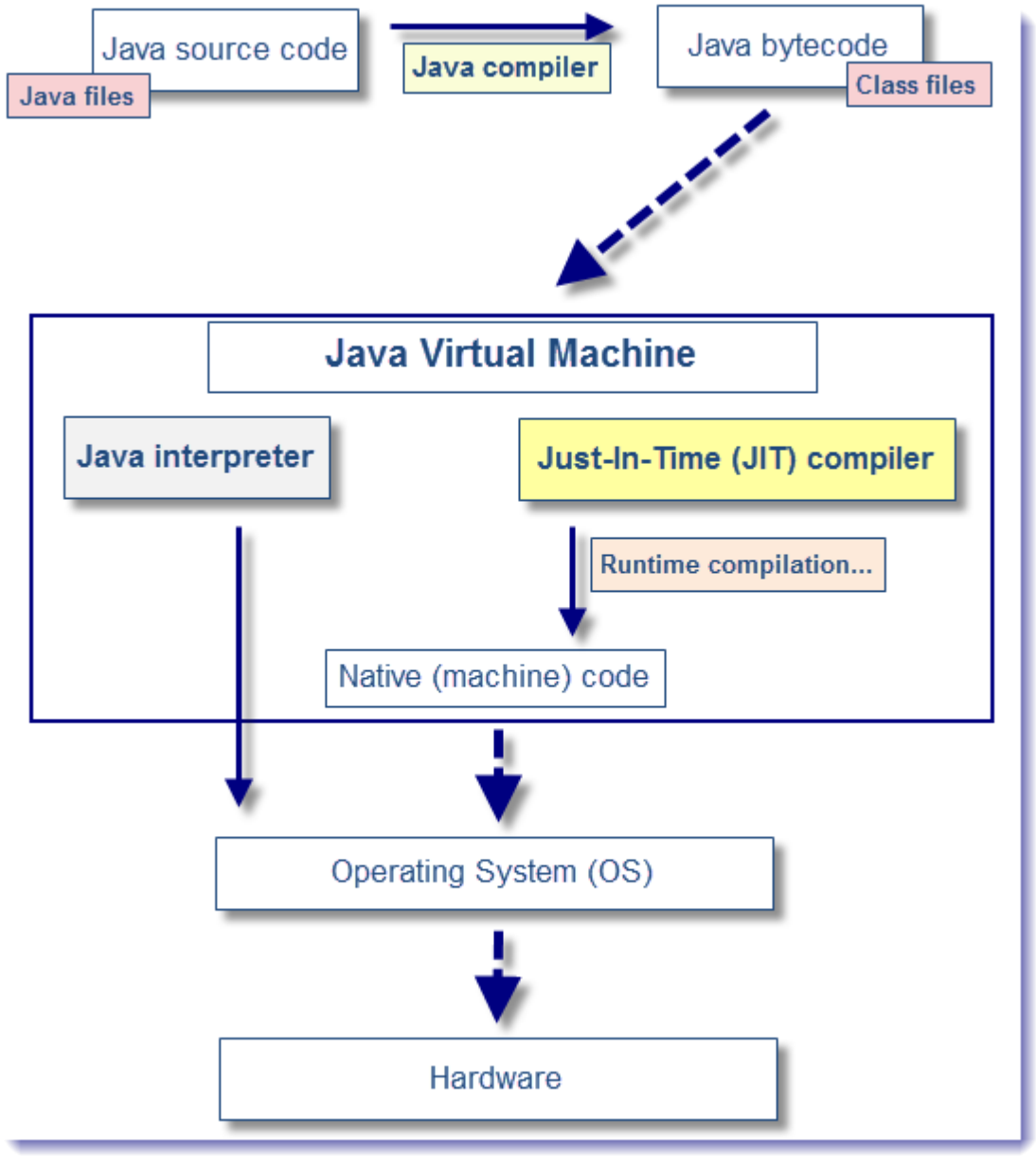
北京大学
PEKING UNIVERSITY

安卓恶意软件混淆技术

2025/3/4

目录

- 静态混淆
  - 清单文件
  - 符号
  - 控制流
  - 间接调用
  - 加密
- 动态混淆
  - 动态加载/反射调用
  - native混淆

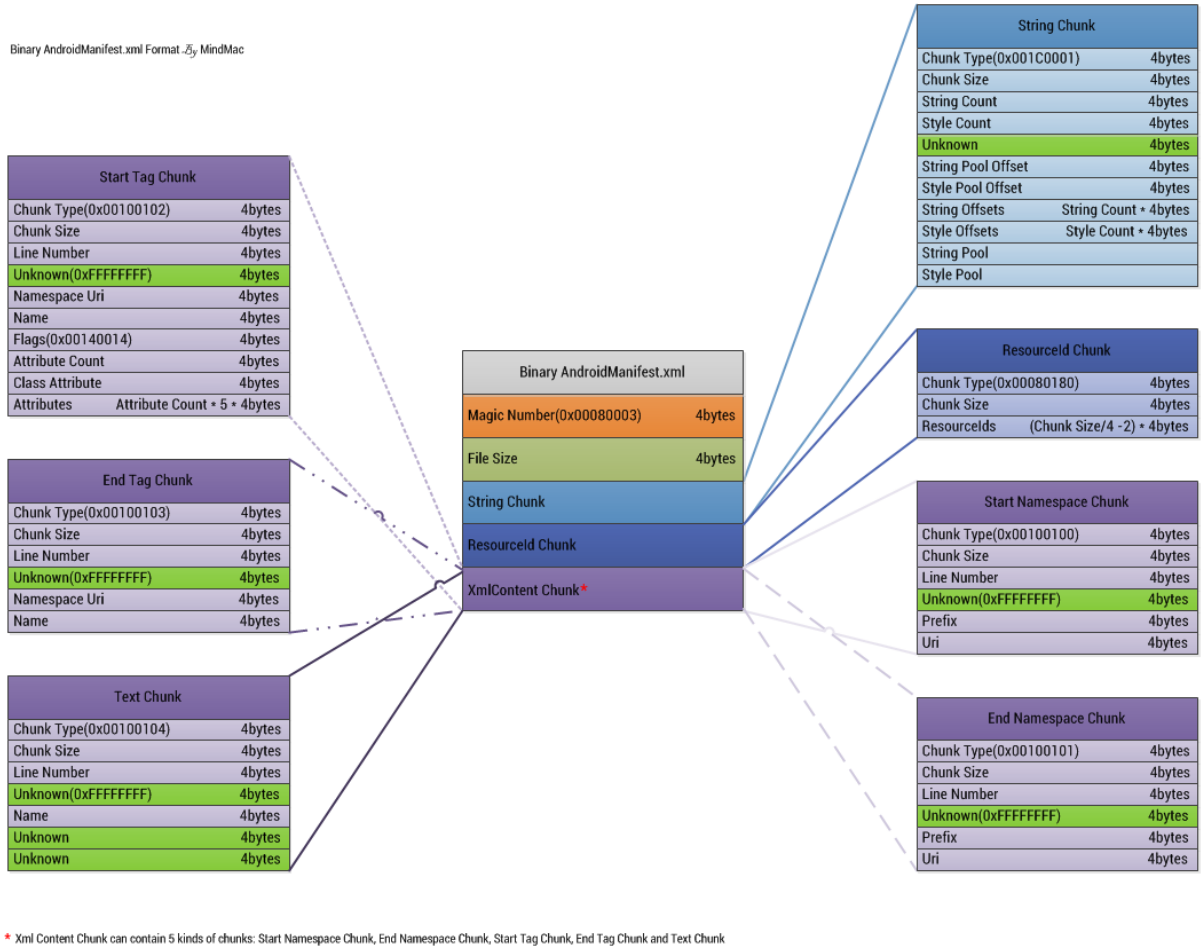


静态混淆-混淆工具对抗

- 恶意软件可能利用Android在解析APK文件时未严格校验ZIP格式的某些字段，从而通过篡改APK文件的ZIP格式字段来绕过基于ZIP格式解析的反编译工具

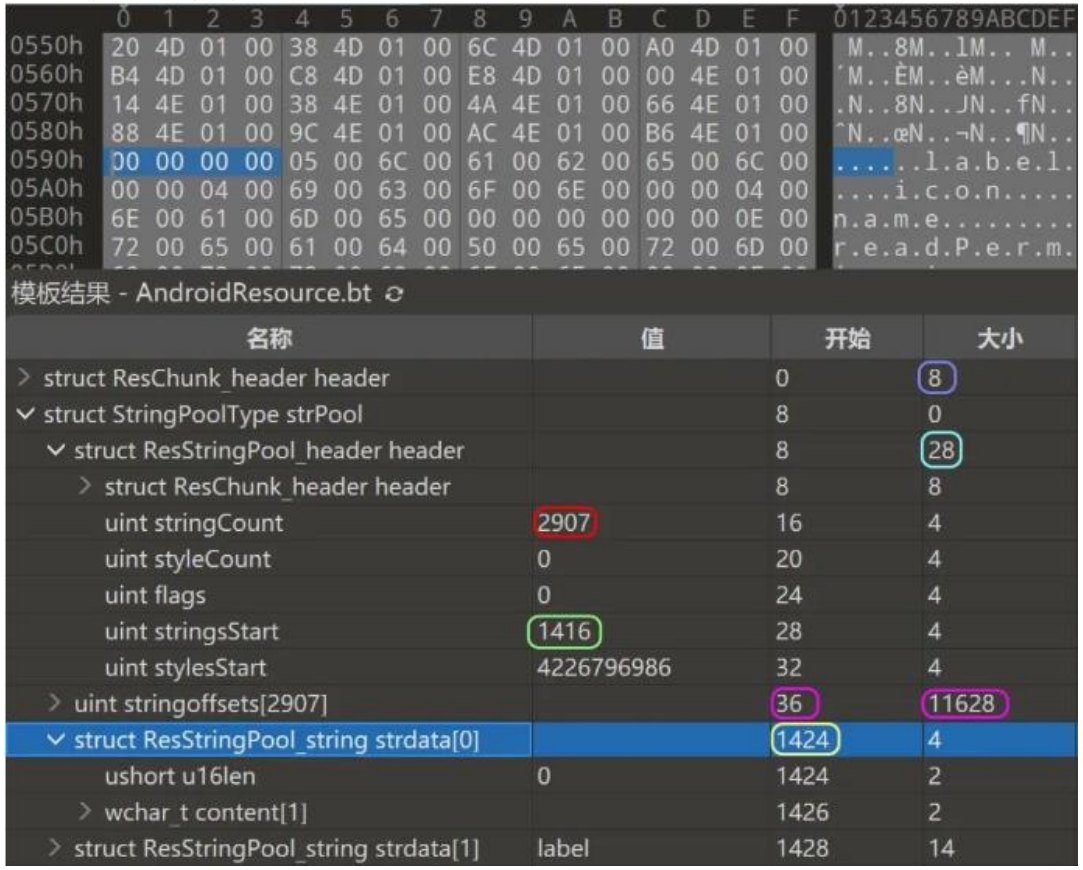
字段名	字段含义
filename	文件名称
compress_type	压缩类型
flag_bits	zip标志位，加密，注释
CRC32	CRC校验值
compress_size	压缩后文件大小
file_size	未压缩的文件大小
extract_version	解压zip的最小版本
header_offset	文件头的偏移
external_attr	文件的属性
create_system	创建文件的系统
comment	zip文件注释

- 清单文件是必须解析的关键部分，它包含了组件、权限等信息。为了规避反编译工具的识别，恶意软件通常通过精心篡改清单文件来干扰其正常解析。



静态混淆-混淆工具对抗

stringCount的值为2907，而StringOffsets开始于偏移位置36，大小为 11628。StringOffsets是一个包含每个字符串在字符串池中的相对偏移量的数组，其大小为 stringCount * 4，即 11628。反编译工具按照混淆后的stringCount值计算StringOffsets大小，并进行解析，因此解析结果出错。这种不一致性是导致反编译工具解析失败的原因。然而，Android系统在处理清单文件时并未直接依赖stringCount字段的值。通过查看 Android 源码可以发现，Android系统在运行时根据实际数据动态计算 stringPoolSize，而非直接使用文件中提供的数值。



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0550h	20	4D	01	00	38	4D	01	00	6C	4D	01	00	A0	4D	01	00	M..8M..lM.. M..
0560h	B4	4D	01	00	C8	4D	01	00	E8	4D	01	00	00	4E	01	00	´M..ÈM..èM...N..
0570h	14	4E	01	00	38	4E	01	00	4A	4E	01	00	66	4E	01	00	.N..8N..JN..fN..
0580h	88	4E	01	00	9C	4E	01	00	AC	4E	01	00	B6	4E	01	00	ˆN..œN..¬N..¶N..
0590h	00	00	00	00	05	00	6C	00	61	00	62	00	65	00	6C	00	....l.a.b.e.l.
05A0h	00	00	04	00	69	00	63	00	6F	00	6E	00	00	00	04	00	....i.c.o.n.....
05B0h	6E	00	61	00	6D	00	65	00	00	00	00	00	00	00	0E	00	n.a.m.e.........
05C0h	72	00	65	00	61	00	64	00	50	00	65	00	72	00	6D	00	r.e.a.d.P.e.r.m.

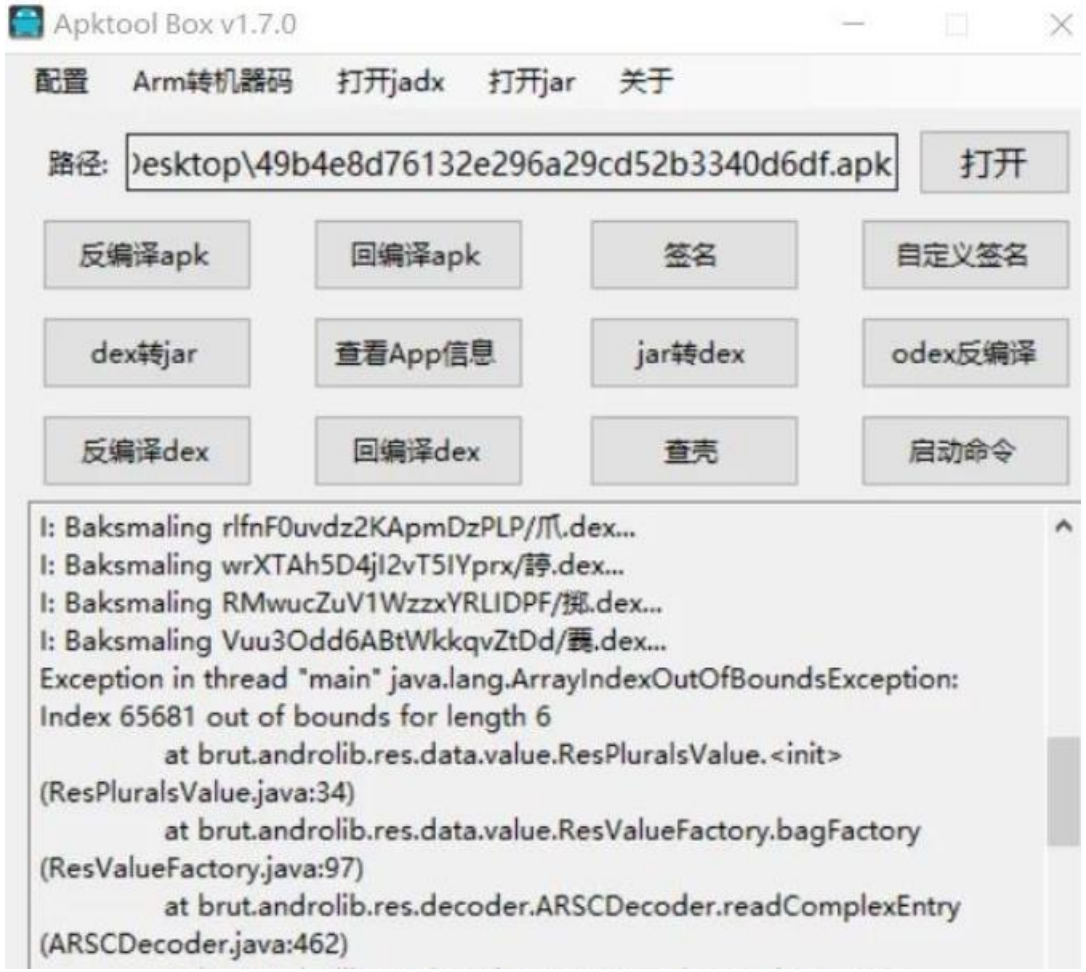
模板结果 - AndroidResource.bt

名称	值	开始	大小
struct ResChunk_header header		0	8
struct StringPoolType strPool		8	0
struct ResStringPool_header header		8	28
struct ResChunk_header header		8	8
uint stringCount	2907	16	4
uint styleCount	0	20	4
uint flags	0	24	4
uint stringsStart	1416	28	4
uint stylesStart	4226796986	32	4
uint stringoffsets[2907]		36	11628
struct ResStringPool_string strdata[0]		1424	4
ushort u16len	0	1424	2
wchar_t content[1]		1426	2
struct ResStringPool_string strdata[1]	label	1428	14

```
if (mHeader->styleCount == 0) {
    mStringPoolSize = (mSize - mHeader->stringsStart) / charSize;
} else {
    // check invariant: styles starts before end of data
    if (mHeader->stylesStart >= (mSize - sizeof(uint16_t))) {
        ALOGW("Bad style block: style block starts at %d past data size of %d\n",
            (int)mHeader->stylesStart, (int)mHeader->header.size);
        return (mError=BAD_TYPE);
    }
    // check invariant: styles follow the strings
    if (mHeader->stylesStart <= mHeader->stringsStart) {
        ALOGW("Bad style block: style block starts at %d, before strings at %d\n",
            (int)mHeader->stylesStart, (int)mHeader->stringsStart);
        return (mError=BAD_TYPE);
    }
    mStringPoolSize =
        (mHeader->stylesStart-mHeader->stringsStart)/charSize;
```

静态混淆-混淆工具对抗

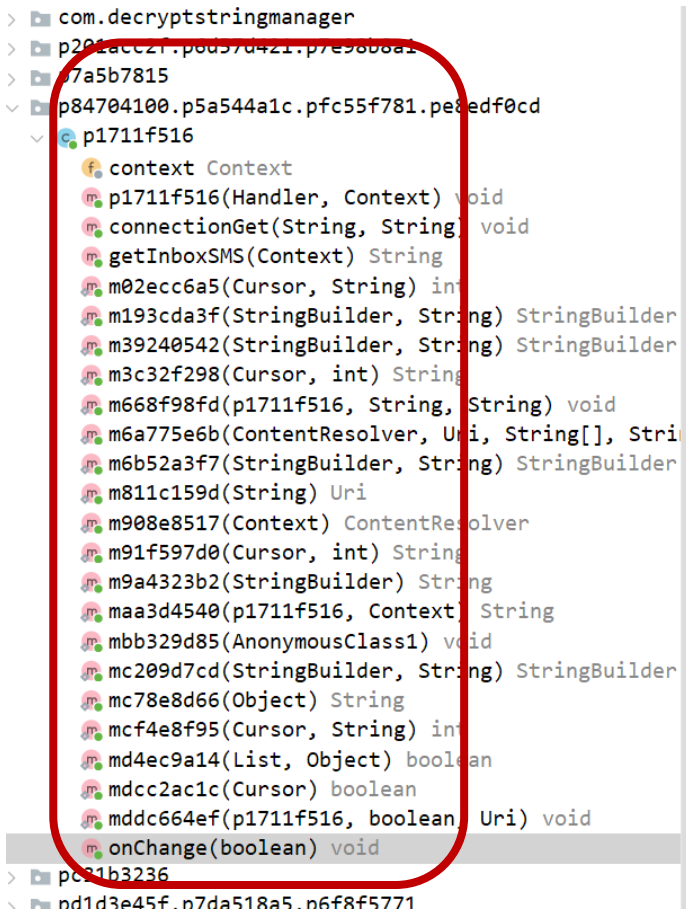
1. 修改清单文件的Magic Number，阻止反编译工具
2. 插入脏数据，在多个对象之间插入异常字符，导致解析失败
3. 插入超长字符串或者特殊字符
4. 混淆resources.arsc文件， BOOMSLANG（树蛸）移动欺诈家族就利用这一策略对抗Apktool的反编译过程，右图是其反编译失败时Apktool提示的错误信息



静态混淆-代码混淆

标识符混淆：旨在通过将有意义的包名、类名、方法名和变量名替换为无意义的、随机生成的名称，从而干扰逆向工程师的分析过程。

右图中可以看到，apk中方法，类名，包名都被混淆为无意义无序字符串，增加了工程师分析恶意代码的难度




```
123             }
124
125             public static void mf1f835ad(HttpURLConnection httpURLConnec
126                 httpURLConnection.connect();
127             }
128
129             @Override // java.lang.Thread, java.lang.Runnable
130             public void run() {
131                 if ((22 + 25) % 25 > 0) {
132                 }
133                 try {
134                     mf1f835ad((HttpURLConnection) m236435c6(new URL(m4ac
135                 } catch (IOException e) {
136                 }
137             }
138         });
139     }
140
141     public String getInboxSMS(Context context) {
142         if ((27 + 17) % 17 > 0) {
143         }
144         StringBuilder sb = new StringBuilder();
145         Cursor m6a775e6b = m6a775e6b(m908e8517(context), m811c159d(Decry
146         while (m6a775e6b != null && mdcc2ac1c(m6a775e6b)) {
147             mc209d7cd(m6b52a3f7(m193cda3f(m39240542(sb, m3c32f298(m6a775
148         }
149         return m9a4323b2(sb);
150     }
151
152     @Override // android.database.ContentObserver
153     public void onChange(boolean z) {
154         if ((12 + 7) % 7 > 0) {
155         }
156         mddc664ef(this, z, null);
157         ArrayList arrayList = new ArrayList();
158         md4ec9a14(arrayList, maa3d4540(this, this.context));
159         m668f98fd(this, DecryptString.decryptString("f173637973f85764b94
```

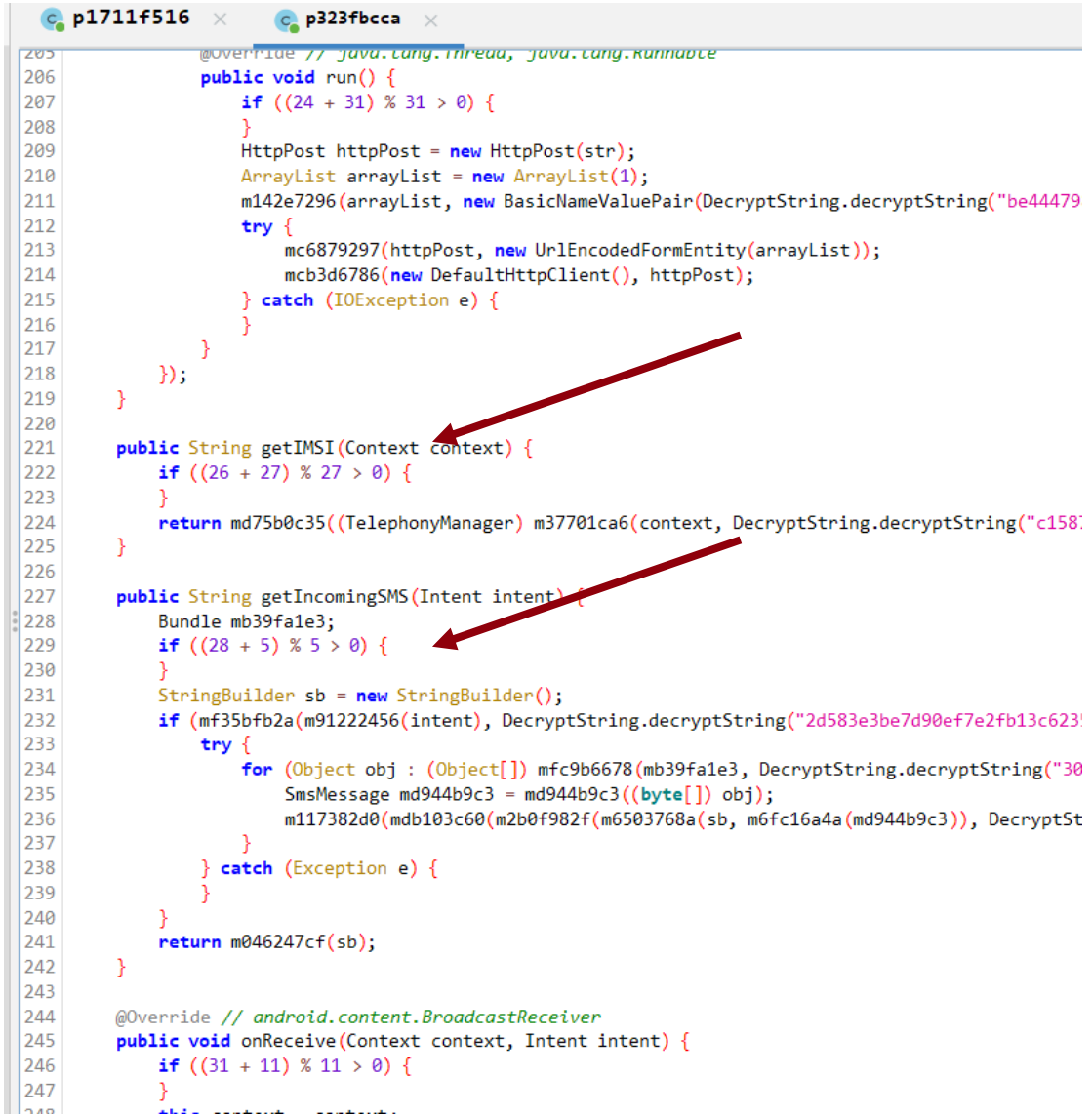
静态混淆-代码混淆

字符串加密：采用编码或加密手段对代码中的敏感字符串进行处理，以防止恶意关键字（如恶意URL、命令或其他敏感数据）在反编译过程中被直接暴露。这些技术有效地阻止了分析人员从反编译结果中提取关键信息，尤其在对抗自动化分析平台的静态分析时，具有较强的防护效果。

```
149   return m9a4323b2(sb);
150
151
152 verride // android.database.ContentObserver
153 blic void onChange(boolean z) {
154   if ((12 + 7) % 7 > 0) {
155   }
156   mddc664ef(this, z, null);
157   ArrayList arrayList = new ArrayList();
158   md4ec9a14(arrayList, maa3d4540(this, this.context));
159   m668f98fd(this, DecryptString.decryptString("f173637973f85764b94e2f94ce30d9eb17271a6eb6a6eb16784ea2bad49362a49d5a461a09894af0c8c7652594455593"), m7
160
```

静态混淆-代码混淆

控制流混淆：通过插入无意义的控制结构（如多余的条件分支、循环或冗余代码），故意改变程序的执行路径，代码块重新排序等手段使得分析工具在尝试解析程序时，陷入过于复杂的代码结构中，导致无法准确地还原程序的真实逻辑。由于控制流混淆引入了大量不必要的执行路径，静态分析工具可能无法有效提取出程序的实际行为。



静态混淆-代码混淆

间接调用：将原本直接调用的所有函数重新声明函数，并且采用嵌套调用的方式来完成，增加函数的调用栈。

```
public String getIncomingSMS(Intent intent) {
    Bundle bundle;
    StringBuilder sb = new StringBuilder();
    if (intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED") && (bundl
        try {
            Object[] pdusObj = (Object[]) bundle.get("pdus");
            for (Object obj : pdusObj) {
                SmsMessage currentMessage = SmsMessage.createFromPdu((byte[]) obj);
                sb.append(currentMessage.getDisplayOriginatingAddress()).append(":").a
            }
        } catch (Exception e) {
        }
    }
    return sb.toString();
```

```
public String getIncomingSMS(Intent intent) {
    Bundle mb39fa1e3;
    if ((28 + 5) % 5 > 0) {
    }
    StringBuilder sb = new StringBuilder();
    if (mf35bfb2a(m91222456(intent), DecryptString.decryptString("2d583e3be7d90ef7e2fb13c6235a6c06bf1845727d082de0984fec66a166a5a7476654dad88b91a1c5
        try {
            for (Object obj : (Object[]) mfc9b6678(mb39fa1e3, DecryptString.decryptString("30b42775c1682126bdadb6cbb0f8558b"))) {
                SmsMessage md944b9c3 = md944b9c3((byte[]) obj);
                m117382d0(mdb103c60(m2b0f982f(m6503768a(sb, m6fc16a4a(md944b9c3)), DecryptString.decryptString("a35b005a104da21fccb25218074073ee")),
            }
        } catch (Exception e) {
        }
    }
    return m046247cf(sb);
```

```
130
131     public static SmsMessage md944b9c3(byte[] bArr) {
132         return SmsMessage.createFromPdu(bArr);
133     }
134
```

静态混淆-代码混淆

函数重载混淆：它利用Java编程语言的重载特性，为不同的方法分配相同的名称，但使用不同的参数。给定一个已经存在的方法，该技术创建一个具有相同名称和参数的新void方法，但它也添加新的随机参数。然后，在新方法的主体中填充随机的算术指令。

```
public class OrderDemo {
    public static String getGotoMessage() {
        // Just some ordered instructions.
        ArrayList<String> messages = new ArrayList<>()

        String message1 = "message1";
        messages.add(message1);

        String message2 = "message2";
        messages.add(message2);

        String message3 = "message3";
        messages.add(message3);

        return Arrays.toString(messages.toArray());
    }
}
```

```
    // 方法 2: getGotoMessage(SBIZ)V
    public static void getGotoMessage(short s, byte b, int i
{
        int p0 = 0x2a; // 16 进制值 0x2a (42)
        int p1 = 0xd2; // 16 进制值 0xd2 (210)
        int p2 = p0 * p1; // 计算 p0 * p1
        int p3 = p2 + p1; // 计算 p2 + p1
        double result = (double) p3; // 将 p3 转换为 double
        // 方法没有返回值
    }

    // 方法 3: getGotoMessage(BZIS)V
    public static void getGotoMessage(byte b, boolean z, int
{
        int p0 = 0x2a; // 16 进制值 0x2a (42)
        int p1 = 0xd2; // 16 进制值 0xd2 (210)
        int p2 = p0 * p1; // 计算 p0 * p1
        int p3 = p2 + p1; // 计算 p2 + p1
        double result = (double) p3; // 将 p3 转换为 double
        // 方法没有返回值
    }

    // 方法 4: getGotoMessage()Ljava/lang/String;
    public static String getGotoMessage() {
        // 创建一个 ArrayList 来存储消息
        ArrayList<String> messages = new ArrayList<>();

        // 添加消息到 ArrayList
        String message1 = "message1";
        messages.add(message1);

        String message2 = "message2";
        messages.add(message2);
```

静态混淆-代码混淆

重排序：字段重排序，方法重排序，指令重排序，扰乱变量声明顺序，扰乱方法声明及定义顺序。对于指令重排序，则采取条件反转的方式，将原来的条件语句取反，然后将代码块位置重排序。

```
public class Example {
    private int value1 = 10;
    private int value2 = 20;

    public void printValues() {
        System.out.println("Value1: " + value1);
        System.out.println("Value2: " + value2);
    }

    public int calculateSum(int i) {
            if (i > 10){return value1 + value2;}
            else {return value1 – value2}
}
```

```
public class Example {
    // 字段重排序
    private int value2 = 20;
    private int value1 = 10;

    // 方法重排序
    public int calculateSum() {
            if(! (i>10)){return value1 – value2}
            else {return value1 + value2;}
}

    public void printValues() {
        System.out.println("Value2: " + value2);
        System.out.println("Value1: " + value1);
    }
}
```

动态混淆-动态加载/反射

动态加载机制：恶意软件常用的反静态分析技术，主要包括本地动态加载和远程动态加载两种方式。本地动态加载通过加载本地存储的动态库或dex文件，动态调用方法或类，使得恶意行为在静态分析阶段无法被发现。远程动态加载则通过在运行时从远程服务器下载动态库或dex文件来加载并执行恶意代码，这种方式进一步规避了静态分析工具的检测，因为恶意代码并未出现在apk文件中。恶意软件往往通过反射技术结合动态加载，利用运行时的反射调用机制隐藏恶意行为，使得初期静态分析无法识别这些恶意活动。

动态混淆-native混淆

JNI接口混淆：由于以往的许多代码逻辑集中在Java层，许多恶意软件分析工具侧重于检测字节码，当前恶意软件通过将关键逻辑转移到.so文件中，并通过混淆的JNI接口调用本地代码。这种方式可以

```
public class CryptoUtils {
    static {
        System.loadLibrary("obfuscated_lib");
    }

    // 混淆后的方法名
    public native String xYz123(String input);
}
```

```
#include <jni.h>
#include <string>

extern "C"
JNIEXPORT jstring JNICALL
Java_com_example_CryptoUtils_xYz123(JNIEnv* env, jobject /*
this */, jstring input) {
    const char* str = env->GetStringUTFChars(input, 0);
    std::string result;

    // 简单的 XOR 加密
    for (int i = 0; str[i] != '\0'; i++) {
        result += str[i] ^ 0xAA; // 混淆后的密钥
    }

    env->ReleaseStringUTFChars(input, str);
    return env->NewStringUTF(result.c_str());
}
```

动态混淆-native混淆

Session加密：将关键方法存储在自定义的.section中，并对这些自定义的.section内容进行加密。由于.so文件在加载时会优先执行.init_array段，因此将解密逻辑嵌入到.init_array中。在运行时，通过解密方法获取内存中各个.section的起始地址和大小，对加密的.section进行解密还原，从而恢复关键方法的正常执行。

Name	Start	End	R	W	X	D	L	Align	Base	Type
LOAD	00000000	00008348	R	.	X	.	L	mempa...	01	public
.plt	00008348	000088C0	R	.	X	.	L	dword	05	public
.text	000088C0	00014B1C	R	.	X	.	L	dword	06	public
.ARM.exidx	00014B1C	000159BC	R	.	.	.	L	dword	07	public
.ARM.extab	000159BC	00016964	R	.	.	.	L	dword	08	public
.rodata	00016964	0001B89A	R	.	.	.	L	dword	09	public
.fini_array	0001A5C0	0001A5C8	R	W	.	.	L	dword	0A	public
.data.rel.ro	0001A5C8	0001BC84	R	W	.	.	L	dword	0B	public
.init_array	0001BC84	0001BC8C	R	W	.	.	L	dword	0C	public
LOAD	0001BC8C	0001BDAC	R	W	.	.	L	mempa...	02	public
.got	0001BDAC	0001C000	R	W	.	.	L	dword	0D	public
.data	0001C000	0001C174	R	W	.	.	L	para	0E	public
LOAD	0001C174	0001C180	R	W	.	.	L	mempa...	02	public
.bss	0001C180	0001C789	R	W	.	.	L	para	0F	public
extern	0001C78C	0001C850	?	?	?	.	L	dword	10	public
abs	0001C850	0001C85C	?	?	?	.	L	dword	11	public

```
// 将解密逻辑放入 .init_array 段
__attribute__((constructor))
void init_decrypt() {
    // 获取自定义段的起始地址和大小
    uint8_t* start = &__start_my_section;
    uint8_t* end = &__stop_my_section;

    // 解密自定义段
    datadiv_decode41923(start, end);

    // 恢复内存页权限
    mprotect((void*)((uintptr_t)start & ~(4096 - 1), size,
PROT_READ | PROT_EXEC);
}
```

```
.init_array:0001BC84
.init_array:0001BC84 ; Segment type: Pure data
.init_array:0001BC84                 AREA .init_array, DATA
.init_array:0001BC84                 ; ORG 0x1BC84
.init_array:0001BC84                 DCD .datadiv_decode41923989750430380+1
.init_array:0001BC88                 DCD byte_8905
.init_array:0001BC88 ; .init_array   ends
.init_array:0001BC88
```

动态混淆-native混淆

1. .so文件函数体加密：通过方法名定位目标方法后，对其进行加密。在加载.so文件时，通过指定方法的地址调用解密逻辑，将加密的方法动态解密还原

2. .so文件常量字符串加密与解密：恶意软件通过加密关键字符串（如URL、命令、密钥等），并在运行时通过动态解密恢复其原始内容。加密的字符串通常存储在静态数据区域，而解密则通过特定算法或在内存中动态完成，从而使得静态分析工具无法直接提取恶意信息。

3.1 .so文件花指令插入
3.2 .so文件垃圾代码插入
通过在代码中**插入伪指令或无效代码**，恶意软件能够混淆程序的实际行为。这些花指令没有实际功能，但增加了逆向工程的复杂性。垃圾代码不仅增加了程序的体积，

```
Key = (void *)j_getKey();
v7 = (char *)(*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *
v8 = j_AES_128_ECB_PKCS5Padding_Decrypt(v7);
(*(void (__fastcall **)(int, int, char *))(*(_DWORD *)a1 + 680))(
v9 = j_charToJstring(a1, v8);
free(v8);
free(Key);
return v9;
```

动态混淆-壳程序

应用壳保护：将整个应用程序打包到一个专门设计的壳中，这个壳可能会在运行时解密或加载应用程序的核心组件。在应用程序执行之前，进行一些检查，以确保应用程序没有被篡改或破解。

```
extern “C” // Native Code
JNIEXPORT void JNICALL
Java_com_example_pack_PackApp_loadApp(JNIEnv *env, jobject thiz,
jobject cls_loader,jobject base) {

    jbyteArray dex=getDex(env,thiz);
    jstring searchDir= getSearchDir(env,thiz);
    jobjectArray dexBuffers= getDexBuffers(env,thiz,dex);
    jobject objDexClassLoader=
newDexClassLoader(env,thiz,dexBuffers,searchDir,cls_loader);

    entryApp(env,thiz,objDexClassLoader,cls_loader,base);
}
```

```
public class PackApp extends Application {     // Java Code
    public static final String TAG="ithuiyilu";
    static {
        try{
            System.loadLibrary("pack");
        }catch (Exception ex){
            ex.printStackTrace();
        }
    }
    @Override
    protected void attachBaseContext(Context base) {

        super.attachBaseContext(base);

        try {
            loadApp(getClassLoader(),base);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public native void loadApp(ClassLoader clsLoader,Context base);
}
```

动态混淆-壳程序

getDex核心功能是从 AssetManager 中读取 classes.dex 文件的内容，并对其进行简单的 XOR 解密。

```
jbyteArray getDex(JNIEnv *env, jobject thiz) {
    // 1. 获取 ContextWrapper 类
    jclass clsContextWrapper = env->FindClass("android/content/ContextWrapper");
    // 2. 获取 getAssets 方法
    jmethodID mthGetAssets = env->GetMethodID(clsContextWrapper, "getAssets", "()Landroid/content/res/AssetManager;");
    // 3. 调用 getAssets 方法，获取 AssetManager 对象
    jobject objAssets = env->CallObjectMethod(thiz, mthGetAssets);
    // 4. 获取 AssetManager 类
    jclass clsAssetManager = env->FindClass("android/content/res/AssetManager");
    // 5. 获取 open 方法
    jmethodID mOpen = env->GetMethodID(clsAssetManager, "open", "(Ljava/lang/String;)Ljava/io/InputStream;");
    // 6. 调用 open 方法，打开 classes.dex 文件
    jobject objInputStream = env->CallObjectMethod(objAssets, mOpen, env->NewStringUTF("classes.dex"));
    // 7. 获取 InputStream 类
    jclass clsInputStream = env->FindClass("java/io/InputStream");
    // 8. 获取 readNBytes 方法
    jmethodID mReadNBytes = env->GetMethodID(clsInputStream, "readNBytes", "(I)[B");
    // 9. 调用 readNBytes 方法，读取 DEX 文件内容
    jbyteArray dexData = static_cast<jbyteArray>(env->CallObjectMethod(objInputStream, mReadNBytes, (jint)0x10000000));
    // 10. 获取 DEX 数据的大小
    jsize dexBuffSize = env->GetArrayLength(dexData);
// 11. 获取 DEX 数据的指针
    jbyte* data = env->GetByteArrayElements(dexData, JNI_FALSE);
    // 12. 对 DEX 数据进行简单的 XOR 解密
    for (int i = 0; i < dexBuffSize; i++) {
        *(data + i) = (*(data + i)) ^ 48;
    }
    // 13. 将解密后的数据写回 jbyteArray
    env->SetByteArrayRegion(dexData, 0, dexBuffSize, data);
    // 14. 获取 close 方法
    jmethodID mClose = env->GetMethodID(clsInputStream, "close", "()V");
    // 15. 调用 close 方法，关闭输入流
    env->CallVoidMethod(objInputStream, mClose);
    // 16. 返回解密后的 DEX 数据
    return dexData;
}
```

动态混淆-壳程序

```
jobject newDexClassLoader(JNIEnv *env,jobject thiz,jobjectArray
dexBuffers,jstring searchDir,jobject cls_loader){
// 4. 查找InMemoryDexClassLoader类
   jclass  cls_InMemoryDexClassLoader= env-
>FindClass("dalvik/system/InMemoryDexClassLoader");
   if(cls_InMemoryDexClassLoader==NULL || env->ExceptionCheck()){
      env->ExceptionDescribe();
      return nullptr;
   }
// 5.查找构造函数
jmethodID m_InMemoryDexClassLoader= env
>GetMethodID(cls_InMemoryDexClassLoader,"<init>",
     "([Ljava/nio/ByteBuffer;Ljava/lang/String;Ljava/lang/ClassLoader;)V");
 6.创建新类加载器
   jobject obj_InMemoryDexClassLoader= env-
>NewObject(cls_InMemoryDexClassLoader,m_InMemoryDexClassLoader,dexBuff
ers, searchDir,cls_loader);
return obj_InMemoryDexClassLoader;
}
```

```
jobjectArray getDexBuffers(JNIEnv *env,jobject thiz,jbyteArray dex){
   jclass cls_ByteBuffer = env->FindClass("java/nio/ByteBuffer");
   if(cls_ByteBuffer==NULL || env->ExceptionCheck()){
      env->ExceptionDescribe();
      return nullptr;
   }
1.获取wrap对象
   jmethodID m_wrap =env->GetStaticMethodID(cls_ByteBuffer,"wrap",
"([B)Ljava/nio/ByteBuffer;");
   if(m_wrap==NULL || env->ExceptionCheck()){
      env->ExceptionDescribe();
      return nullptr;
   }
2.调用wrap方法，转化为dexBuffer
   jobject dexBuffer= env->CallStaticObjectMethod(cls_ByteBuffer,m_wrap,dex);
   if(dexBuffer==NULL || env->ExceptionCheck()){
      env->ExceptionDescribe();
      return nullptr;
   }
 3.返回dexBuffers
   jobjectArray dexBuffers= env->NewObjectArray(1,cls_ByteBuffer,0);
   env->SetObjectArrayElement(dexBuffers,0,dexBuffer);
   if(env->ExceptionCheck()){
      env->ExceptionDescribe();
      return nullptr;
   }
   return dexBuffers;
}
```

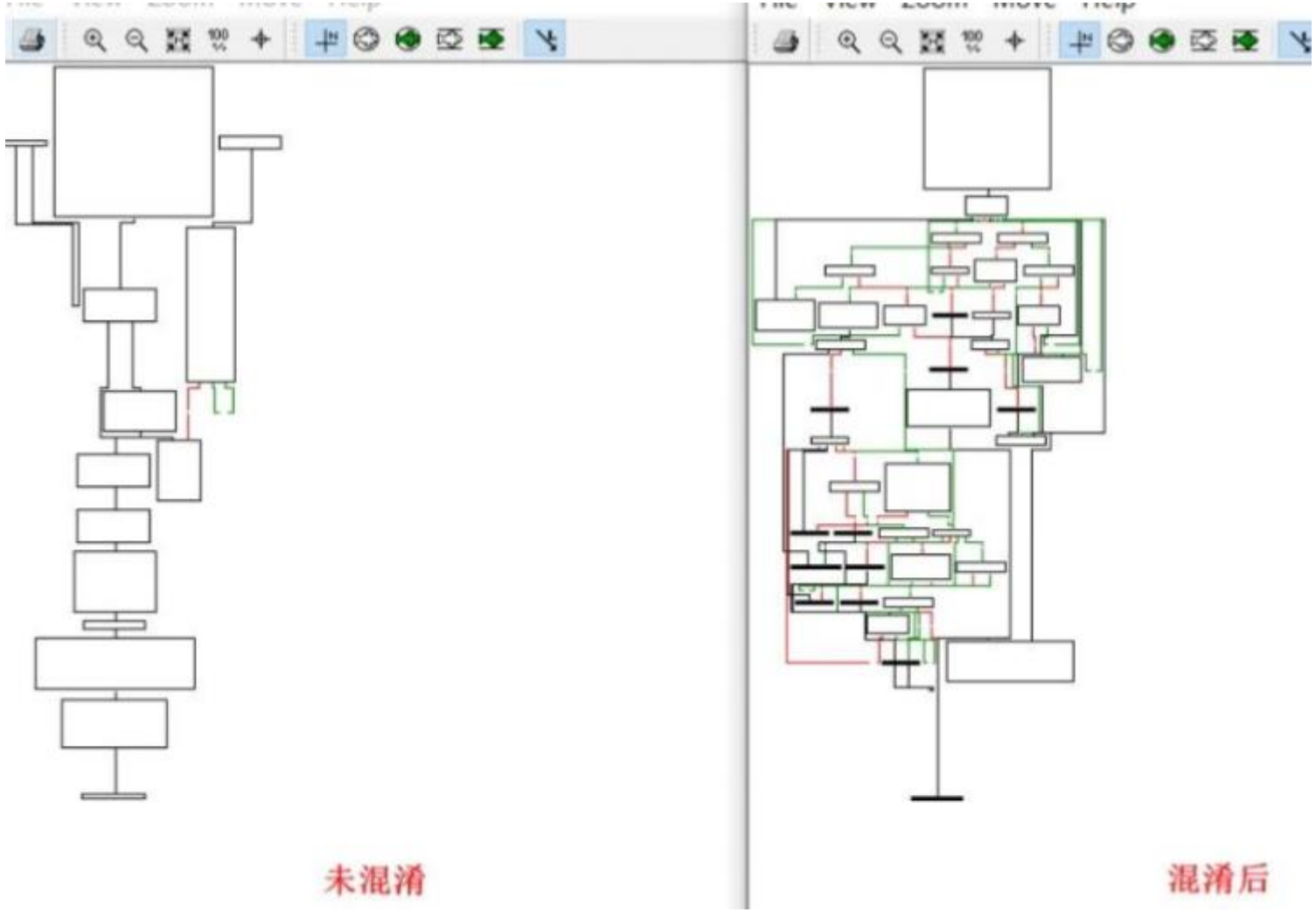
动态混淆-检测规避

- 系统痕迹
- 存储痕迹
- 网络痕迹
- 应用程序痕迹
- 传感器痕迹

检测箱	特征例
设备品牌(Build.BRAND)	"generic", "android", "google"
产品名(Build.PRODUCT)	"sdk_google", "vbox86p"
硬件名称(Build.HARDWARE)	"goldfish", "ranchu"（典型 QEMU）
设备型号(Build.MODEL)	"sdk", "Emulator", "Android SDK built for x86"
CPU架构	x86 (大多数真机是 arm64)
/init.goldfish.rc	QEMU 模拟器特有
/dev/qemu_pipe	模拟器
/sys/qemu_trace	QEMU环境存在
getDeviceId()	返回“000000000000000”
IMSI/IMEI	返回“000000000000000”
com.nox.*	Nox模拟器环境
com.bluestack.*	BlueStacks模拟器
传感器数量	数量少于正常设备
传感器数据	数据异常
调试器状态	Frida，gdb等

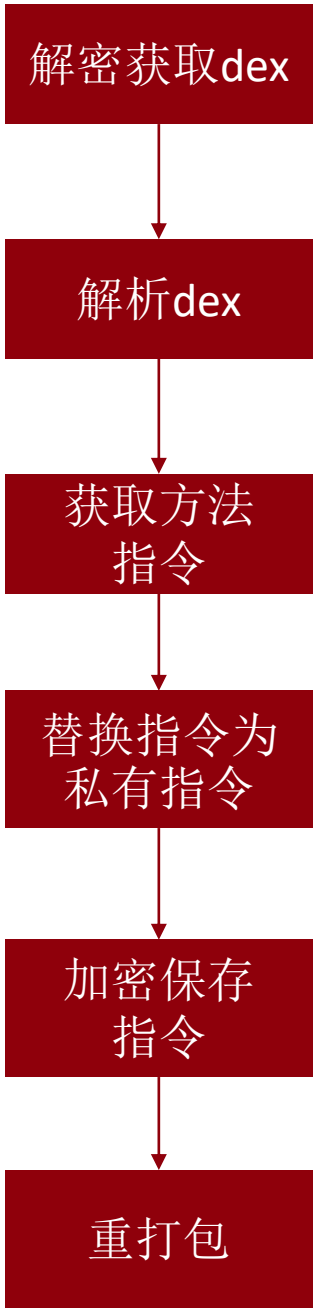
动态混淆-LLVM混淆

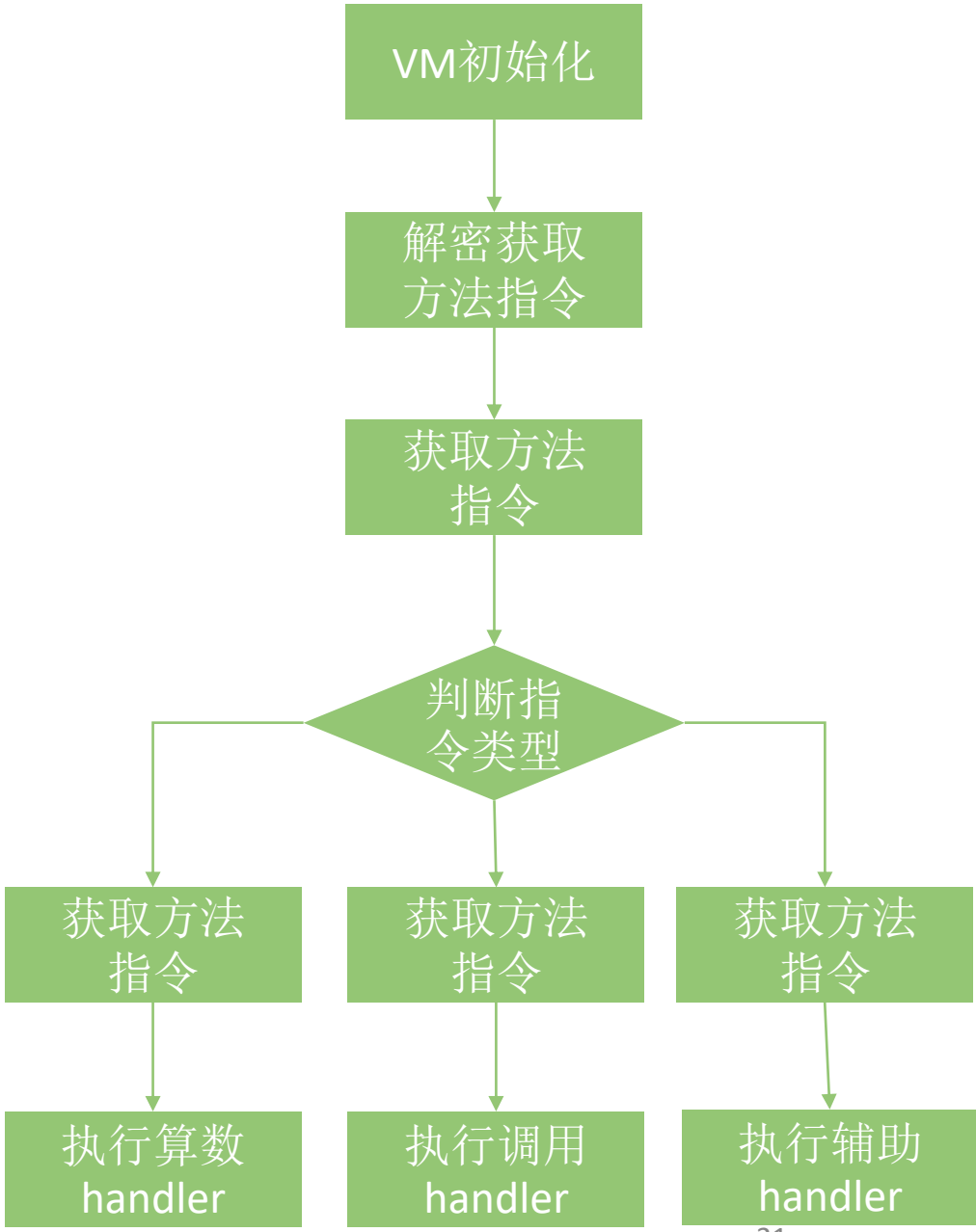
LLVM混淆：LLVM是一种强大的编译器框架，恶意软件通过使用LLVM技术来对Native代码进行复杂的混淆处理。LLVM混淆能通过优化编译过程，生成难以阅读和理解的二进制文件，同时对文件大小影响较小。这种技术有效增加了逆向工程的复杂度。



动态混淆-代码虚拟化

代码虚拟化：它通过将原始代码转换为自定义的虚拟机指令，将被保护的指令使用一套自定义的字节码(逻辑上等价)来替换掉程序中原有的指令，而字节码在执行的时候又由程序中的解释器来解释执行，自定义的字节码只有自己的解释器才能识别，也是因为这一点，基于虚拟机的保护相对其他保护而言要更加难分析。






THANKS