第十二章 逆向实战

本书内容的展开，从安卓源码编译再到系统功能内置，从源码分析到独有功能添加，无一不是遵循由简入繁，原理与实践相结合，因为笔者始终认为这才是系统化学习知识的最佳方案。

在本书最后一章中，将结合前文所学知识进行一个案例实战。在实战过程中，首先需要对需求进行分析，并将需要实现的目标进行拆分。然后逐一实现这些目标，并测试其效果。在实现的过程中，思考实现的思路和方向是至关重要的。随后可以深入追寻源码，在其中寻找切入点以便更好地完成任务。

12.1 案例实战

JniTrace是一个常用的逆向分析工具，基于frida实现。在调用本地函数时，C++代码可以通过JNI访问Java类、成员和函数。而JniTrace工具主要用于监控所有JNI函数调用，并输出这些函数的调用、传递参数和返回值。

然而，由于frida过于知名，为了防止软件被调试与分析，导致大多数情况下恶意软件开发者与互联网业务软件厂商会对其进行检测。因此，在使用时经常面临各种反制手段使得无法继续分析。为了躲避检测并继续使用JniTrace, 逆向人员将其迁移到更隐蔽的框架中（如LSPosed）。

相比Hook方案，从AOSP中进行修改则完全没有Hook痕迹存在。但相应地需要更深入地理解系统，并重复编译系统以进行测试。

在本章的实战中，将讲解如何从AOSP角度完成类似JniTrace功能，并使用配置管理使其仅对目标进程和本地函数生效。

在前文中提到了处理RegisterNative输出时的问题：它会在运行时对所有进程监控，并生成大量的运行时日志输出。为优化这一点，在处理时可以通过配置获取当前进程是否为目标进程来选择性打印日志信息。同样的优化方法也适用于这个例子中的配置管理。

12.2 需求

本案例的需求是参考JniTrace，修改AOSP源码实现对JNI函数调用的监控。所以第一步，是了解JniTrace，安装该工具，并开发简单的demo来测试其对JNI函数监控的效果。

12.2.1 功能分析

首先是安装JniTrace，该工具是使用python开发的，该工具是开源的，想要分析其实现的原理也非常方便，地址：https://github.com/chame1eon/jnitrace。安装起来非常方便，使用pip安装即可。

```
pip install jnitrace
```

由于该工具是基于frida实现的，需要在手机中运行frida-server，在地址https://github.com/frida/frida/releases中下载frida-server，开发环境是AOSP 12的情况直接下载16任意版本即可。然后将其推送到手机的/data/local/tmp目录中，并运行。具体命令如下。

```
adb push ./frida-server-16.1.1-android-arm64 /data/local/tmp
adb forward tcp:27042 tcp:27042
adb shell
su
cd /data/local/tmp
chmod +x ./frida-server-16.1.1-android-arm64

// 为防止出现错误，先将selinux关闭
setenforce 0

./frida-server-16.1.1-android-arm64
```

JniTrace的启动环境准备就绪后，接下来准备测试的案例，案例实现如下。

```
public class MainActivity extends AppCompatActivity {
    static {
        System.loadLibrary("nativedemo");
    }
    private ActivityMainBinding binding;
    Button btn1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        TextView tv = binding.sampleText;
        btn1=findViewById(R.id.button);
        btn1.setOnClickListener(v->{
            tv.setText(stringFromJNI());
        });
    }
    public String demo(){
        return "hello";
    }
    public native String stringFromJNI();
}
```

修改stringFromJNI的实现，让其通过JNI调用MainActivity中的demo函数。

```
extern "C" JNIEXPORT jstring JNICALL
Java_cn_rom_nativedemo_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject obj /* this */) {
    jclass cls= env->FindClass("cn/rom/nativedemo/MainActivity");
    jmethodID mid=env->GetMethodID(cls,"demo","()Ljava/lang/String;");
    jstring data= (jstring)env->CallObjectMethod(obj,mid);
    std::string datatmp= env->GetStringUTFChars(data,nullptr);
```

```
    return env->NewStringUTF(datatmp.c_str());
}
```

案例准备就绪后，接着通过命令，让JniTrace启动应用并监控JNI的调用，操作如下。

```
jnitrace -l libnativedemo.so cn.rom.nativedemo
```

默认会以spawn的方式进行附加，所以应用会自动拉起，点击按钮触发JNI调用，JniTrace则会输出日志如下。

```
        /* TID 6996 */

    309 ms [+] JNIEnv->FindClass                            // 调用的JNI函数
    309 ms |- JNIEnv*          : 0x7d3892f610               // 参数1的类型和值
    309 ms |- char*            : 0x7c011aaf00               // 参数2的类型和值
    309 ms |:     cn/rom/nativedemo/MainActivity
    309 ms |= jclass           : 0x71    { cn/rom/nativedemo/MainActivity }// 参数
3的类型和值
    // 下面是调用的堆栈
    309 ms ---------------------------------------Backtrace----------------------
----------------
    309 ms |->       0x7c011919c4: _ZN7_JNIEnv9FindClassEPKc+0x2c
(libnativedemo.so:0x7c01183000)
    309 ms |->       0x7c011919c4: _ZN7_JNIEnv9FindClassEPKc+0x2c
(libnativedemo.so:0x7c01183000)

           /* TID 6996 */
    310 ms [+] JNIEnv->GetMethodID
    310 ms |- JNIEnv*          : 0x7d3892f610
    310 ms |- jclass           : 0x71    { cn/rom/nativedemo/MainActivity }
    310 ms |- char*            : 0x7c011aaf1f
    310 ms |:     demo
    310 ms |- char*            : 0x7c011aaf24
    310 ms |:     ()Ljava/lang/String;
    310 ms |= jmethodID        : 0x39    { demo()Ljava/lang/String; }

    310 ms ------------------------------------------------Backtrace---------------
------------------------------
    310 ms |->       0x7c01191a0c: _ZN7_JNIEnv11GetMethodIDEP7_jclassPKcS3_+0x3c
(libnativedemo.so:0x7c01183000)
    310 ms |->       0x7c01191a0c: _ZN7_JNIEnv11GetMethodIDEP7_jclassPKcS3_+0x3c
(libnativedemo.so:0x7c01183000)

           /* TID 6996 */
    311 ms [+] JNIEnv->CallObjectMethodV
    311 ms |- JNIEnv*          : 0x7d3892f610
    311 ms |- jobject          : 0x7ff8e863e8
    311 ms |- jmethodID        : 0x39    { demo()Ljava/lang/String; }
    311 ms |- va_list          : 0x7ff8e861f0
```

```
    311 ms |= jobject          : 0x85

    311 ms ----------------------------------------------------Backtrace--------
-------------------------------------------
    311 ms |->       0x7c01191adc:
_ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz+0xc4
(libnativedemo.so:0x7c01183000)
    311 ms |->       0x7c01191adc:
_ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz+0xc4
(libnativedemo.so:0x7c01183000)

           /* TID 6996 */
    313 ms [+] JNIEnv->GetStringUTFChars
    313 ms |- JNIEnv*          : 0x7d3892f610
    313 ms |- jstring          : 0x85
    313 ms |- jboolean*        : 0x0
    313 ms |= char*            : 0x7c8893f330

    313 ms ------------------------------------------------Backtrace-------------
---------------------------------
    313 ms |->       0x7c01191b4c:
_ZN7_JNIEnv17GetStringUTFCharsEP8_jstringPh+0x34 (libnativedemo.so:0x7c01183000)
    313 ms |->       0x7c01191b4c:
_ZN7_JNIEnv17GetStringUTFCharsEP8_jstringPh+0x34 (libnativedemo.so:0x7c01183000)

           /* TID 6996 */
    314 ms [+] JNIEnv->NewStringUTF
    314 ms |- JNIEnv*          : 0x7d3892f610
    314 ms |- char*            : 0x7ff8e862c1
    314 ms |:     hello
    314 ms |= jstring          : 0x99    { hello }

    314 ms ----------------------------------------Backtrace---------------------
--------------------
    314 ms |->       0x7c01191bdc: _ZN7_JNIEnv12NewStringUTFEPKc+0x2c
(libnativedemo.so:0x7c01183000)
    314 ms |->       0x7c01191bdc: _ZN7_JNIEnv12NewStringUTFEPKc+0x2c
(libnativedemo.so:0x7c01183000)
```

从日志中能非常清晰的看到JNI调用函数的具体参数和参数类型，被调用的Java函数，以及调用的堆栈等信息。在该工具分析时，能帮助逆向分析人员快速定位到JNI函数的调用位置。

12.2.2 模块划分

有了一个输出的样例作为参考后，就可以开始对该功能进行模块划分了，将一个完整的需求拆分为若干个小块，再针对每个小块逐步实现，下面是对功能进行细化的分割。

- 配置管理，在进程启动后，在Java层中，读取配置文件，该配置信息中存储着需要被监控JNI调用的进程名称，需要被监控的动态库名称，以及需要监控的native函数（监控该函数调用中触发的所有JNI），将这些信息传递到AOSP的native中，并存储在一个全局都能很方便访问到的位置。

- JNI调用分析，并进行打桩，从存储在某个全局的配置来判断当前调用是否应该输出，符合条件则打桩输出基本信息。
- 打桩函数分类，由于JNI调用的各类函数需要输出的信息不一致，但大致的输出格式一致，所以要准备几种函数来分别处理。
- 调用堆栈信息展示，为了便于追踪调用位置，所以需要输出其调用栈信息。
- 解析参数的类型和值，进行细化输出信息，参考JniTrace的输出进行优化展示。

12.3 配置管理

12.3.1 配置文件的访问权限

既然是配置管理，那么肯定是从一个文件中读取数据，而该配置文件必须符合条件是所有APP应用都有权限读取，而在Android中，每个应用都有各自的用户身份，而不同用户之间的访问权限是受限的。在Android8以前，sdcard中还没有用户访问具体目录时，只要打开sdcard权限，即可访问同一个文件。但是在当前编译的AOSP12中已经无法访问sdcard下的任意文件了。

要解决这种各类应用访问同一个配置文件，有多种解决方式。例如通过自定义系统服务来访问具体文件，这样所有进程只要调用系统服务获取配置数据即可。例如通过共享内存，也可以达到相同的效果。

在这个案例中，将采用另一种简单的方式来解决该问题。在Android中有一个特殊的目录是/data/local/tmp，下面开始简单测试，在该目录创建一个文件。

```
echo "test" > /data/local/tmp/config.json
```

接着写一个简单的案例，来尝试在该目录读取测试文件。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    String res= FileHelper.readTextFile("/data/local/tmp/mydemo");
    Log.i("MainActivity",res);
}
```

测试发现，能成功的读取到文件，这里的重点在于，该文件是shell身份创建的，不带有身份标识，所以只要有访问权限就能正常读取，selinux不会拦截该操作。而App应用创建的文件则无法进行读取。下面看看shell创建的文件和应用创建的文件之间的区别。

```
-rw-r--r-- 1 root    root    u:object_r:shell_data_file:s0
5 2023-04-13 23:19 config.json
```

```
-rw-rw-rw- 1 u0_a240 u0_a240 u:object_r:shell_data_file:s0:c240,c256,c512,c768
4 2023-04-13 23:16 mydemo
```

可以看到mydemo的setlinux安全策略限制了哪些用户才能访问该文件。因此对于配置文件的处理，只需要用shell创建即可满足条件。

12.3.2 配置文件的结构

为了访问方便，配置文件以json的格式进行存储，在执行进入应用主进程后，则读取该配置文件，然后再根据配置的值进行相应的处理。下面是该配置文件的内容。

```
[{"packageName":"cn.rom.nativedemo","isJNIMethodPrint":true,"isRegisterNativePrint
":true,"jniModuleName":"libnativedemo.so","jniFuncName":"stringFromJNI"}]
```

为了便于访问，使用一个对应的类对象来解析该配置文件，类结构定义如下。

```
public class PackageItem {
    //应用包名
    public String packageName;
    //是否打印native函数注册
    public boolean isRegisterNativePrint;
    //是否打印JNI的函数调用
    public boolean isJNIMethodPrint;
    //监控触发JNI调用的模块名
    public String jniModuleName;
    //监控触发JNI调用的函数名
    public String jniFuncName;

    public PackageItem(){
        packageName="";
        jniModuleName="";
        jniFuncName="";
    }
}
```

12.3.3 解析配置文件

当任意应用程序启动到ActivityThread中的主进程入口时，就可以执行解析配置文件逻辑，然后进行相应的处理了，而在ActivityThread中Application创建后调用的时机，和应用中的onCreate调用时机其实相差不大的，但是在测试的时候，在ActivityThread中写代码会导致每次修改后，要等待重新编译和刷机，所以完全可以选择先在正常的应用onCreate中写入要解析的代码，在最后流程完全跑通后，再将测试无误的代码放入ActivityThread中。

这里使用fastjson将配置文件内容解析成类对象，下面是解析的代码。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    String packageName= this.getPackageName();
    String configJson= FileHelper.readTextFile("/data/local/tmp/config.json");
    if(configJson.isEmpty()){
        Log.i(TAG,"not found config json "+packageName);
        return;
    }
    if(!configJson.contains("{")){
        Log.i(TAG,"config data is error "+packageName);
        return;
    }

    List<PackageItem> packageItems= JSON.parseObject(configJson,new
TypeReference<List<PackageItem>>(){});
    if(packageItems.size()<=0){
        Log.i(TAG,"not found config json parse "+packageName);
        return;
    }
}
```

12.3.4 配置参数的传递

由于JNI的调用部分是在native中进行，所以获取到的配置内容，需要将其传递到native层，并将其保存在一个可以全局访问的位置。便于后续打桩时获取配置的参数。

传递数据到native层，必然是需要新定义一个native函数，在这个案例实现中，在文件libcore/dalvik/src/main/java/dalvik/system/DexFile.java中添加了native函数实现配置数据的传递。修改如下。

```
public final class DexFile {
    ...
    @UnsupportedAppUsage
    private static native void initConfig(Object item);
}
```

接着找到其对应的实现文件art/runtime/native/dalvik_system_DexFile.cc，添加对应的实现。

```
static void
DexFile_initConfig(JNIEnv* env, jobject ,jobject item) {
    ...
}
```

```
static JNINativeMethod gMethods[] = {
  ...
  NATIVE_METHOD(DexFile, getDexFileOptimizationStatus,
                "(Ljava/lang/String;Ljava/lang/String;)[Ljava/lang/String;"),
  NATIVE_METHOD(DexFile, setTrusted, "(Ljava/lang/Object;)V"),
  NATIVE_METHOD(DexFile, initConfig,"(Ljava/lang/Object;)V"),
};
```

参数传递到native层后，需要将其保存到一个全局能够访问的位置，便于后续JNI触发时进行判断。在案例中，我选择将其存放在Runtime中。修改art/runtime/runtime.h文件如下。

```
typedef struct{
    char packageName[128];
    char jniModuleName[128];
    char jniFuncName[128];
    bool isRegisterNativePrint;
    bool isJNIMethodPrint;
    bool jniEnable;
}PackageItem;

class Runtime {
    ...
    public
    ...
        void SetConfigItem(PackageItem item){
            configItem=item;
        }

        PackageItem GetConfigItem(){
            return configItem;
        }
    ...
    private:
        ...
        PackageItem configItem;
        ...
}
```

这样在能访问到Runtime的任意地方都能获取到该配置了。现在就可以实现前面的initConfig函数了，将java传递过来的对象，转换为c++对象存储到Runtime中。具体实现如下。

```
static void
DexFile_initConfig(JNIEnv* env, jobject ,jobject item) {

    Runtime* runtime=Runtime::Current();
    // 将各字段取出
    jclass jcInfo = env->FindClass("cn/krom/PackageItem");
```

```
    jfieldID jPackageName = env->GetFieldID(jcInfo, "packageName",
"Ljava/lang/String;");
    jfieldID jJniModuleName = env->GetFieldID(jcInfo, "jniModuleName",
"Ljava/lang/String;");
    jfieldID jJniFuncName = env->GetFieldID(jcInfo, "jniFuncName",
"Ljava/lang/String;");
    jfieldID jIsRegisterNativePrint = env->GetFieldID(jcInfo,
"isRegisterNativePrint", "Z");
    jfieldID jIsJNIMethodPrint = env->GetFieldID(jcInfo, "isJNIMethodPrint", "Z");

    PackageItem citem;
    // 将java的值转换为c++的值
    jstring jstrPackageName = (jstring)env->GetObjectField(item, jPackageName);
    const char* pPackageName = (char*)env->GetStringUTFChars(jstrPackageName, 0);
    strcpy(citem.packageName, pPackageName);

    jstring jstrJniModuleName = (jstring)env->GetObjectField(item,
jJniModuleName);
    const char* pJniModuleName = (char*)env->GetStringUTFChars(jstrJniModuleName,
0);
    strcpy(citem.jniModuleName, pJniModuleName);

    jstring jstrJniFuncName = (jstring)env->GetObjectField(item, jJniFuncName);
    const char* pJniFuncName = (char*)env->GetStringUTFChars(jstrJniFuncName, 0);
    strcpy(citem.jniFuncName, pJniFuncName);

    citem.isRegisterNativePrint = env->GetBooleanField(item,
jIsRegisterNativePrint);
    citem.isJNIMethodPrint = env->GetBooleanField(item, jIsJNIMethodPrint);

    // 配置存储到全局
    runtime->SetConfigItem(citem);
}
```

到这里就成功从配置文件中读取数据，并解析后通过native函数将其存储到全局能访问的位置了。最后在成功读取配置后，通过反射调用initConfig函数，即可完成配置的初始化工作。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    String packageName= this.getPackageName();
    // 读取配置文件
    String configJson= FileHelper.readTextFile("/data/local/tmp/config.json");
    if(configJson.isEmpty()){
        Log.i(TAG,"not found config json "+packageName);
        return;
```

```
        }
        // 判断是否是json格式
        if(!configJson.contains("{")){
            Log.i(TAG,"config data is error "+packageName);
            return;
        }
        // 将json转换为对象
        List<PackageItem> packageItems= JSON.parseObject(configJson,new
    TypeReference<List<PackageItem>>(){});
        if(packageItems.size()<=0){
            Log.i(TAG,"not found config json parse "+packageName);
            return;
        }
        // 判断当前app是否为目标应用
        PackageItem currentItem=null;
        for(PackageItem item : packageItems){
            if(item.packageName.contains(this.getPackageName())){
                currentItem=item;
                break;
            }
        }
        if(currentItem==null){
            return;
        }
        // 是目标应用则反射调用初始化函数，将配置内容传递到native层。
        try {
            Class
    dexFileClazz=ClassLoader.getSystemClassLoader().loadClass("dalvik.system.DexFile")
    ;
            Method method=dexFileClazz.getMethod("initConfig",Object.class);
            method.invoke(null,currentItem);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (NoSuchMethodException e) {
            throw new RuntimeException(e);
        } catch (InvocationTargetException e) {
            throw new RuntimeException(e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
```

12.4 JNI调用分析

JNI的调用流程并不是非常复杂，env中对应的相关函数定义是在文件libnativehelper/include_jni/jni.h中，JNIEnv的定义描述如下。

```
#if defined(__cplusplus)
// 判断当前是否在C++环境下
typedef _JNIEnv JNIEnv;
typedef _JavaVM JavaVM;
```

```
#else
// 如果在C环境下
typedef const struct JNINativeInterface* JNIEnv;
typedef const struct JNIInvokeInterface* JavaVM;
#endif
```

接着看看_JNIEnv的定义描述。

```
struct _JNIEnv {
    /* do not rename this; it does not seem to be entirely opaque */
    const struct JNINativeInterface* functions;

#if defined(__cplusplus)

    ...
    void CallStaticVoidMethod(jclass clazz, jmethodID methodID, ...)
    {
        va_list args;
        va_start(args, methodID);
        functions->CallStaticVoidMethodV(this, clazz, methodID, args);
        va_end(args);
    }
    void CallStaticVoidMethodV(jclass clazz, jmethodID methodID, va_list args)
    { functions->CallStaticVoidMethodV(this, clazz, methodID, args); }
    void CallStaticVoidMethodA(jclass clazz, jmethodID methodID, const jvalue*
args)
    { functions->CallStaticVoidMethodA(this, clazz, methodID, args); }
    ...
#endif /*__cplusplus*/
};
```

可以看到虽然c++的情况下是使用结构体进行一层包装，但是最终实际也调用的JNINativeInterface下的函数实现。继续看看该结构体的定义。

```
struct JNINativeInterface {
    void*       reserved0;
    void*       reserved1;
    void*       reserved2;
    void*       reserved3;
    ...
    jmethodID   (*GetMethodID)(JNIEnv*, jclass, const char*, const char*);
    jobject     (*CallObjectMethod)(JNIEnv*, jobject, jmethodID, ...);
    jobject     (*CallObjectMethodV)(JNIEnv*, jobject, jmethodID, va_list);
    jobject     (*CallObjectMethodA)(JNIEnv*, jobject, jmethodID, const jvalue*);
    ...
};
```

根据上面源码分析，能够看到JNIEnv实际就是JNINativeInterface的指针，而该指针对应的结构体中存储着函数表，接下来看是如何给functions进行赋值的。

```
JNIEnvExt::JNIEnvExt(Thread* self_in, JavaVMExt* vm_in, std::string* error_msg)
    : self_(self_in),
      vm_(vm_in),
      local_ref_cookie_(kIRTFirstSegment),
      locals_(kLocalsInitial, kLocal,
IndirectReferenceTable::ResizableCapacity::kYes, error_msg),
      monitors_("monitors", kMonitorsInitial, kMonitorsMax),
      critical_(0),
      check_jni_(false),
      runtime_deleted_(false) {
  MutexLock mu(Thread::Current(), *Locks::jni_function_table_lock_);
  check_jni_ = vm_in->IsCheckJniEnabled();
  // 函数指针赋值
  functions = GetFunctionTable(check_jni_);
  unchecked_functions_ = GetJniNativeInterface();
}
```

继续分析GetFunctionTable的实现。

```
const JNINativeInterface* JNIEnvExt::GetFunctionTable(bool check_jni) {
  const JNINativeInterface* override = JNIEnvExt::table_override_;
  if (override != nullptr) {
    return override;
  }
  return check_jni ? GetCheckJniNativeInterface() : GetJniNativeInterface();
}
```

继续进入查看GetJniNativeInterface的实现逻辑。

```
const JNINativeInterface* GetJniNativeInterface() {
  return Runtime::Current()->GetJniIdType() == JniIdType::kPointer
             ? &JniNativeInterfaceFunctions<false>::gJniNativeInterface
             : &JniNativeInterfaceFunctions<true>::gJniNativeInterface;
}
```

到这里就对JNI对应函数进行赋值了。

```
template<bool kEnableIndexIds>
struct JniNativeInterfaceFunctions {
  using JNIImpl = JNI<kEnableIndexIds>;
  static constexpr JNINativeInterface gJniNativeInterface = {
    nullptr,  // reserved0.
    nullptr,  // reserved1.
```

```
    nullptr,  // reserved2.
    nullptr,  // reserved3.
    ...
    JNIImpl::GetMethodID,
    JNIImpl::CallObjectMethod,
    JNIImpl::CallObjectMethodV,
    JNIImpl::CallObjectMethodA,
    ...
  };
};
```

根据上面的源码分析，知道了JNI中函数对应的实现就在文件art/runtime/jni/jni_internal.cc中实现。明白了这个原理后，接下来添加一个打桩函数简单的输出信息，来确定流程是否正确。

```
static jobject CallObjectMethodV(JNIEnv* env, jobject obj, jmethodID mid, va_list
args) {
    CHECK_NON_NULL_ARGUMENT(obj);
    CHECK_NON_NULL_ARGUMENT(mid);
    ALOGD("jnitrace %s",__FUNCTION__);
    ScopedObjectAccess soa(env);
    JValue result(InvokeVirtualOrInterfaceWithVarArgs(soa, obj, mid, args));
    return soa.AddLocalReference<jobject>(result.GetL());
  }
```

编译后成功看到了大量该函数调用的日志，接下来需要封装一个函数，在这个函数中根据前文传递的配置进行判断是否需要输出JNI调用的相关信息。符合条件才进行打桩。具体实现如下。

```
void ShowVarArgs(const ScopedObjectAccessAlreadyRunnable& ,
                 const char* funcname,
                 jmethodID ,
                 va_list ){
    // 从Runtime获取配置信息，配置了需要打印JNI，并且当前符合输出条件才进行打桩
    Runtime* runtime=Runtime::Current();
    if(!runtime->GetConfigItem().isJNIMethodPrint ||!runtime-
>GetConfigItem().jniEnable){
        return;
    }
    // 打桩信息
    ALOGD("jnitrace ShowVarArgs %s %s %s %s %p",runtime-
>GetProcessPackageName().c_str(),runtime->GetConfigItem().jniModuleName,
          runtime->GetConfigItem().jniFuncName,funcname,Thread::Current());

}
```

这里使用了两个条件来控制打桩，isJNIMethodPrint表示是否要对JNI监控打桩，而jniEnable则表示，当前是否应该打桩，例如在指定的native函数调用期间，才打桩输出，或者指定动态库加载后，才进行打桩，这个过滤条件大大的降低了对无效日志的输出，提高分析的效率。

jniEnable默认是false的，值不应由配置文件决定，而是在调用过程中进行赋值，所有native函数开始执行和执行结束时都会经过JniMethodStart和JniMethodEnd函数，所以只需要在进入该函数时，将该字段修改为true，在结束时，再将其关闭即可。下面是实现代码。

```
extern uint32_t JniMethodStart(Thread* self) {
  JNIEnvExt* env = self->GetJniEnv();
  DCHECK(env != nullptr);
  uint32_t saved_local_ref_cookie = bit_cast<uint32_t>(env->GetLocalRefCookie());
  env->SetLocalRefCookie(env->GetLocalsSegmentState());
  //add
  Runtime* runtime=Runtime::Current();
  if(runtime->GetConfigItem().isJNIMethodPrint){
      ArtMethod* native_method = *self->GetManagedStack()->GetTopQuickFrame();
      std::string methodname=native_method->PrettyMethod();
      // 当前开始函数为要监控的目标函数时，则开启输出JNI
      if(strstr(methodname.c_str(),runtime->GetConfigItem().jniFuncName)){
          runtime->GetConfigItem().jniEnable=true;
          ALOGD("jnitrace enter jni %s",methodname.c_str());
      }
  }
  //endadd
  if (kIsDebugBuild) {
    ArtMethod* native_method = *self->GetManagedStack()->GetTopQuickFrame();
    CHECK(!native_method->IsFastNative()) << native_method->PrettyMethod();
  }
  // Transition out of runnable.
  self->TransitionFromRunnableToSuspended(kNative);
  return saved_local_ref_cookie;
}


extern void JniMethodEnd(uint32_t saved_local_ref_cookie, Thread* self) {
    //add
    Runtime* runtime=Runtime::Current();
    if(runtime->GetConfigItem().isJNIMethodPrint){
        ArtMethod* native_method = *self->GetManagedStack()->GetTopQuickFrame();
        std::string methodname=native_method->PrettyMethod();
        // 当前结束函数为要监控的目标函数时，则关闭输出JNI
        if(strstr(methodname.c_str(),runtime->GetConfigItem().jniFuncName)){
            runtime->GetConfigItem().jniEnable=false;
            ALOGD("jnitrace leave jni %s",methodname.c_str());
        }
    }
    //endadd

  GoToRunnable(self);
  PopLocalReferences(saved_local_ref_cookie, self);
}


static mirror::Object* JniMethodEndWithReferenceHandleResult(jobject result,
                                                              uint32_t
```

```
    saved_local_ref_cookie,
                                                        Thread* self)
        NO_THREAD_SAFETY_ANALYSIS {
        //add
        Runtime* runtime=Runtime::Current();
        if(runtime->GetConfigItem().isJNIMethodPrint){
            ArtMethod* native_method = *self->GetManagedStack()->GetTopQuickFrame();
            std::string methodname=native_method->PrettyMethod();
            if(strstr(methodname.c_str(),runtime->GetConfigItem().jniFuncName)){
                runtime->GetConfigItem().jniEnable=false;
                ALOGD("jnitrace leave jni %s",methodname.c_str());
            }
        }
        //endadd
        ...
    }
```

12.5 打桩函数分类

前文中仅仅对其中类似CallObjectMethodV函数，进行简单的输出，而实际场景中，大量的JNI函数调用，并非有着这些参数，所以需要将ShowVarArgs进行封装，并有多种重载实现。具体重载参数需要根据实际JNI函数中有哪些参数来决定。在这里篇幅有限，所以不会将所有的JNI函数情况进行处理，主要将前文中测试中调用到的JNI函数进行对应处理。

根据JniTrace中的日志，需要对四个JNI函数进行打桩处理，分别是GetMethodID、GetStringUTFChars、NewStringUTF、CallObjectMethodV。根据这些函数对应的参数，对打桩的函数进行重载处理。

12.5.1 GetMethodID插桩

在开始修改代码前，先看看该函数的定义。

```
// 根据函数名，以及对应的函数签名来获取对应函数
static jmethodID GetMethodID(JNIEnv* env, jclass java_class, const char* name,
const char* sig);
```

再看看JniTrace对于该函数的输出。

```
            /* TID 6996 */
    310 ms [+] JNIEnv->GetMethodID
    310 ms |- JNIEnv*          : 0x7d3892f610
    310 ms |- jclass           : 0x71    { cn/rom/nativedemo/MainActivity }
    310 ms |- char*            : 0x7c011aaf1f
    310 ms |:     demo
    310 ms |- char*            : 0x7c011aaf24
    310 ms |:     ()Ljava/lang/String;
    310 ms |= jmethodID        : 0x39    { demo()Ljava/lang/String; }

    310 ms ----------------------------------------------Backtrace---------------
------------------------------
```

```
    310 ms |->       0x7c01191a0c: _ZN7_JNIEnv11GetMethodIDEP7_jclassPKcS3_+0x3c
(libnativedemo.so:0x7c01183000)
    310 ms |->       0x7c01191a0c: _ZN7_JNIEnv11GetMethodIDEP7_jclassPKcS3_+0x3c
(libnativedemo.so:0x7c01183000)
```

该输出中，关键展示了函数所在类的类名称、函数名称、函数签名，以及所找到的对应函数id，调用堆栈。参考该类型的JNI调用，下面重构一个相应的打桩函数。

```
// 是否需要打印
bool HasShow(){
    Runtime* runtime=Runtime::Current();
    if(!runtime->GetConfigItem().isJNIMethodPrint ||!runtime-
>GetConfigItem().jniEnable){
        return false;
    }
    return true;
}

// JNI打桩函数重载,针对GetMethodID进行输出
void ShowVarArgs(const ScopedObjectAccessAlreadyRunnable& soa,const char*
funcname,jclass java_class, const char* name, const char* sig,jmethodID methodID){
    if(!HasShow()){
        return;
    }
    ObjPtr<mirror::Class> c = soa.Decode<mirror::Class>(java_class);
    std::string temp;
    const char* className= c->GetDescriptor(&temp);
    ArtMethod* method = jni::DecodeArtMethod(methodID);
    pid_t pid = getpid();
    // 前面加上标志是为了方便搜索日志
    ALOGD("%s           /* TID %d */","jnitrace",pid);
    ALOGD("%s           [+] JNIEnv->%s","jnitrace",funcname);
    ALOGD("%s           |- jclass           :%s","jnitrace",className);
    ALOGD("%s           |- char*            :%p","jnitrace",name);
    ALOGD("%s           |:     %s","jnitrace",name);
    ALOGD("%s           |- char*            :%p","jnitrace",sig);
    ALOGD("%s           |:     %s","jnitrace",sig);
    ALOGD("%s           |= jmethodID        :0x%x   {%s}","jnitrace",method-
>GetMethodIndex(),method->PrettyMethod().c_str());
}
```

最后在JNI函数调用处，使用该打桩函数。

```
static jmethodID GetMethodID(JNIEnv* env, jclass java_class, const char* name,
const char* sig) {
    CHECK_NON_NULL_ARGUMENT(java_class);
    CHECK_NON_NULL_ARGUMENT(name);
    CHECK_NON_NULL_ARGUMENT(sig);
    ScopedObjectAccess soa(env);
```

```
    jmethodID result = FindMethodID<kEnableIndexIds>(soa, java_class, name, sig,
false);
    ShowVarArgs(soa,__FUNCTION__,java_class,name,sig,result);
    return result;
  }
```

12.5.2 GetStringUTFChars插桩

参考上面的流程，首先了解该函数的定义结构。

```
static const char* GetStringUTFChars(JNIEnv* env, jstring java_string, jboolean*
is_copy);
```

接着查看JniTrace的输出显示。

```
          /* TID 6996 */
    313 ms [+] JNIEnv->GetStringUTFChars
    313 ms |- JNIEnv*          : 0x7d3892f610
    313 ms |- jstring          : 0x85
    313 ms |- jboolean*        : 0x0
    313 ms |= char*            : 0x7c8893f330

    313 ms ------------------------------------------------Backtrace-------------
---------------------------------
    313 ms |->       0x7c01191b4c:
_ZN7_JNIEnv17GetStringUTFCharsEP8_jstringPh+0x34 (libnativedemo.so:0x7c01183000)
    313 ms |->       0x7c01191b4c:
_ZN7_JNIEnv17GetStringUTFCharsEP8_jstringPh+0x34 (libnativedemo.so:0x7c01183000)
```

看的出来这个函数非常的简单，主要是对返回值进行输出即可。添加打桩函数如下。

```
void ShowVarArgs(const ScopedObjectAccessAlreadyRunnable& ,
                 const char* funcname,
                 jboolean* is_copy ,
                 const char* data ){
    if(!HasShow()){
        return;
    }
    pid_t pid = getpid();
    ALOGD("%s           /* TID %d */","jnitrace",pid);
    ALOGD("%s           [+] JNIEnv->%s","jnitrace",funcname);
    if(is_copy== nullptr){
        ALOGD("%s           |- jboolean*        : %d","jnitrace",false);
    }else{
        ALOGD("%s           |- jboolean*        : %d","jnitrace",*is_copy);
    }
```

```
    ALOGD("%s           |= char*           : %s","jnitrace",data);
}
```

修改原调用函数如下。

```
  static const char* GetStringUTFChars(JNIEnv* env, jstring java_string, jboolean*
is_copy) {
    ...
    bytes[byte_count] = '\0';
    ShowVarArgs(soa,__FUNCTION__,is_copy,bytes);
    return bytes;
  }
``
`

### 12.5.3 NewStringUTF插桩

该函数同样非常简单，和上一个函数相反，只需要将参数打印即可，无需处理返回值，函数定义如下。

```c++
static jstring NewStringUTF(JNIEnv* env, const char* utf);
```

接着看JniTrace的输出，同样非常简单。

```
          /* TID 6996 */
    314 ms [+] JNIEnv->NewStringUTF
    314 ms |- JNIEnv*          : 0x7d3892f610
    314 ms |- char*            : 0x7ff8e862c1
    314 ms |:     hello
    314 ms |= jstring          : 0x99    { hello }
```

添加对应打桩函数如下。

```
void ShowVarArgs(const ScopedObjectAccessAlreadyRunnable& ,
                 const char* funcname,
                 const char* data ){
    if(!HasShow()){
        return;
    }
    pid_t pid = getpid();
    ALOGD("%s           /* TID %d */","jnitrace",pid);
    ALOGD("%s           [+] JNIEnv->%s","jnitrace",funcname);
    ALOGD("%s           |- char*        : %d","jnitrace",data);
}
```
```

调整原函数调用该打桩如下。

```
static jstring NewStringUTF(JNIEnv* env, const char* utf) {
    ...
    ScopedObjectAccess soa(env);
    ShowVarArgs(soa,__FUNCTION__,utf);
    ObjPtr<mirror::String> result =
        mirror::String::AllocFromModifiedUtf8(soa.Self(), utf16_length, utf,
utf8_length);
    return soa.AddLocalReference<jstring>(result);
}
```

12.5.4 CallObjectMethodV插桩

这个JNI函数不同于前面几种函数，在前几个函数中，参数是明确固定的，而CallObjectMethodV是通过JNI，调用一个java函数，而为此java函数提供的所有参数的类型，以及参数个数。都是未知的。而这些参数的信息同样是需要打桩展示出来的。

将测试样例中，被调用的java函数进行调整，将测试函数新增参数，并且使用JniTrace观察CallObjectMethodV的输出结果。样例函数修改如下。

```
public String demo(int a,float b,long c,String d){
    return a+b+c+d;
}
```

同时修改native函数中使用JNI调用的逻辑。

```
extern "C" JNIEXPORT jstring JNICALL
Java_cn_rom_nativedemo_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject obj /* this */) {
    jclass cls= env->FindClass("cn/rom/nativedemo/MainActivity");
    jmethodID mid=env->GetMethodID(cls,"demo", "
(IFLjava/lang/String;)Ljava/lang/String;");
    jstring c=env->NewStringUTF("newdemo");
    jstring data= (jstring)env->CallObjectMethod(obj,mid,1,2.0f,c);
    std::string datatmp= env->GetStringUTFChars(data,nullptr);
    return env->NewStringUTF(datatmp.c_str());
}
```

再次使用JniTrace观察到的CallObjectMethodV输出如下。

```
            /* TID 18863 */
   2169 ms [+] JNIEnv->CallObjectMethodV
   2169 ms |- JNIEnv*          : 0x704e856090
```

```
   2169 ms |- jobject          : 0x7fe4d55d38
   2169 ms |- jmethodID        : 0x3d    {
demo(IFLjava/lang/String;)Ljava/lang/String; }
   2169 ms |- va_list          : 0x7fe4d55b30
   2169 ms |:     jint         : 1
   2169 ms |:     jfloat       : 2
   2169 ms |:     jstring      : 0x81    { newdemo }
   2169 ms |= jobject          : 0x95    { java/lang/String }

   2169 ms ----------------------------------------------------Backtrace--------
-----------------------------------------------
   2169 ms |->       0x6f5d458ae4:
_ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz+0xc4
(libnativedemo.so:0x6f5d44a000)
   2169 ms |->       0x6f5d458ae4:
_ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz+0xc4
(libnativedemo.so:0x6f5d44a000)
```

从日志中能看到，参数列表被解析后将具体的值进行输出，而返回值的部分，如果是`jobject`，则将其类型进行输出。明白具体需求后，接着就可以开始根据`CallObjectMethodV`类型参数定义来准备打桩函数了。定义描述如下。

```
static jobject CallObjectMethodV(JNIEnv* env, jobject obj, jmethodID mid, va_list
args);
```

根据参考，需要输出调用的目标函数，参数，以及其返回值。而其他的类似调用函数的情况，和该函数差不多的处理，只是返回值的输出不同。优化后的打桩函数如下。

```
// 输出JNI的参数部分
void ShowVarArgs(const ScopedObjectAccessAlreadyRunnable& soa,
                 const char* funcname,
                 jmethodID mid,
                 va_list vaList){
    if(!HasShow()){
        return;
    }

    ArtMethod* method = jni::DecodeArtMethod(mid);
    pid_t pid = getpid();
    ALOGD("%s           /* TID %d */","jnitrace",pid);
    ALOGD("%s           [+] JNIEnv->%s","jnitrace",funcname);
    ALOGD("%s           |- jmethodID        :0x%x   {%s}","jnitrace",method-
>GetMethodIndex(),method->PrettyMethod().c_str());
    ALOGD("%s           |- va_list          :%p","jnitrace",&vaList);

    uint32_t shorty_len = 0;
    const char* shorty =
            method->GetInterfaceMethodIfProxy(kRuntimePointerSize)-
>GetShorty(&shorty_len);
```

```
    ArgArray arg_array(shorty, shorty_len);
    arg_array.VarArgsShowArg(soa, vaList);
}


void ShowVarArgs(const ScopedObjectAccessAlreadyRunnable& soa,
                 const char* funcname,
                 jmethodID mid,
                 va_list valist,
                 jobject ret){
    if(!HasShow()){
        return;
    }
    // 输出JNI的参数部分
    ShowVarArgs(soa,funcname,mid,valist);
    // 输出JNI的返回值
    ObjPtr<mirror::Object> receiver =soa.Decode<mirror::Object>(ret);
    if(receiver==nullptr){
        return;
    }
    ObjPtr<mirror::Class> cls=receiver->GetClass();
    if (cls->DescriptorEquals("Ljava/lang/String;")){
        ObjPtr<mirror::String> retStr =soa.Decode<mirror::String>(ret);
        ALOGD("%s           |= jstring          :%s","jnitrace",(const
char*)retStr->GetValue());
    }else{
        std::string temp;
        const char* className= cls->GetDescriptor(&temp);
        ALOGD("%s           |= jobject          :%p
{%s}","jnitrace",&ret,className);
    }
}
```

很多JNI的调用函数除了返回值的处理不同，其他调用部分都是相同的，所以将打印参数部分和打印返回值部分拆分开来，而参数va_list的解析，可以直接参考AOSP源码中，函数BuildArgArrayFromVarArgs对其的处理。实现如下。

```
void VarArgsShowArg(const ScopedObjectAccessAlreadyRunnable& soa,
                        va_list ap)
    REQUIRES_SHARED(Locks::mutator_lock_) {
            std::stringstream ss;
            for (size_t i = 1; i < shorty_len_; ++i) {
                switch (shorty_[i]) {
                    case 'Z':
                    case 'B':
                    case 'C':
                    case 'S':
                    case 'I':
                        ss<<"jnitrace"<<"           |:     jint         : "
<<va_arg(ap, jint)<<"\n";
                        break;
```

```
                    case 'F':
                        ss<<"jnitrace"<<"           |:     jfloat       : "
<<va_arg(ap, jdouble)<<"\n";
                        break;
                    case 'L':{
                        jobject obj=va_arg(ap, jobject);
                        ObjPtr<mirror::Object> receiver
=soa.Decode<mirror::Object>(obj);
                        if(receiver==nullptr){
                            ss<<"jnitrace"<<"           |:     jobject      :
null\n";
                            break;
                        }
                        ObjPtr<mirror::Class> cls=receiver->GetClass();
                        if (cls->DescriptorEquals("Ljava/lang/String;")){
                            ObjPtr<mirror::String> argStr
=soa.Decode<mirror::String>(obj);
                            ss<<"jnitrace"<<"           |:     jstring      : "
<<(const char*)argStr->GetValue()<<"\n";
                        }else{
                            ss<<"jnitrace"<<"           |:     jobject      : "
<<&obj<<"\n";
                        }
                        break;
                    }
                    case 'D':
                        ss<<"jnitrace"<<"           |:     jdouble      : "
<<va_arg(ap, jdouble)<<"\n";
                        break;
                    case 'J':
                        ss<<"jnitrace"<<"           |:     jlong        : "
<<va_arg(ap, jlong)<<"\n";
                        break;
                }
            }
            ALOGD("%s",ss.str().c_str());
    }
```

到这里案例中使用的相关JNI函数处理就添加完成了，编译后刷入测试机。当点击样例中的按钮时，最后输出效果如下所示。

```
jnitrace enter jni java.lang.String cn.rom.nativedemo.MainActivity.stringFromJNI()
0x74656a77b0
jnitrace          /* TID 5465 */
jnitrace          [+] JNIEnv->GetMethodID
jnitrace          |- jclass          :Lcn/rom/nativedemo/MainActivity;
jnitrace          |- char*           :0x7275d69f1b
jnitrace          |:     demo
jnitrace          |- char*           :0x7275d69eef
jnitrace          |:     (IFLjava/lang/String;)Ljava/lang/String;
jnitrace          |= jmethodID       :0x277   {java.lang.String
```

```
cn.rom.nativedemo.MainActivity.demo(int, float, java.lang.String)}
jnitrace           /* TID 5465 */
jnitrace           [+] JNIEnv->NewStringUTF
jnitrace           |- char*         : newdemo
jnitrace           /* TID 5465 */
jnitrace           [+] JNIEnv->CallObjectMethodV
jnitrace           |- jmethodID         :0x277   {java.lang.String
cn.rom.nativedemo.MainActivity.demo(int, float, java.lang.String)}
jnitrace           |- va_list           :0x7fe0461bb0
jnitrace           |:     jint          : 1
jnitrace           |:     jfloat        : 2
jnitrace           |:     jstring       : newdemo
jnitrace           |= jstring           :3.0newdemo
jnitrace           /* TID 5465 */
jnitrace           [+] JNIEnv->GetStringUTFChars
jnitrace           |- jboolean*         : 0
jnitrace           |= char*             : 3.0newdemo
jnitrace           /* TID 5465 */
jnitrace           [+] JNIEnv->NewStringUTF
jnitrace           |- char*         : 3.0newdemo
jnitrace leave jni java.lang.String cn.rom.nativedemo.MainActivity.stringFromJNI()
```

12.6 调用栈展示

经过调整后，打桩函数已经非常接近JniTrace的输出效果了，但是还有最后的一点区别是在于JNI函数的调用堆栈，不仅仅需要看到函数的参数和返回值，还需要知道是在哪里触发了该函数。而获取调用堆栈地址，在AOSP源码是有相关支持的，当应用崩溃时，在logcat中能看到详细的堆栈信息。

12.6.1 xUnwind获取调用栈

xUnwind是一个开源工具，该工具将堆栈获取进行了封装，并且有简单的demo演示如何获取堆栈。下载地址：https://github.com/hexhacking/xUnwind.git。接下来将分析该工具是如何实现的获取堆栈，然后再将其内置到AOSP中，在JNI函数调用结束时获取堆栈进行输出。

xUnwind提供了三种调用堆栈回溯方案。

- CFI (Call Frame Info)：由安卓系统库提供。
- EH (Exception handling GCC extension)：由编译器提供。
- FP (Frame Pointer)：只支持 ARM64。

其中CFI主要针对java层调用回溯，FP仅支持ARM64。根据功能需要，选择EH方案进行调用栈回溯。下面看看该工具是如何实现的，首先查看JNI_OnLoad的实现。

```
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved) {
  JNIEnv *env;
  jclass cls;

  (void)reserved;

  if (NULL == vm) return JNI_ERR;
```

```
  if (JNI_OK != (*vm)->GetEnv(vm, (void **)&env, SAMPLE_JNI_VERSION)) return
JNI_ERR;
  if (NULL == env || NULL == *env) return JNI_ERR;
  if (NULL == (cls = (*env)->FindClass(env, SAMPLE_JNI_CLASS_NAME))) return
JNI_ERR;
  // 注册对应的函数
  if (0 != (*env)->RegisterNatives(env, cls, sample_jni_methods,
                                   sizeof(sample_jni_methods) /
sizeof(sample_jni_methods[0])))
    return JNI_ERR;
  // 注册信号处理
  sample_signal_register();

  return SAMPLE_JNI_VERSION;
}
```

然后看看sample_signal_register对信号进行了什么处理

```
static void sample_signal_register(void) {
  struct sigaction act;
  // 清空act
  memset(&act, 0, sizeof(act));

  // 填充所有信号
  sigfillset(&act.sa_mask);
  // 删除SIGSEGV信号
  sigdelset(&act.sa_mask, SIGSEGV);
  // 设置信号动作的对应函数
  act.sa_sigaction = sample_sigabrt_handler;
  act.sa_flags = SA_RESTART | SA_SIGINFO | SA_ONSTACK;
  // 设置指定信号的动作
  sigaction(SIGABRT, &act, NULL);
  // 清空act
  memset(&act, 0, sizeof(act));
  // 填充所有信号
  sigfillset(&act.sa_mask);
  // 设置信号动作的对应函数
  act.sa_sigaction = sample_sigsegv_handler;
  act.sa_flags = SA_RESTART | SA_SIGINFO | SA_ONSTACK;
  // 设置指定信号的动作
  sigaction(SIGSEGV, &act, &g_sigsegv_oldact);
}
```

在这个函数中主要是对信号指定了处理函数，SIGSEGV是一种表示段错误的信号，通常意味着进程访问了无法访问的内存地址。

SIGABRT是一种表示异常终止的信号，通常由调用abort()函数或 C++ 异常处理程序显式引发。abort()函数会向进程发送SIGABRT信号，并导致进程异常终止。

如果进程接收到SIGABRT信号，操作系统将其发送给进程，并中断进程的正常执行流程。此时，进程通常会尝试处理该信号并进行恢复或退出。如果进程没有为SIGABRT信号设置信号处理程序，则默认行为是终止进程。

接着开始跟踪EH的调用栈获取函数sample_test_eh的实现。

```
static void sample_test_eh(JNIEnv *env, jobject thiz, jboolean with_context,
jboolean signal_interrupted) {
  (void)env, (void)thiz;

  sample_test(SAMPLE_SOLUTION_EH, JNI_FALSE, with_context, signal_interrupted);
}
```

继续查看sample_test的实现。

```
static void sample_test(int solution, jboolean remote_unwind, jboolean
with_context,
                        jboolean signal_interrupted) {
  // 将参数存储到全局变量
  g_solution = solution;
  g_remote_unwind = (JNI_TRUE == remote_unwind ? true : false);
  g_with_context = (JNI_TRUE == with_context ? true : false);
  g_signal_interrupted = (JNI_TRUE == signal_interrupted ? true : false);

  // 原子请求，为了保证获取到的g_frames_sz没问题
  __atomic_store_n(&g_frames_sz, 0, __ATOMIC_SEQ_CST);
  // 触发SIGABRT信号
  tgkill(getpid(), gettid(), SIGABRT);

  if ((solution == SAMPLE_SOLUTION_FP || solution == SAMPLE_SOLUTION_EH) &&
      __atomic_load_n(&g_frames_sz, __ATOMIC_SEQ_CST) > 0)
    // 根据堆栈的地址信息，打印堆栈日志
    xunwind_frames_log(g_frames, g_frames_sz, SAMPLE_LOG_TAG, SAMPLE_LOG_PRIORITY,
NULL);
}
```

由于堆栈信息需要通过触发信号后，在信号处理函数中获取，所以这里的参数没办法传过去，这里就将参数放到了全局变量中。然后信号函数执行完毕后，将会填充g_frames调用栈的地址信息，最后使用xunwind_frames_log函数解析调用栈，最后输出详细的调用栈。

接下来查看SIGABRT信号的处理函数sample_sigabrt_handler的实现。

```
static void sample_sigabrt_handler(int signum, siginfo_t *siginfo, void *context)
{
  (void)signum, (void)siginfo;

  if (g_solution == SAMPLE_SOLUTION_FP || g_solution == SAMPLE_SOLUTION_EH) {
```

```
    if (!g_signal_interrupted) {
      if (g_solution == SAMPLE_SOLUTION_FP) {
        // FP local unwind
        ...
      } else if (g_solution == SAMPLE_SOLUTION_EH) {
        // EH local unwind
        size_t frames_sz = xunwind_eh_unwind(g_frames, sizeof(g_frames) /
sizeof(g_frames[0]),
                                             g_with_context ? context : NULL);
        __atomic_store_n(&g_frames_sz, frames_sz, __ATOMIC_SEQ_CST);
      }
    } else {
      // trigger a segfault, we will do "FP local unwind" in the sigsegv's signal
handler
      ...
    }
  } else if (!g_remote_unwind) {
    // CFI local unwind
    ...
  } else {
    // CFI remote unwind
    ...
  }
}
```

EH方案是通过xunwind_eh_unwind函数获取的调用栈信息，继续查看其具体实现。

```
size_t xunwind_eh_unwind(uintptr_t *frames, size_t frames_cap, void *context) {
  return xu_eh_unwind(frames, frames_cap, context);
}

size_t xu_eh_unwind(uintptr_t *frames, size_t frames_cap, void *context) {
  if (NULL == frames || 0 == frames_cap) return 0;

  xu_eh_info_t info;
  info.frames = frames;
  info.frames_cap = frames_cap;
  info.frames_sz = 0;
  info.prev_sp = 0;
  info.uc = (ucontext_t *)context;

  _Unwind_Backtrace(xu_eh_unwind_cb, &info);

  return info.frames_sz;
}
```

_Unwind_Backtrace函数用于获取当前线程的调用堆栈信息。它使用C++异常处理机制中的unwind操作来实现，因此只能在支持C++异常处理的系统上使用。到这里就知道堆栈获取的来源了。但是这里的信息和真实看的还是有一定差距，继续看看xunwind_frames_log是如何对其进行转换展示的。

```
void xunwind_frames_log(uintptr_t *frames, size_t frames_sz, const char *logtag,
android_LogPriority priority,
                        const char *prefix) {
  if (priority < ANDROID_LOG_VERBOSE || ANDROID_LOG_FATAL < priority) return;

  xu_printer_t printer;
  xu_printer_init_log(&printer, logtag, priority);

  xu_formatter_print(frames, frames_sz, prefix, &printer);
}
```

继续跟踪xu_formatter_print的实现。

```
void xu_formatter_print(uintptr_t *frames, size_t frames_sz, const char *prefix,
xu_printer_t *printer) {
  if (NULL == frames || 0 == frames_sz) return;

  if (NULL == prefix) prefix = "";

  void *cache = NULL;
  xdl_info_t info;
  for (size_t i = 0; i < frames_sz; i++) {
    memset(&info, 0, sizeof(xdl_info_t));
    int r = 0;

    if (0 != frames[i]) {
      // 根据调用栈地址获取动态库的信息
      r = xdl_addr((void *)(frames[i]), &info, &cache);

      // 如果查找到的动态库的起始地址大于这条调用栈信息的地址，则从maps中重新找动态库信
息
      char buf[512];
      if (0 == r || (uintptr_t)info.dli_fbase > frames[i])
        r = xu_formatter_maps_addr(frames[i], &info, buf, sizeof(buf));
    }

    // 输出打印
    if (0 == r || (uintptr_t)info.dli_fbase > frames[i])
      xu_printer_append_format(printer, XU_FORMATTER_PREFIX "<unknown>\n", prefix,
i, frames[i]);
    else if (NULL == info.dli_fname || '\0' == info.dli_fname[0])
      xu_printer_append_format(printer, XU_FORMATTER_PREFIX "<anonymous:"
XU_FORMATTER_ADDR ">\n", prefix, i,
                               frames[i] - (uintptr_t)info.dli_fbase,
(uintptr_t)info.dli_fbase);
    else if (NULL == info.dli_sname || '\0' == info.dli_sname[0])
      xu_printer_append_format(printer, XU_FORMATTER_PREFIX "%s\n", prefix, i,
                               frames[i] - (uintptr_t)info.dli_fbase,
info.dli_fname);
    else if (0 == (uintptr_t)info.dli_saddr || (uintptr_t)info.dli_saddr >
```

```
frames[i])
      xu_printer_append_format(printer, XU_FORMATTER_PREFIX "%s (%s)\n", prefix,
i,
                               frames[i] - (uintptr_t)info.dli_fbase,
info.dli_fname, info.dli_sname);
    else
      xu_printer_append_format(printer, XU_FORMATTER_PREFIX "%s (%s+%" PRIuPTR
")\n", prefix, i,
                               frames[i] - (uintptr_t)info.dli_fbase,
info.dli_fname, info.dli_sname,
                               frames[i] - (uintptr_t)info.dli_saddr);
  }
  xdl_addr_clean(&cache);
}
```

继续查看是如何输出的。

```
void xu_printer_append_format(xu_printer_t *self, const char *format, ...) {
  va_list ap;
  va_start(ap, format);

  char tmpbuf[1024];
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wformat-nonliteral"
  vsnprintf(tmpbuf, sizeof(tmpbuf), format, ap);
#pragma clang diagnostic pop

  va_end(ap);

  xu_printer_append_string(self, tmpbuf);
}


void xu_printer_append_string(xu_printer_t *self, const char *str) {
  if (XU_PRINTER_TYPE_LOG == self->type) {
    __android_log_print(self->data.log.priority, self->data.log.logtag, "%s",
str);
  } else if (XU_PRINTER_TYPE_DUMP == self->type) {
    size_t len = strlen(str);
    if (len > 0) {
      xu_util_write(self->data.dump.fd, str, len);
      if ('\n' != str[len - 1]) xu_util_write(self->data.dump.fd, "\n", 1);
    }
  } else if (XU_PRINTER_TYPE_GET == self->type) {
    size_t len = strlen(str);
    if (len > 0) {
      xu_printer_string_append_to_buf(self, str);
      if ('\n' != str[len - 1]) xu_printer_string_append_to_buf(self, "\n");
    }
  }
}
```

到了最后输出的部分可以看到，支持三种方式输出，在调用时，根据不同的需求来选择合适的输出方式。

- `XU_PRINTER_TYPE_LOG` 直接`logcat`输出信息
- `XU_PRINTER_TYPE_DUMP` 将堆栈信息写入到指定描述符
- `XU_PRINTER_TYPE_GET` 返回堆栈信息的字符串。

了解完整个实现原理后，最后看看其输出效果。`EH`方案的调用栈输出如下。

```
I/xunwind_tag: >>> EH: Local Process (pid: 6713), Current Thread (tid: 6713) <<<
I/xunwind_tag: #00 pc 00000000000093a4
/data/app/~~H824r8gCTUibCXd65QZfPw==/io.github.hexhacking.xunwind.sample-
1gWu3tav0VA7-qGsCc7nZQ==/lib/arm64/libxunwind.so (xunwind_eh_unwind+72)
I/xunwind_tag: #01 pc 00000000000014a8
/data/app/~~H824r8gCTUibCXd65QZfPw==/io.github.hexhacking.xunwind.sample-
1gWu3tav0VA7-qGsCc7nZQ==/lib/arm64/libsample.so
I/xunwind_tag: #02 pc 00000000000008b0  [vdso] (__kernel_rt_sigreturn+0)
I/xunwind_tag: #03 pc 000000000009e598
/apex/com.android.runtime/lib64/bionic/libc.so (tgkill+8)
I/xunwind_tag: #04 pc 0000000000001320
/data/app/~~H824r8gCTUibCXd65QZfPw==/io.github.hexhacking.xunwind.sample-
1gWu3tav0VA7-qGsCc7nZQ==/lib/arm64/libsample.so
I/xunwind_tag: #05 pc 0000000000001254
/data/app/~~H824r8gCTUibCXd65QZfPw==/io.github.hexhacking.xunwind.sample-
1gWu3tav0VA7-qGsCc7nZQ==/lib/arm64/libsample.so
I/xunwind_tag: #06 pc 0000000000222248  /apex/com.android.art/lib64/libart.so
(art_quick_generic_jni_trampoline+152)
I/xunwind_tag: #07 pc 0000000000218bec  /apex/com.android.art/lib64/libart.so
(art_quick_invoke_static_stub+572)
I/xunwind_tag: #08 pc 0000000000290300  /apex/com.android.art/lib64/libart.so
(_ZN3art9ArtMethod6InvokeEPNS_6ThreadEPjjPNS_6JValueEPKc+536)
I/xunwind_tag: #09 pc 00000000003f09e4  /apex/com.android.art/lib64/libart.so
(_ZN3art11interpreter34ArtInterpreterToCompiledCodeBridgeEPNS_6ThreadEPNS_9ArtMeth
odEPNS_11ShadowFrameEtPNS_6JValueE+404)
...
...
I/xunwind_tag: #43 pc 0000000000507c68  /apex/com.android.art/lib64/libart.so
(_ZN3art3JNIILb1EE21CallStaticVoidMethodVEP7_JNIEnvP7_jclassP10_jmethodIDSt9__va_l
ist+620)
I/xunwind_tag: #44 pc 00000000000aeac8  /system/lib64/libandroid_runtime.so
(_ZN7_JNIEnv20CallStaticVoidMethodEP7_jclassP10_jmethodIDz+124)
I/xunwind_tag: #45 pc 00000000000ba060  /system/lib64/libandroid_runtime.so
(_ZN7android14AndroidRuntime5startEPKcRKNS_6VectorINS_7String8EEEb+840)
I/xunwind_tag: #46 pc 00000000000025a8  /system/bin/app_process64 (main+1364)
I/xunwind_tag: #47 pc 00000000000498cc
/apex/com.android.runtime/lib64/bionic/libc.so (__libc_init+100)
I/xunwind_tag: >>> finished <<<
```

12.6.2 优化调用栈样例

尽管在该工具中的样例可以直接使用，但是想要将其内置到AOSP中还需要将其进行优化处理，简单写一个模拟注入流程的样例，对样例做出以下调整。

- 去掉非EH获取方案的代码部分
- 优化获取调用栈信息的代码，让其结果仅输出想要关注的目标动态库的调用栈。
- 直接输出日志调整为返回调用栈字符串。

首先将xUnwind中的样例apk解压，找到lib目录查看其依赖的动态库，分别是libxdl.so、libxunwind.so。在源码中能看到libxunwind.so中使用libxdl.so来查询指定地址对应的动态库信息。所以在加载动态库时，需要优先加载libxdl.so。

新建native的项目，将libxdl.so、libxunwind.so复制到新项目的libs目录中，然后修改build.gradle文件如下。

```
android {
    ...
    defaultConfig {
        ...
        ndk {
            abiFilters 'arm64-v8a'
        }
        ...
    }
    ...
    sourceSets {
        main {
            jniLibs.srcDirs = ['libs']
        }
    }
    ...

}
```

然后添加测试代码如下。

```
public class MainActivity extends AppCompatActivity {

    // 注入三个动态库
    static {
        System.loadLibrary("xdl");
        System.loadLibrary("xunwind");
        System.loadLibrary("nativecppdemo");
    }

    private ActivityMainBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
        TextView tv = binding.sampleText;
        tv.setText(stringFromJNI());
    }
    public native String stringFromJNI();
}
```

然后在stringFromJNI函数调时，获取调用栈，参考样例中的代码，并删除非EH方案的代码。创建一个存放头文件kbacktrace.h，内容如下。

```
#ifndef __KBACKTRACE_H__
#define __KBACKTRACE_H__


#include <sys/wait.h>

void sigabrt_handler(int signum, siginfo_t *siginfo, void *context);
void signal_register(void);
const char* kbacktrace(bool with_context,const char* moduleName);

#endif
```

接着创建一个对应的源码文件kbacktrace.cpp，优化后的内容如下。

```
#include "kbacktrace.h"
#include <string>
#include <unistd.h>
#include "xunwind.h"
#include <setjmp.h>

static bool g_with_context = false;

static uintptr_t g_frames[128];
static size_t g_frames_sz = 0;


void sigabrt_handler(int signum, siginfo_t *siginfo, void *context) {
    (void)signum, (void)siginfo;
    // EH local unwind
    size_t frames_sz = xunwind_eh_unwind(g_frames, sizeof(g_frames) /
sizeof(g_frames[0]),
                                         g_with_context ? context : NULL);
    __atomic_store_n(&g_frames_sz, frames_sz, __ATOMIC_SEQ_CST);
}

void signal_register(void) {
    struct sigaction act;
    memset(&act, 0, sizeof(act));
```

```
    sigfillset(&act.sa_mask);
    sigdelset(&act.sa_mask, SIGSEGV);
    act.sa_sigaction = sigabrt_handler;
    act.sa_flags = SA_RESTART | SA_SIGINFO | SA_ONSTACK;
    sigaction(SIGABRT, &act, NULL);
}


static bool isInit=false;

const char* kbacktrace(bool with_context,const char* moduleName) {
    if(!isInit){
        signal_register();
        isInit=true;
    }
    g_with_context = with_context;

    __atomic_store_n(&g_frames_sz, 0, __ATOMIC_SEQ_CST);

    tgkill(getpid(), gettid(), SIGABRT);

    if ( __atomic_load_n(&g_frames_sz, __ATOMIC_SEQ_CST) > 0){
        const char* res= xunwind_frames_get(g_frames,
g_frames_sz,nullptr,moduleName);
        return res;
    }
    return "";
}
```

在原来的样例中有两种信号方式获取调用栈，内置在AOSP中固定使用一种方式即可，所以仅保留SIGABRT信号获取堆栈即可。接着就可以使用该函数获取调用栈了。示例如下。

```
#include <jni.h>
#include <android/log.h>
#include "kbacktrace.h"


#define TAG        "jnitrace"
#define LOG_PRIORITY   ANDROID_LOG_INFO
#define ALOGI(fmt, ...) __android_log_print(LOG_PRIORITY, TAG, fmt, ##__VA_ARGS__)

extern "C" JNIEXPORT jstring JNICALL
Java_cn_rom_nativecppdemo_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject obj /* this */) {
    jstring resObj=env->NewStringUTF("test");
    const char* res= kbacktrace(true,"libnativecppdemo.so");
    ALOGI("------------------------Backtrace------------------------\n%s",res);
    return resObj;
}
```

```
// 在动态库加载时调用，完成本地方法的注册
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *, void *) {
    return JNI_VERSION_1_6;
}
```

修改完样例后，动态库中的xunwind_frames_get函数也需要调整，为其新增参数，将动态库名称传递进去，让其仅返回包含该动态库名称的调用栈。修改如下。

```
char *xunwind_frames_get(uintptr_t *frames, size_t frames_sz, const char
*prefix,const char* moduleName) {
  xu_printer_t printer;
  xu_printer_init_get(&printer);

  xu_formatter_print(frames, frames_sz, prefix, &printer,moduleName);
  return xu_printer_get(&printer);
}
```

继续修改xu_formatter_print函数，添加对动态库的过滤。

```
void xu_formatter_print(uintptr_t *frames, size_t frames_sz, const char *prefix,
xu_printer_t *printer,const char* moduleName) {
  if (NULL == frames || 0 == frames_sz) return;
  if (NULL == prefix) prefix = "";
  void *cache = NULL;
  xdl_info_t info;
  for (size_t i = 0; i < frames_sz; i++) {
    memset(&info, 0, sizeof(xdl_info_t));
    int r = 0;
    if (0 != frames[i]) {
      // find info from linker
      r = xdl_addr((void *)(frames[i]), &info, &cache);
      // 非目标动态库，则直接跳过
      if(!strstr(info.dli_fname,moduleName)){
        continue;
      }
      // find info from maps
      char buf[512];
      if (0 == r || (uintptr_t)info.dli_fbase > frames[i])
        r = xu_formatter_maps_addr(frames[i], &info, buf, sizeof(buf));

    }
    ...
  }
  xdl_addr_clean(&cache);
}
```

完成修改后，编译时发现错误，是因为没有指定依赖的动态库，修改CMakeLists文件如下。

```
if(CMAKE_SYSTEM_PROCESSOR MATCHES "aarch64*")
    # 当前 CPU 架构为 arm64
    target_link_libraries( # Specifies the target library.
            kbacktrace
            ${PROJECT_SOURCE_DIR}/../../../../app/libs/arm64-v8a/libxunwind.so
            # Links the target library to the log library
            # included in the NDK.
            ${log-lib})
endif()

if(CMAKE_SYSTEM_PROCESSOR MATCHES "arm*")
    if(CMAKE_SYSTEM_PROCESSOR MATCHES "v7*")
        # 当前 CPU 架构为 armeabi-v7a
        target_link_libraries( # Specifies the target library.
                kbacktrace
                ${PROJECT_SOURCE_DIR}/../../../../app/libs/armeabi-
v7a/libxunwind.so
                # Links the target library to the log library
                # included in the NDK.
                ${log-lib})
    endif()
endif()
```

最后运行这个新的测试样例，输出日志如下。

```
cn.rom.nativecppdemo                I  -------------------------Backtrace---------
----------------
    #01 pc 000000000001e02c  /data/app/~~-
XZEZm9rvOBJhceZK8LuKg==/cn.rom.nativecppdemo-SoSNN-
J1eghWN4JW7rNYqw==/base.apk!/lib/arm64-v8a/libnativecppdemo.so
    #02 pc 000000000001dd74  /data/app/~~-
XZEZm9rvOBJhceZK8LuKg==/cn.rom.nativecppdemo-SoSNN-
J1eghWN4JW7rNYqw==/base.apk!/lib/arm64-v8a/libnativecppdemo.so
(Java_cn_rom_nativecppdemo_MainActivity_stringFromJNI+180)
```

12.6.3 内置获取调用栈

在测试样例中将主要功能逻辑完成后，最后需要将该功能内置到AOSP中，提供JNI中打桩函数调用。首先是将libxdl.so、libxunwind.so、libkbacktrace.so动态库内置到系统目录中。

在AOSP源码目录frameworks/base/packages/apps/下新建目录，用于存放内置的动态库，然后将样例程序armeabi-v7a和arm64-v8中的动态库放在新目录下。在新目录下添加规则文件Android.mk。内容如下。

```
#--------------------------------
include $(CLEAR_VARS)
```

```
LOCAL_MODULE := libxdl
LOCAL_SRC_FILES_arm := libxdl.so
LOCAL_SRC_FILES_arm64 := libxdl_arm64.so
LOCAL_MODULE_TARGET_ARCHS:= arm arm64
LOCAL_MULTILIB := both
LOCAL_MODULE_SUFFIX := .so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_TAGS := optional
LOCAL_SHARED_LIBRARIES := liblog

include $(BUILD_PREBUILT)

#--------------------------------
include $(CLEAR_VARS)

LOCAL_MODULE := libxunwind
LOCAL_SRC_FILES_arm := libxunwind.so
LOCAL_SRC_FILES_arm64 := libxunwind_arm64.so
LOCAL_MODULE_TARGET_ARCHS:= arm arm64
LOCAL_MULTILIB := both
LOCAL_MODULE_SUFFIX := .so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_TAGS := optional
LOCAL_SHARED_LIBRARIES := liblog libxdl

include $(BUILD_PREBUILT)

#--------------------------------
include $(CLEAR_VARS)

LOCAL_MODULE := libkbacktrace
LOCAL_SRC_FILES_arm := libkbacktrace.so
LOCAL_SRC_FILES_arm64 := libkbacktrace_arm64.so
LOCAL_MODULE_TARGET_ARCHS:= arm arm64
LOCAL_MULTILIB := both
LOCAL_MODULE_SUFFIX := .so
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_TAGS := optional
LOCAL_SHARED_LIBRARIES := liblog libxunwind

include $(BUILD_PREBUILT)
```

然后在文件`build/make/target/product/generic_system.mk`中添加这模块。

```
PRODUCT_PACKAGES += \
    libxdl \
    libxunwind \
    libkbacktrace \
```

最后将这三个动态库添加到公共库的清单文件，修改文件`system/core/rootdir/etc/public.libraries.android.txt`，添加内容如下。

```
libxdl.so
libxunwind.so
libkbacktrace.so
```

编译刷入手机后，即可在系统动态库目录中找到内置进去的动态库。那么应该如何使用内置的动态库来获取堆栈信息呢，在直接修改`JNI`调用时机前，先用一个简单的代码测试能否轻易的获取堆栈。

新建`native c++`项目，修改`stringFromJNI`函数，添加打印调用栈信息的代码如下。

```
#include <jni.h>
#include <string>
#include <android/log.h>
#include <dlfcn.h>

#define TAG        "jnitrace"
#define LOG_PRIORITY   ANDROID_LOG_INFO
#define ALOGI(fmt, ...) __android_log_print(LOG_PRIORITY, TAG, fmt, ##__VA_ARGS__)

typedef const char* (*kbacktraceFunc)(bool with_context,const char* moduleName);

void showBacktrace(const char* moduleName){
    kbacktraceFunc kbacktrace = (kbacktraceFunc)dlsym(RTLD_DEFAULT,
"_Z10kbacktracebPKc");
    if(kbacktrace==NULL){
        const char* error = dlerror();
        ALOGI("not found method.%s",error);
        return;
    }
    const char* res=kbacktrace(true,moduleName);
    ALOGI("%s\n",res);
}

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_backtracedemo_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    std::string hello = "Hello from C++";

    showBacktrace("");

    return env->NewStringUTF(hello.c_str());
}
```

运行该项目后，成功输出调用栈信息，到这里已经能随时获取调用栈了。在初始化配置的时候获取到该函数指针，并存储起来。修改`initConfig`代码如下。

```
static void
DexFile_initConfig(JNIEnv* env, jobject ,jobject item) {

    Runtime* runtime=Runtime::Current();
    jclass jcInfo = env->FindClass("java/krom/PackageItem");
    jfieldID jPackageName = env->GetFieldID(jcInfo, "packageName",
"Ljava/lang/String;");
    jfieldID jJniModuleName = env->GetFieldID(jcInfo, "jniModuleName",
"Ljava/lang/String;");
    jfieldID jJniFuncName = env->GetFieldID(jcInfo, "jniFuncName",
"Ljava/lang/String;");
    jfieldID jIsRegisterNativePrint = env->GetFieldID(jcInfo,
"isRegisterNativePrint", "Z");
    jfieldID jIsJNIMethodPrint = env->GetFieldID(jcInfo, "isJNIMethodPrint", "Z");

    PackageItem citem;

    jstring jstrPackageName = (jstring)env->GetObjectField(item, jPackageName);
    const char* pPackageName = (char*)env->GetStringUTFChars(jstrPackageName, 0);
    strcpy(citem.packageName, pPackageName);

    jstring jstrJniModuleName = (jstring)env->GetObjectField(item,
jJniModuleName);
    const char* pJniModuleName = (char*)env->GetStringUTFChars(jstrJniModuleName,
0);
    strcpy(citem.jniModuleName, pJniModuleName);

    jstring jstrJniFuncName = (jstring)env->GetObjectField(item, jJniFuncName);
    const char* pJniFuncName = (char*)env->GetStringUTFChars(jstrJniFuncName, 0);
    strcpy(citem.jniFuncName, pJniFuncName);

    citem.isRegisterNativePrint = env->GetBooleanField(item,
jIsRegisterNativePrint);
    citem.isJNIMethodPrint = env->GetBooleanField(item, jIsJNIMethodPrint);

    if(citem.isJNIMethodPrint){
        void* handle_xdl=NULL;
        void* handle_xunwind=NULL;
        void* handle_kbacktrace=NULL;
        #if defined(__aarch64__)
                // 当前 CPU 架构为 arm64
            handle_xdl= dlopen("/system/lib64/libxdl.so",RTLD_NOW);
            handle_xunwind= dlopen("/system/lib64/libxunwind.so",RTLD_NOW);
            handle_kbacktrace= dlopen("/system/lib64/libkbacktrace.so",RTLD_NOW);
        #else
            // 当前 CPU 架构为 armeabi-v7a 或更早版本
            handle_xdl= dlopen("/system/lib/libxdl.so",RTLD_NOW);
            handle_xunwind= dlopen("/system/lib/libxunwind.so",RTLD_NOW);
            handle_kbacktrace= dlopen("/system/lib/libkbacktrace.so",RTLD_NOW);
        #endif
        if(handle_kbacktrace!=nullptr){
            citem.kbacktrace= dlsym(handle_kbacktrace, "_Z10kbacktracebPKc");
```

```
            if(citem.kbacktrace==nullptr){
                ALOGD("jnitrace kbacktrace is null.err:%s",dlerror());
            }else{
                ALOGD("jnitrace kbacktrace:%p.",citem.kbacktrace);
            }
        }else{
            ALOGD("jnitrace handle_kbacktrace is null.err:%s",dlerror());
        }
    }
    runtime->SetConfigItem(citem);
}
```

然后就可以在JNI打桩函数时输出堆栈信息了，下面将代码进行简单封装和调用。

```
typedef const char* (*kbacktraceFunc)(bool,const char*);

const char* getBacktrace(const char* moduleName){
    Runtime* runtime=Runtime::Current();
    if(runtime->GetConfigItem().kbacktrace== nullptr){
        ALOGD("jnitrace kbacktrace is null");
        return nullptr;
    }
    kbacktraceFunc kbacktrace=(kbacktraceFunc)runtime->GetConfigItem().kbacktrace;
    return kbacktrace(true,moduleName);
}

void ShowVarArgs(const ScopedObjectAccessAlreadyRunnable& soa,const char*
funcname,jclass java_class, const char* name, const char* sig,jmethodID methodID){
    if(!HasShow()){
        return;
    }
    ObjPtr<mirror::Class> c = soa.Decode<mirror::Class>(java_class);
    std::string temp;
    const char* className= c->GetDescriptor(&temp);
    ArtMethod* method = jni::DecodeArtMethod(methodID);
    pid_t pid = getpid();
    ALOGD("%s           /* TID %d */","jnitrace",pid);
    ALOGD("%s           [+] JNIEnv->%s","jnitrace",funcname);
    ALOGD("%s           |- jclass           :%s","jnitrace",className);
    ALOGD("%s           |- char*            :%p","jnitrace",name);
    ALOGD("%s           |:     %s","jnitrace",name);
    ALOGD("%s           |- char*            :%p","jnitrace",sig);
    ALOGD("%s           |:     %s","jnitrace",sig);
    ALOGD("%s           |= jmethodID        :0x%x   {%s}","jnitrace",method-
>GetMethodIndex(),method->PrettyMethod().c_str());
    Runtime* runtime=Runtime::Current();
    const char* backtrace= getBacktrace(runtime->GetConfigItem().jniModuleName);
    ALOGD("-------------------------Backtrace-------------------------
\n%s\n",backtrace);
}
```

到这里JniTrace的AOSP版本就完成了，其他的函数调用参考前面的做法即可，最后优化后的日志输入如下。

```
jnitrace enter jni java.lang.String cn.rom.nativedemo.MainActivity.stringFromJNI()
0x7a0bc06010
jnitrace           /* TID 5641 */
jnitrace           [+] JNIEnv->GetMethodID
jnitrace           |- jclass           :Lcn/rom/nativedemo/MainActivity;
jnitrace           |- char*            :0x781eb6cf1b
jnitrace           |:     demo
jnitrace           |- char*            :0x781eb6ceef
jnitrace           |:     (IFLjava/lang/String;)Ljava/lang/String;
jnitrace           |= jmethodID        :0x277   {java.lang.String
cn.rom.nativedemo.MainActivity.demo(int, float, java.lang.String)}
-------------------------Backtrace-------------------------
#05 pc 000000000000e9dc  /data/app/~~MjwExmAtQBa8X1Xp3ifz_g==/cn.rom.nativedemo-
YbmCkQ7SdhNqbL7iXfOeug==/lib/arm64/libnativedemo.so
(_ZN7_JNIEnv11GetMethodIDEP7_jclassPKcS3_+60)
#06 pc 000000000000e888  /data/app/~~MjwExmAtQBa8X1Xp3ifz_g==/cn.rom.nativedemo-
YbmCkQ7SdhNqbL7iXfOeug==/lib/arm64/libnativedemo.so
(Java_cn_rom_nativedemo_MainActivity_stringFromJNI+80)
jnitrace           /* TID 5641 */
jnitrace           [+] JNIEnv->NewStringUTF
jnitrace           |- char*        : newdemo
-------------------------Backtrace-------------------------
......
```

12.7 本章小结

本章完整介绍了系统代码插桩方式来实现Jnitrace功能。该功能用于动态跟踪分析程序执行时的JNI调用参数与栈详情信息。对比python版本的Jnitrace工具，使用Frida代码编写的跟踪工具拥有着便捷与扩展性强等特点，但缺点是这种方式需要处理大量调用，对程序的并发处理能力要求极高，在一些JNI调用频繁的程序上执行分析，可能会出现并发崩溃的情况，使用系统代码插桩的优点是，在高并发调用下，跟踪代码仍然能正常的输出分析内容，具有较高的稳定性。

系统定制涉及到技术原理与代码解读的部分着实枯燥，很感谢读者朋友们能坚持看完本书。本书所有的内容到这里就要讲完了，然后，值得被扩展与定制的功能非常之多，限于篇幅限制，作者水平十分有限，很多功能没有在本书中涉及与讨论。有兴趣的朋友，可以关注微信公众号[软件安全与逆向分析]，微信号：feicong_sec。作者会在上面讨论一些系统定制修改与安全技术相关的话题。