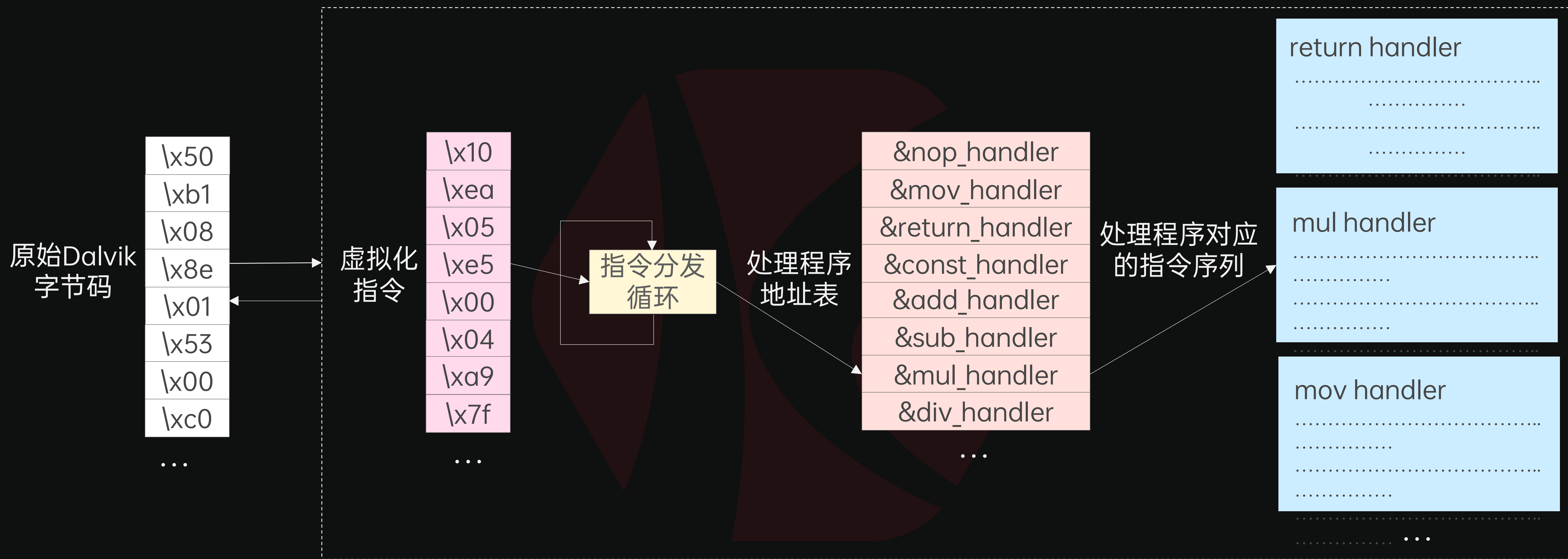
...

加固后的程序

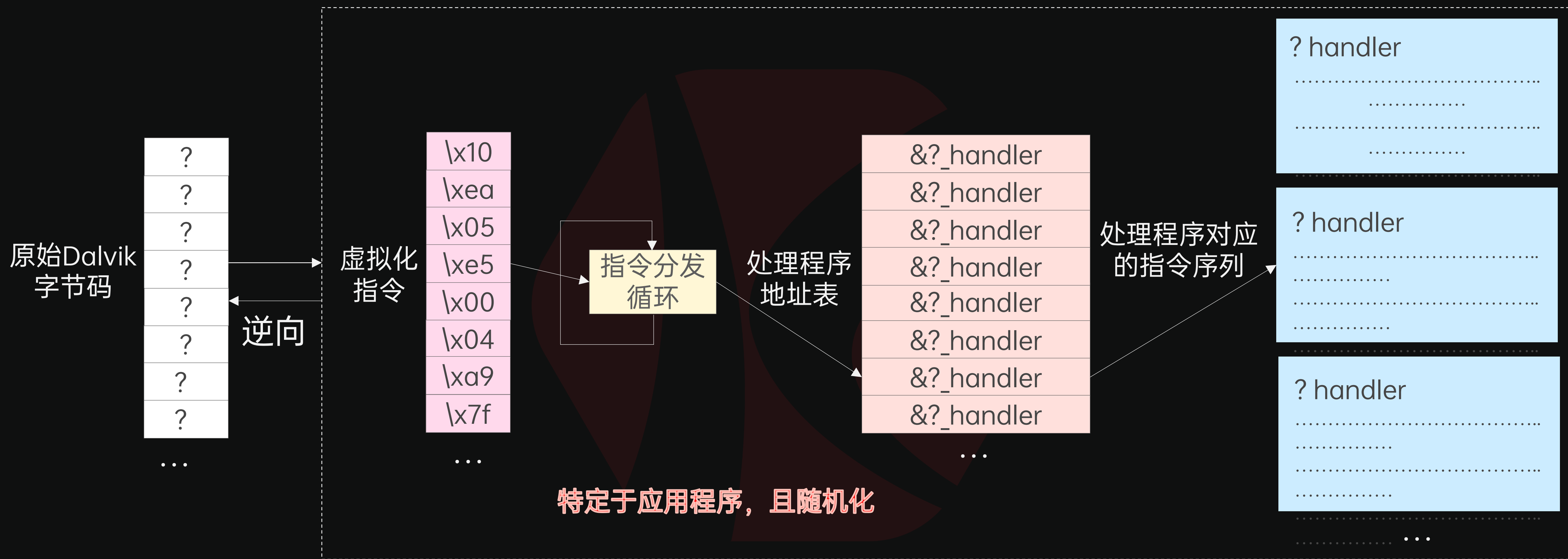
执行过程



执行过程续



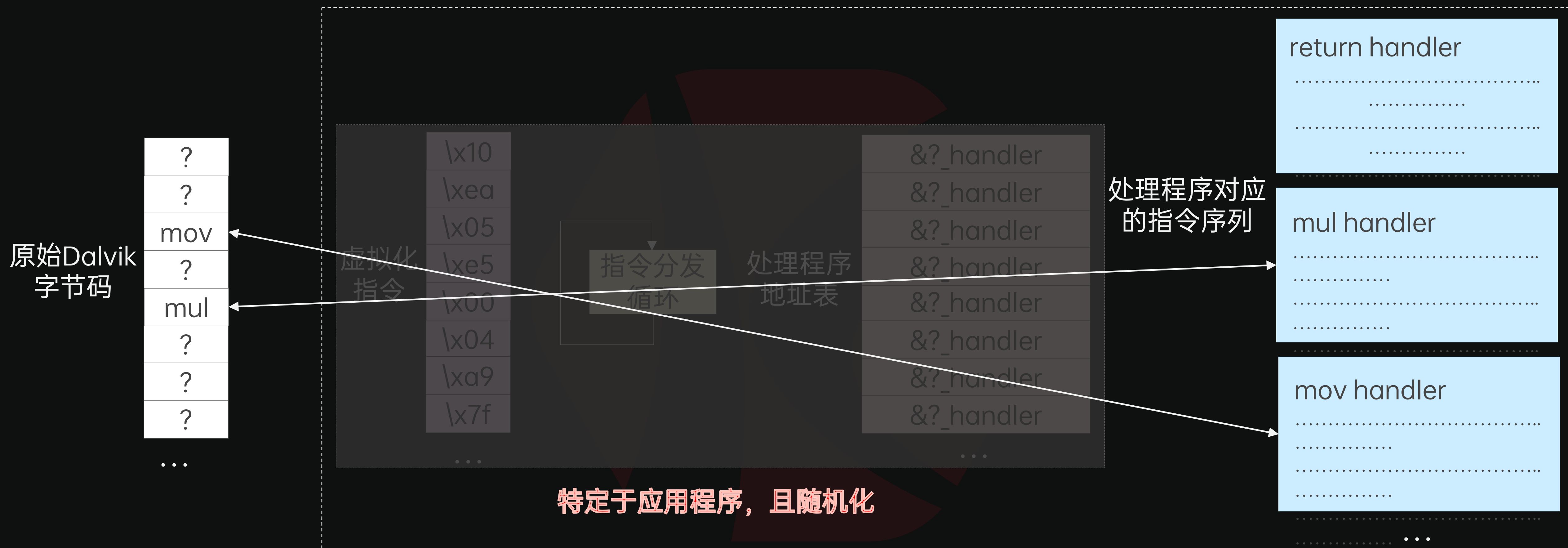
在野应用程序的逆向过程



加固过程中保持不变的部分

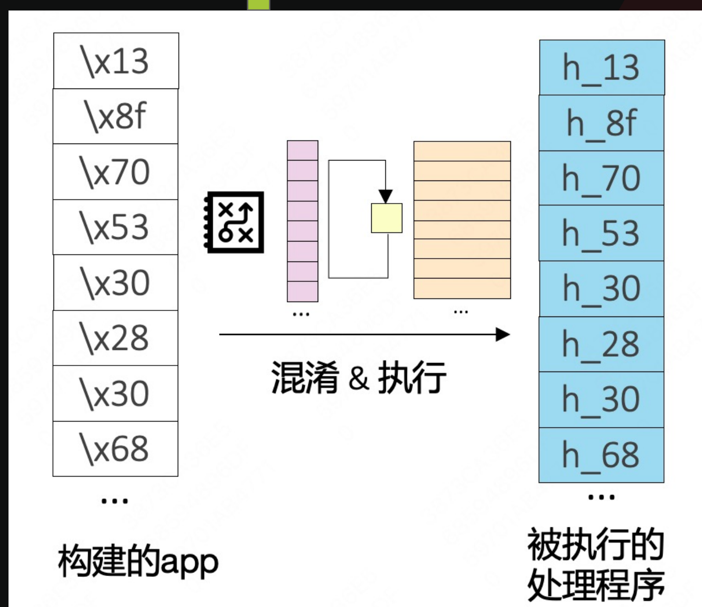
- 为了保持混淆后，程序的语义仍与原始程序相同，每个处理程序都是从一组简单的原始Dalvik字节码转换而来的
- 虽然中间步骤存在随机化与应用特定的加密参数，但原始Dalvik字节码和处理程序内容之间的关系是固定的（见下页图表）
- 此外，混淆后的函数通常使用与原始程序相同的方式传递参数

加固过程中保持不变的部分-续

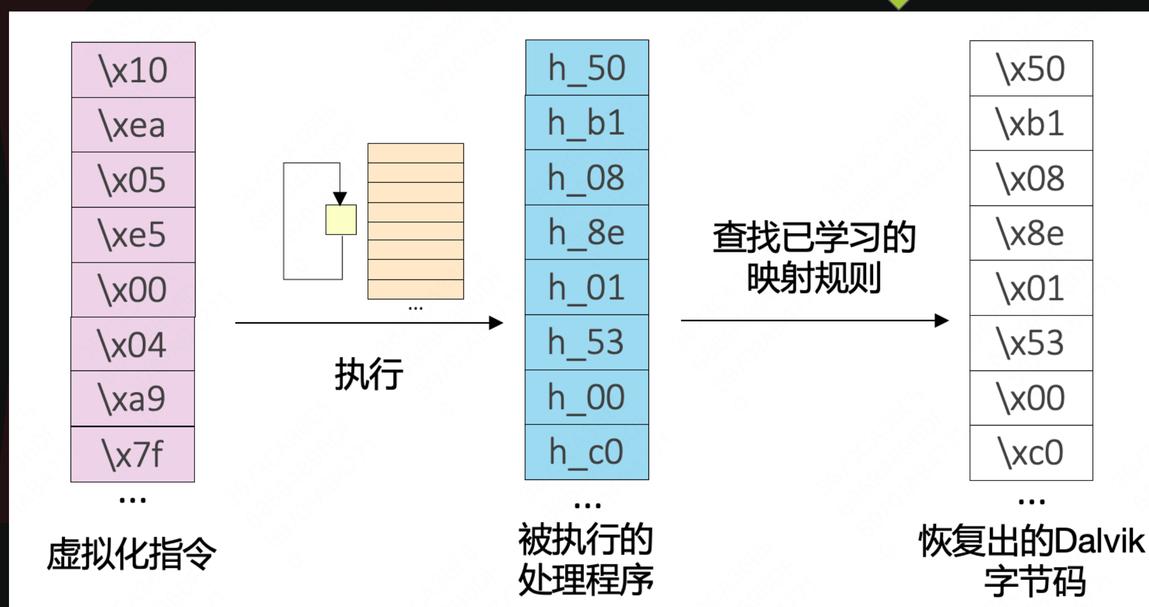


学习映射规则

- 根据上述内容，我们可以构建应用程序，对其进行混淆，并执行混淆程序，以学习原始Dalvik字节码和被执行的处理程序指令序列之间的映射关系。
- 然后再将学习到的规则应用于逆向其他在野捕获应用程序（分析执行处理程序的跟踪日志）。



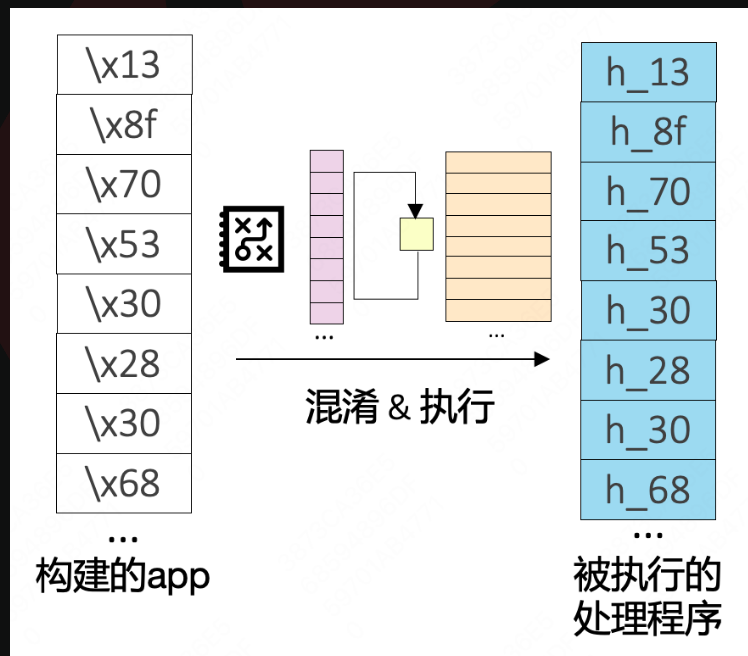
学习映射关系



将学习到的映射关系应用于逆向在野应用程序

学习映射规则阶段的问题

1. 确定每个函数的虚拟化指令。
2. 找出虚拟化指令和处理程序地址之间的关系
3. 通过内容识别处理程序，以便在执行实际应用程序时识别每个处理程序。



学习映射关系

建议的解法

1. 确定每个函数对应的虚拟化指令

- Hook 并跟踪用于注册Native方法的函数

2. 找出虚拟化指令和处理程序地址之间的映射关系

- 收集动态跟踪信息，并从其中定位分发器和处理程序地址，建立映射

3. 通过处理程序的内容识别它们，以便在处理在野应用程序的跟踪日志时识别

- 生成“软件基因”以识别每个处理程序。

1.虚拟化指令

```
.method protected onCreate(Landroid/os/Bundle;)V
    .registers 6

    .param p1, "savedInstanceState":Landroid/os/Bundle;

                          .line 20
    000014f8: 6f20 0100 5400          0000: invoke-super        {p0, p1}, Landroidx/activity/ComponentActivity;->onCreate(Landroid/os/Bundle;)V # method@0001
                          .line 21
    000014fe: 0740                    0003: move-object         v0, p0
    00001500: 1f00 0400               0004: check-cast          v0, Landroidx/activity/ComponentActivity; # type@0004
    00001504: 6201 0c00               0006: sget-object         v1, Lcom/example/myapplication/ComposableSingletons$MainActivityKt;->INSTANCE:Lcom/example/myapplication/ComposableSingletons$MainActivityKt; # field@000c
    00001508: 6e10 2b00 0100          0008: invoke-virtual      {v1}, Lcom/example/myapplication/ComposableSingletons$MainActivityKt;->getLambda-3$app_debug()Lkotlin/jvm/functions/Function2; # method@002b
    0000150e: 0c01                    000b: move-result-object  v1
```

```java
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ComponentActivityKt.setContent$default(this, null, ComposableSingletons$MainActivityKt.INSTANCE.m4078getLambda3$app_debug(), 1, null);
```

混淆前

```java
public native void onCreate(Bundle bundle);
```

混淆后

```c
__int64 __fastcall sub_C8C4(__int64 a1, __int64 a2)
{
  void *v3; // [xsp+0h] [xbp-70h] BYREF
  int v4; // [xsp+8h] [xbp-68h]
  __int128 *v5; // [xsp+10h] [xbp-60h]
  int *v6; // [xsp+18h] [xbp-58h]
  void *v7; // [xsp+20h] [xbp-50h]
  int v8; // [xsp+28h] [xbp-48h] BYREF
  __int16 v9; // [xsp+2Ch] [xbp-44h]
  __int128 v10[2]; // [xsp+30h] [xbp-40h] BYREF
  __int64 v11; // [xsp+50h] [xbp-20h]
  __int64 v12; // [xsp+58h] [xbp-18h]
  __int64 v13; // [xsp+68h] [xbp-8h]

  v13 = *(_QWORD *)(_ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2)) + 40);
  v4 = 61;
  v11 = 0LL;
  v12 = a2;
  v9 = 256;
  v3 = &unk_6CC8;
  memset(v10, 0, sizeof(v10));
  v8 = 0;
  v5 = v10;
  v6 = &v8;
  v7 = &unk_6D42;
  return vmInterpret(a1, &v3, &off_10720);
}
```

```
data:0000000000006CC8 E6 00 46 00 00 00 05 00 01 00+word_6CC8 DCW
data:0000000000006CC8 38 00 6F 00 6B 00 D9 01 2D 00+
data:0000000000006CC8 20 02 67 27 D3 30 1B 01 10 02+DCW 0x11E, 0,
data:0000000000006CC8 9F 10 1E 01 00 00 05 01 A1 02+DCW 0xE1, 1,
data:0000000000006CC8 01 01 06 00 E6 10 47 00 02 00+DCW 0x54D3, 0x
data:0000000000006D42 02                         unk_6D42 DCB
```

虚拟化指令

1.确定函数对应的虚拟化指令

- Hook 并跟踪传递给 env->RegisterNatives( ) 的参数，以获取函数的签名和虚拟指令的地址

```
static int registerNativeMethods(JNIEnv* env)
{
    jclass clazz;
    clazz = env->FindClass("np/manager/Protect");
    if (clazz == NULL) {
        return JNI_FALSE;
    }
    static JNINativeMethod gMethods[] = {
    {"classesInit0", "(I)V", (void*)Jni_classesInit0},
    ...
    };
    if (env->RegisterNatives(clazz, gMethods, sizeof(gMethods) / sizeof(gMethods[0])) < 0) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

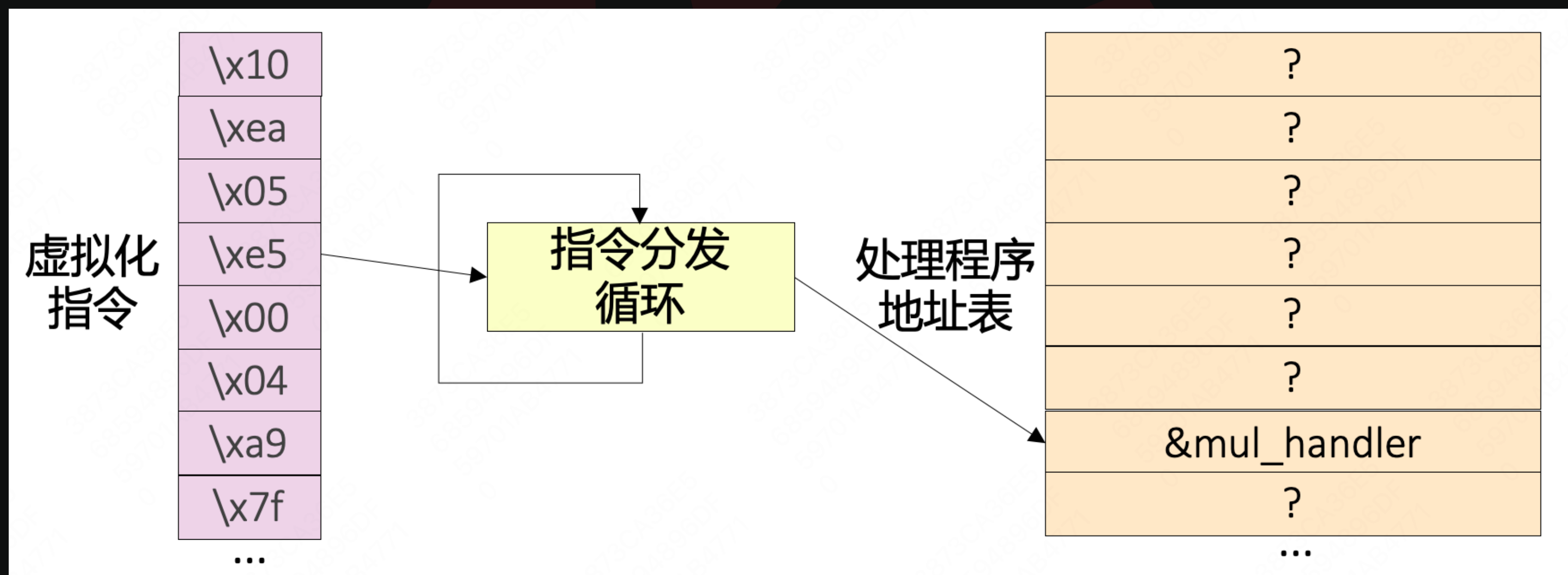
Hook

函数名	签名	虚拟化指令
onCreate	(Landroid/os/Bundle;)V	\x12\x20\x00\x00T\x00D@\xd0\x00…
invoke	(Landroidx/compose/runtime/Composer;I)V	\x01\x00\x04\x00\xa7\x20\r\x00\x05\x00]\x00\x06…
…	…	…

部分虚拟化指令与函数对应关系

2.虚拟化指令与处理程序地址的映射

- 跳转在函数内部进行，需要观察函数内部的执行流程，所以仅仅Hook函数的入口和出口是不够的
- 一种可能的方法是动态调试，但它需要大量繁琐的手动工作，并且需要应对反调试机制
- 因此，我们使用插桩+分析跟踪(Trace)日志的方式来建立映射关系



指令级插桩和跟踪的工具和框架

- DBI : Valgrind, DynamoRIO, QBDI, Intel Pin…
- 模拟器: Unicorn, Unidbg…
- Frida-Stalker

- 考虑因素
- 是否支持安卓: 排除 Intel Pin 和 Valgrind
- 补充环境: 使用模拟器时需要

- QBDI或其他能够进行指令级插桩、跟踪的框架皆可

定位分发器和处理程序表

- 分发器首先将处理程序的地址加载到一个寄存器中，然后使用“br”跳转到该地址。
- 该地址指向代码段，但本身存储在数据段中，更确切地说，是在处理程序地址表中。
- 跳转到处理程序之后，将执行该处理程序的内容，即指令序列。
- 只需对代码块开头进行插桩

分发器的部分指令

```
0x7627821b30    sub     sp, sp, #112
0x7627821b34    stp     x29, x30, [sp, #96]
0x7627821b38    add     x29, sp, #96
0x7627821b3c    mrs     x8, TPIDR_EL0
0x7627821b40    ldr     x8, [x8, #40]
0x7627821b44    stur    x8, [x29, #-8]
0x7627821b48    stur    x0, [x29, #-40]
0x7627821b4c    str     x1, [sp, #48]
0x7627821b50    adrp    x1, #-40960
0x7627821b54    add     x1, x1, #1838
0x7627821b58    sub     x0, x29, #32
0x7627821b5c    str     x0, [sp, #8]
0x7627821b60    bl      #149488
0x7627846350    adrp    x16, #16384
0x7627846354    ldr     x17, [x16, #3728]
0x7627846358    add     x16, x16, #3728
0x762784635c    br      x17
```

处理程序表中的一个地址

处理程序的指令序列

```
0x7627821bdc    sub     sp, sp, #80
0x7627821be0    stp     x29, x30, [sp, #64]
0x7627821be4    add     x29, sp, #64
0x7627821be8    mrs     x8, TPIDR_EL0
0x7627821bec    str     x8, [sp, #16]
0x7627821bf0    ldr     x8, [x8, #40]
0x7627821bf4    stur    x8, [x29, #-8]
0x7627821bf8    str     x0, [sp, #32]
0x7627821bfc    str     x1, [sp, #24]
0x7627821c00    ldr     x0, [sp, #32]
0x7627821c04    str     x0, [sp, #8]
0x7627821c08    sub     x1, x29, #16
0x7627821c0c    sub     x2, x29, #24
```

加密的字节码和处理程序地址的映射

- 上述函数每次运行的跟踪日志，都会建立一条虚拟指令、和它对应的处理程序之间的映射关系

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
00000000  94 00 10 00 00 00 78 00 92 00 05 00 bb 00 12 00  ......x.........
00000010  ba 00 9f 00 13 00 92 00 10 00 bb 00 12 00 94 10  ................
00000020  2a 00 00 00 be 00 9a 01 07 00 94 20 11 00 01 00  *.......... ....
00000030  be 00 18 00 13 00 71 10 0e 00 00 00 be 00 98 00  ......q.........
00000040  2d 00 32 10 2b 00 00 00 78 00 ba 00              -.2.+...x...
```

Dump得到的虚拟化指令

```
.data.rel.ro:000000000003C018 30 76 01 00 00 00 00 00    off_3C018 DCQ sub_17630
.data.rel.ro:000000000003C020 1C D0 01 00 00 00 00 00    DCQ loc_1D01C
.data.rel.ro:000000000003C028 C0 96 01 00 00 00 00 00    DCQ loc_196C0
.data.rel.ro:000000000003C030 60 C7 01 00 00 00 00 00    DCQ loc_1C760
.data.rel.ro:000000000003C038 AC 86 01 00 00 00 00 00    DCQ loc_186AC
.data.rel.ro:000000000003C040 64 B0 01 00 00 00 00 00    DCQ loc_1B064
.data.rel.ro:000000000003C048 68 76 01 00 00 00 00 00    DCQ sub_17668
.data.rel.ro:000000000003C050 80 7E 01 00 00 00 00 00    DCQ loc_17E80
.data.rel.ro:000000000003C058 34 84 01 00 00 00 00 00    DCQ loc_18434
.data.rel.ro:000000000003C060 D8 AC 01 00 00 00 00 00    DCQ loc_1ACD8
```

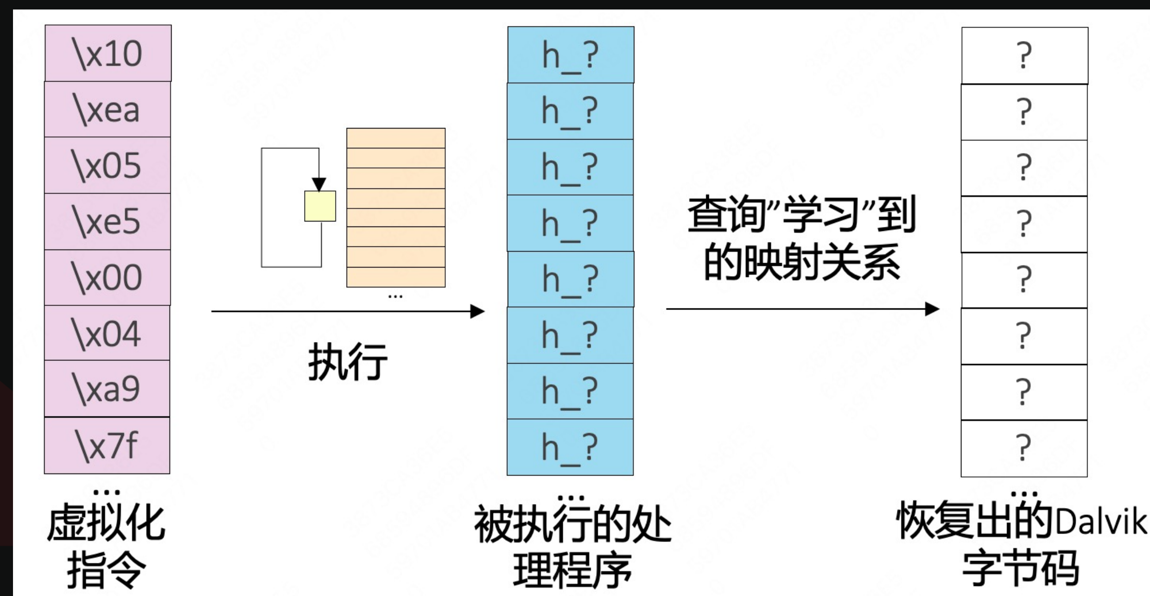
处理程序地址表

3.识别每个处理程序

- 对于在野的被加固应用，由于拿不到原始dex文件，并且处理程序表中的地址顺序又是随机的，仅凭处理程序的地址无法识别一个处理程序，即无法知道它是哪条虚拟指令对应的处理程序
- 因此，在学习映射的阶段，我们就需要基于处理程序的内容，去标识每一个处理程序

```
0x7627821b54    add     x1, x1, #1838
0x7627821b58    sub     x0, x29, #32
0x7627821b5c    str     x0, [sp, #8]
0x7627821b60    bl      #149488
0x7627846350    adrp    x16, #16384
0x7627846354    ldr     x17, [x16, #3728]
0x7627846358    add     x16, x16, #3728
0x762784635c    br      x17
0x7627821bdc    sub     sp, sp, #80
0x7627821be0    stp     x29, x30, [sp, #64]
0x7627821be4    add     x29, sp, #64
0x7627821be8    mrs     x8, TPIDR_EL0
0x7627821bec    str     x8, [sp, #16]
0x7627821bf0    ldr     x8, [x8, #40]
0x7627821bf4    stur    x8, [x29, #-8]
0x7627821bf8    str     x0, [sp, #32]
0x7627821bfc    str     x1, [sp, #24]
0x7627821c00    ldr     x0, [sp, #32]
0x7627821c04    str     x0, [sp, #8]
0x7627821c08    sub     x1, x29, #16
0x7627821c0c    sub     x2, x29, #24
```

如何判断这些指令对应h_50处理程序?



使用已学习的映射关系来逆向恢复应用程序

处理程序的"软件基因"

我们使用“hash(hex(指令序列))”作为处理程序的软件基因，但指令序列需要先经过以下处理步骤：

- 将序列截断，仅包括第一条“br”指令之前的部分。
- 将在不同程序中具有不同对应机器码的指令替换为固定序列。这种指令包括跳转指令（例如b、bl）和PC寄存器相对指令（例如adrp）等。

```
<CsInsn 0x4 [fc03192a]: mov w28, w25>,
<CsInsn 0x4 [880e0012]: and w8, w20, #0xf>,
b,
<CsInsn 0x4 [fb0314aa]: mov x27, x20>,
<CsInsn 0x4 [940a4079]: ldrh w20, [x20, #4]>,
<CsInsn 0x4 [e91f40f9]: ldr x9, [sp, #0x38]>,
<CsInsn 0x4 [385968f8]: ldr x24, [x9, w8, uxtw #3]>,
cbz,
<CsInsn 0x4 [e81740f9]: ldr x8, [sp, #0x28]>,
<CsInsn 0x4 [61074079]: ldrh w1, [x27, #2]>,
......
```

处理程序的指令序列

```
fc03192a
880e0012
0a0a0a0a
fb0314aa
940a4079
e91f40f9
385968f8
0c0c0c0c
e81740f9
61074079
......
```

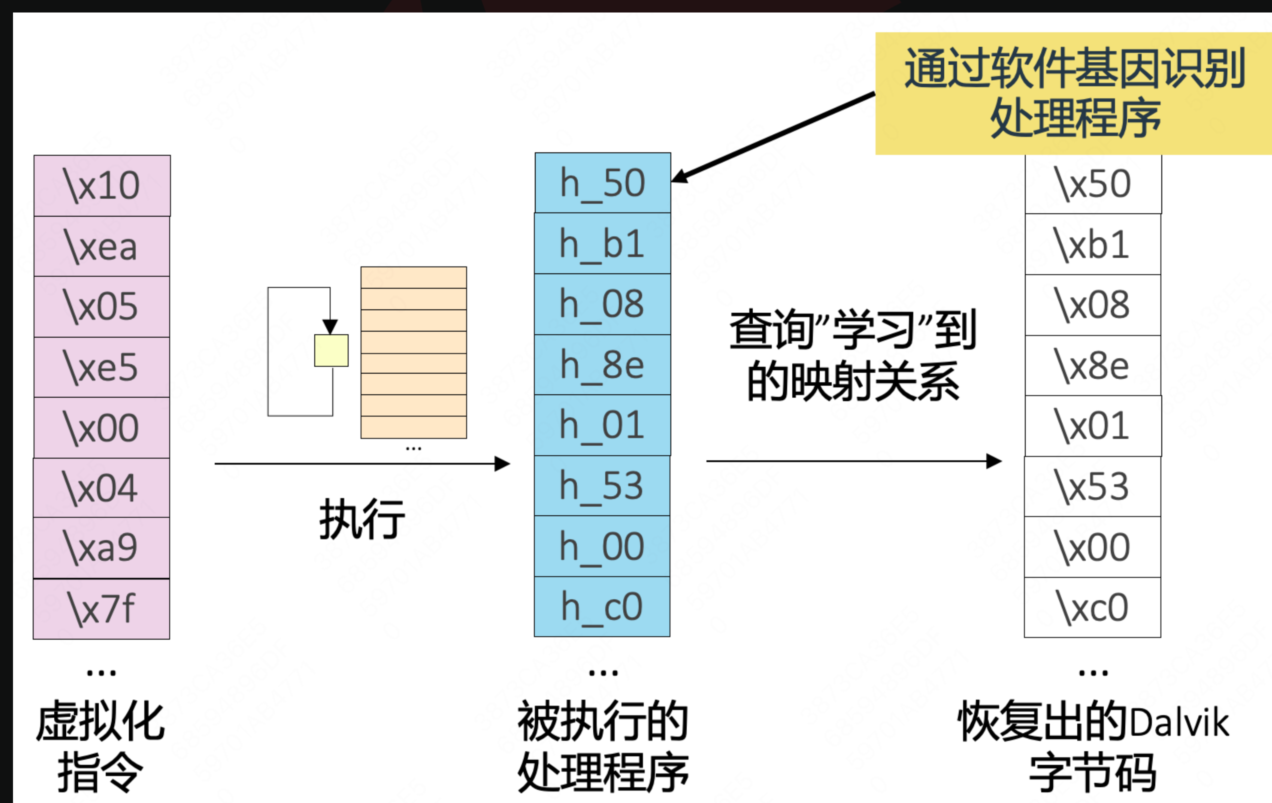
hexed

```
0d627c5bece5fb6ba3273accbaa028ab
```

软件基因

整合汇总: 逆向在野应用

- 执行野外应用程序以获取与执行的虚拟化指令相对应的基因签名，然后使用已学习到的映射关系将其恢复为Dalvik字节码



KCon

归源·智变

Returning to the source · Intelligent transformation

目录 / CONTENTS

01 虚拟化加固
简介及研究背景

02 Rhino逆向
RHINO字节码的特殊案例

03 加固后的程序结构
更常见的虚拟化加固程序模式

04 恢复原始字节码
基于已知模式进行逆向

05 观点和总结
安卓程序加固趋势

廉价打包器在Android恶意软件混淆中的应用

- 我们注意到恶意软件使用廉价的基于虚拟机的打包器的趋势。
- 最著名的商业打包器之一每年收费18000元，而一些廉价的打包器每年仅收费约30元。
- 这对于价格敏感的恶意软件作者很有吸引力——大部分近期发现的虚拟化加固的恶意软件样本，都是使用廉价加固器混淆的。

百度应用加固
使用帮助 联系我们 (普通用户)
产品购买
产品购买
加固管理
帐户信息
产品和服务购买
1 产品选择 2 信息确认 3 订单支付
产品选择 安卓应用加固（专业版） 安卓应用加固（旗舰版） iOS应用加固 安卓SDK加固 iOS SDK加固 H5加固 白盒加密
具备基础SO、DEX代码加固能力，支持基础反调试、防重打包等功能，可以有效抵御常规的逆向、调试、重打包等攻击行为 了解详情>>
服务咨询
购买时长 1年
APP个数 1 个
购买须知
一、产品说明
1、已购买产品的使用时长对已经加固了的包不会产生影响，到期后无法再使用相应产品进行加固。
二、特别说明
1、如果客户使用正常应用进行加固，加固后的包出现使用异常，请联系客服协助解决，直到加固后的包可正常使用。
2、如果客户购买加固产品是为过检，请购买前进行咨询，如购买前未咨询导致使用加固产品后无法过检，客户不可申请退款。
支付金额:¥ 18000.00
购买

01 选择商品
商品说明 若提示未支付，可以通过微信/支付宝账单商家订单号查询卡密！
商品分类: NP管理器
商品名称: 捐赠免一年广告
商品单价: 29.00
购买数量: 1 库存很多
联系方式: 必须为手机号码
特色服务: 短信提醒【收费0.2元】 邮箱提醒【免费】
应付金额: 29.00

安卓上的另一种常见混淆技术

- 隐藏原始Dalvik字节码数据，并在执行期间将dex文件动态释放到内存中是另一种广泛使用的混淆技术，它使得静态分析无法发挥作用。
- 对于这种混淆技术，现有的反混淆工具会从内存中搜索并提取Dex数据。
- 然而，这种搜索内存型的反混淆工具无法逆向受虚拟化加固的应用程序，因为在虚拟化加固的程序执行时，原始的dex从未出现在内存中。

要点总结

综上，我们提出了针对虚拟化加固的安卓应用程序进行逆向的两种不同情况的方法：

- 对于使用Rhino字节码的特定类型的混淆，由于解释器是开源的，我们分析虚拟机，重构AST，并从字节码中恢复源代码
- 对于更常见和普遍的情况，即加固器并非开源，我们通过构建应用程序、收集执行跟踪日志，并使用软件基因，来学习原始字节码和处理程序执行序列之间的映射关系，然后将学习到的关系应用于恢复在野的虚拟化加固应用程序的语义。

参考链接

- https://web.archive.org/web/20230531155247/https://blog.autojs.org/2022/08/24/encryption/
- https://swarm.ptsecurity.com/how-we-bypassed-bytenode-and-decompiled-node-js-bytecode-in-ghidra/
- https://github.com/QBDI/QBDI
- https://www4.comp.polyu.edu.hk/~csxluo/Parema.pdf

KCon

归源 · 智变

Returning to the source · Intelligent transformation

感谢您的观看！

THANK YOU FOR YOUR WATCHING