JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code

Fengguo Wei
University of South Florida
fwei@mail.usf.edu

Xingwei Lin
University of Electronic Science and Technology of China
xwlin.roy@gmail.com

Xinming Ou
University of South Florida
xou@usf.edu

Ting Chen
University of Electronic Science and Technology of China
brokendragon@uestc.edu.cn

Xiaosong Zhang
University of Electronic Science and Technology of China
johnsonzxs@uestc.edu.cn

ABSTRACT

Android allows application developers to use native language (C/C++) to implement a part or the complete program. Recent research and our own statistics show that native payloads are commonly used in both benign and malicious apps. Current state-of-the-art Android static analysis tools, such as Amandroid, FlowDroid, DroidSafe, IccTA, and CHEX avoid handling native method invocation and apply conservative models for their data-flow behavior. None of those tools have the capability to capture the inter-language dataflow. We propose a new approach to conduct inter-language dataflow analysis for security vetting of Android apps and build an analysis framework, called *JN-SAF* to compute flow and context-sensitive inter-language points-to information in an efficient way. We show that: 1) Precise and efficient inter-language dataflow analysis is completely feasible with support of a summary-based bottom-up dataflow analysis (SBDA) algorithm, 2) A comprehensive model of Java Native Interface (JNI) and Native Development Kit (NDK) for binary analysis is essential as none of the existing binary analysis frameworks is able to handle Android binaries, 3) *JN-SAF* is capable of capturing inter-language security issues in real-world Android apps as demonstrated by our evaluation result.

CCS CONCEPTS

• **Security and privacy → Software and application security**;

KEYWORDS

Static Analysis; Mobile Security

ACM Reference Format:
Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. *JN-SAF*: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243835

1 INTRODUCTION

Android continuously dominates the smartphone market with about 76% share according to Statcounter [6]. Recent study [9, 24, 29, 40, 42, 45–47] have shown that native code is a continuous threat which might stealthily leak sensitive information or utilize Android malware to evade AV detection. Our statistics on 100,000 Google Play applications also show that there is substantial usage (39.7%) of native code in benign apps, and the majority (> 80%) of those native method invocations involve data communication. This raises a major concern about how we can make sure the native code are not malicious.

There is a long line of works [10, 12, 13, 15, 17, 21, 23, 25, 28, 30, 34, 38, 41, 43, 44] that design or utilize static analysis tools to detect security issues in Android applications. Only a couple of them [10, 34] address security issues related to native code. However, none of them can track precise inter-language dataflow. The existing state-of-the-art Android static analysis frameworks, such as Amandroid [43, 44], FlowDroid [12], DroidSafe [21], IccTA [23] and CHEX [25], do not currently provide the capability to perform inter-language dataflow analysis or handle native components. When encountering a native method invocation, all of the existing dataflow analysis frameworks either apply a conservative model which assumes any data flow could happen, or ignore the side-effects produced by the native call, which will cause major imprecision in the analysis result. There is an urgent need to design a comprehensive dataflow analysis framework that can track dataflows across language boundaries and understand dataflow behaviors in both the "Java world" and the "native world."

Android Inter-language Analysis Challenges:

(1) Dataflow analysis for Dalvik-bytecode and for native binary have totally different algorithms and representations of object points-to information. How to have a unified representation to integrate the dataflow analysis results from both worlds is a significant challenge.

CCS '18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243835

(2) A practical dataflow analysis framework needs to find the good balance between precision and efficiency. Precise dataflow analysis is computationally heavy for both Java world and native world. Both worlds can influence dataflow facts with each other, leading to many interleaving in the dataflow analysis. How to limit the analysis context switch frequency and still keep good precision is a major challenge.
(3) Android provides a Native Development Kit (NDK) [1] which allows the developer to design app in native language (C/C++). NDK enables native Activity component, provides a set of native libraries to assist native code to access Android-specific features and uses Java Native Interface (JNI) as the communication bridge. Precisely tracking dataflows in native Activity component and modeling NDK libraries and JNI data structures are significant challenges.

The main contributions of this work are as follows.

(1) We adopt a summary-based bottom-up dataflow analysis (SBDA) approach to compute flow and context-sensitive inter-language dataflow information in an efficient way. The summary-based nature of SBDA enables us to design unified heap manipulation summary representation for both Java world and native world dataflow analysis. The bottom-up approach allows us to only visit each method exactly once to compute summary Δ and reuse Δ when a caller method invokes it.
(2) We comprehensively model control and data flow behavior for the Native component, NDK libraries, and JNI data structures to enable existing binary analysis tool, such as *angr* [36] to understand Android-specific data flows.
(3) we design *JN-SAF*— a precise and efficient NDK/JNI-aware inter-language analysis framework for Android apps. For the best of our knowledge, *JN-SAF* is the first Android static analysis framework that performs inter-language dataflow tracking. Our evaluation result shows that *JN-SAF* is capable of doing real-world app vetting, and is able to find interesting cross-language security issues. We plan to release the executable and source code of *JN-SAF* upon publication of the paper.

The rest of the paper is organized as follows. Section 2 presents the background information with a motivating example. Section 3 discusses challenges and our solutions, whereas Section 4 describes in detail *JN-SAF* architecture. We discuss evaluation results of our approach in Section 5, limitation of *JN-SAF* in Section 6, related research in Section 7, and conclude in Section 8.

2 BACKGROUND AND EXAMPLE

We provide necessary background information to understand how Android native world works, and how the inter-language communication is handled. We also provide a motivating example to discuss the challenges to track static data-flow for Android application with the native world.

2.1 Native Code Usage Modes in Android

Android developers can introduce native code in two ways. In the first mode, the developer can write certain functions in native language (C/C++) and include the compiled binary as a shared object as part of the application. Those functions are then called by an Android component that is still written in Java. In the other mode, a complete component can be written in native code and the Android runtime directly calls the life-cycle methods of the component in the native code. Currently Android only allows the second mode for the Activity component (called *native Activity*). Whereas all four Android component types could involve native code through the first mode.

2.2 Native Development Kit (NDK)

The Native Development Kit (NDK) [1] is a set of tools that allow designing part of the Android application using native languages. NDK provides platform libraries to help manage native Activity components and access physical device components. It uses Java Native Interface (JNI) [3] as the interface via which the Java and C++ components talk to one another. It is mainly used in cases such as improving performance, reusing existing third-party C or C++ libraries, and so on.

NDK together with JNI defines how Java code sends data to native functions and receives return values, and how native code creates/modifies/inspects Java objects and invokes Java methods. Since Android 2.3, NDK provides a helper library which allows the developer to design a whole Android Activity using native code. To precisely handle inter-language dataflow in Android, *JN-SAF* must have a comprehensive model for JNI related data structures and native Activity as explained in Section 3.

2.3 A Motivating Example

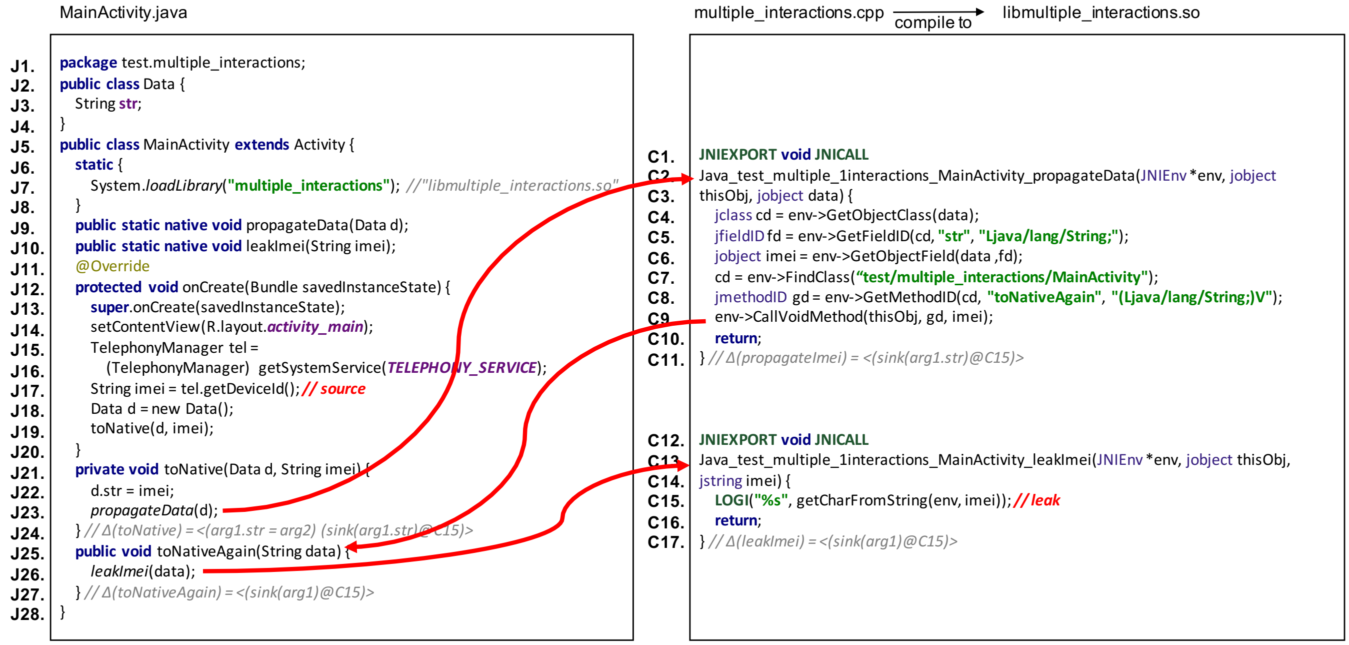
A malicous app developer can make use of NDK and develop part of the app's functionality in the Native world. Figure 1 illustrates an example app (named "IMEI-leaking"). It consists of two worlds, 1) Java world: An *Activity* component which loads a native library "*multiple_interactions*" and imports two native methods *propagateData()* and *leakImei()*; 2) Native world: Export two native functions which leverage NDK libraries to read Java objects and invoke Java methods.

Resolving native method call is different from resolving normal Java calls. In order to find the native method callee, one has to know which native library is loaded by the instance. From the native library we need to know what native functions are exported, then we can find the corresponding function as the native method callee.

To track the data and control flow across language boundaries, a static analyzer must understand the semantics of both languages, as well as understanding the inter-language communication interface and APIs.

As an example, the following sequence of events (as labeled in Figure 1) can happen in reality:

(1) *MainActivity* invokes native method *propagateData()* and passes an object *d* which carries a sensitive data.
(2) *Java_test_multiple_1interactions_MainActivity_propagateData()* receives the Java object *data*, gets *str* field (sensitive data) and then invokes Java method *toNativeAgain().*
(3) *toNativeAgain()* at *MainActivity* receives *data* and passes it to native method *leakImei().*
(4) *Java_test_multiple_1interactions_MainActivity_leakImei()* will receive the *imei* and leaks to the log.



```
J1.  package test.multiple_interactions;
J2.  public class Data {
J3.     String str;
J4.  }
J5.  public class MainActivity extends Activity {
J6.     static {
J7.        System.loadLibrary("multiple_interactions"); //"libmultiple_interactions.so"
J8.     }
J9.     public static native void propagateData(Data d);
J10.    public static native void leakImei(String imei);
J11.    @Override
J12.    protected void onCreate(Bundle savedInstanceState) {
J13.       super.onCreate(savedInstanceState);
J14.       setContentView(R.layout.activity_main);
J15.       TelephonyManager tel =
J16.          (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
J17.       String imei = tel.getDeviceId(); // source
J18.       Data d = new Data();
J19.       toNative(d, imei);
J20.    }
J21.    private void toNative(Data d, String imei) {
J22.       d.str = imei;
J23.       propagateData(d);
J24.    } // Δ(toNative) = <(arg1.str = arg2) (sink(arg1.str)@C15)>
J25.    public void toNativeAgain(String data) {
J26.       leakImei(data);
J27.    } // Δ(toNativeAgain) = <(sink(arg1)@C15)>
J28. }
```

```
C1.  JNIEXPORT void JNICALL
C2.  Java_test_multiple_1interactions_MainActivity_propagateData(JNIEnv *env, jobject
C3.  thisObj, jobject data) {
C4.     jclass cd = env->GetObjectClass(data);
C5.     jfieldID fd = env->GetFieldID(cd, "str", "Ljava/lang/String;");
C6.     jobject imei = env->GetObjectField(data ,fd);
C7.     cd = env->FindClass("test/multiple_interactions/MainActivity");
C8.     jmethodID gd = env->GetMethodID(cd, "toNativeAgain", "(Ljava/lang/String;)V");
C9.     env->CallVoidMethod(thisObj, gd, imei);
C10.    return;
C11. } // Δ(propagateImei) = <(sink(arg1.str)@C15)>

C12. JNIEXPORT void JNICALL
C13. Java_test_multiple_1interactions_MainActivity_leakImei(JNIEnv *env, jobject thisObj,
C14. jstring imei) {
C15.    LOGI("%s", getCharFromString(env, imei)); // leak
C16.    return;
C17. } // Δ(leakImei) = <(sink(arg1)@C15)>
```

Figure 1: The IMEI-leaking App: The arrowed lines among the app components highlight some of the inter-language-communication.

To track the data and control flow across language boundary, a static analyzer needs to understand the bridge interface – JNI. For example, when *MainActivity* invokes *propagateData()* at J23, the static analyzer needs to know: 1) the *libmultiple_interactions.so* has been loaded at J7; 2) the corresponding native function name is *Java_test_multiple_1interactions_MainActivity_propagateData* via applying naming convention. Furthermore, when native function *Java_test_multiple_1interactions_MainActivity_propagateData()* invokes *MainActivity.toNativeAgain()* at C9, the static analyzer needs to model and analyze the reflection style JNI functions: 1) C4-C6 read *str* field from *data* and assign to *imei*; 2) C7 and C8 construct a method identifier to Java method *MainActivity.toNativeAgain()*; 3) C9 invokes *MainActivity.toNativeAgain()* with parameter *imei*.

After resolving the native method call at J23 and J26 and the native reflection call at C9 we can track dataflow between the two worlds. Then at C15 we will be able to say that the variable *imei* to be written to the log is sensitive.

3 CORE CHALLENGES AND OUR SOLUTIONS

For both Java world and native world, there are already mature static analysis tools for either one of them [12, 16, 25, 36, 37, 43, 44]. Instead of building a new analyzer from scratch, it is advantageous to leverage these existing static analyzers to build an inter-language dataflow analysis framework for Android. However, there are several challenges in such an effort.

3.1 Challenge 1: Inter-language Analysis Challenge

(1) **Difference in intermediate data representation:** Java data flow analysis typically tracks points-to facts, whereas binary dataflow analysis typically uses *symbolic execution*. Thus the two analysis engines use different data representations in the analysis process, making it hard to integrate. How to design a unified dataflow representation for both analyses is a challenge.
(2) **Efficiency:** Both Java dataflow analysis and binary symbolic execution are computationally expensive. The traditional dataflow analysis requires propagating dataflow facts continuously over the complete program's control flow graph until a fixed point is reached. For inter-language analysis, this means the analysis process need to constantly switch between the Java and binary analysis context. This further exacerbates analysis time.

To address above challenges, we adopt the *Summary-based Bottom-up Dataflow Analysis* (SBDA) algorithm introduced in [19]. The benefit of this method is that we only need to visit each method exactly once to generate a unified heap manipulation summary for both Java and native procedures, while still preserving a flow and context-sensitive dataflow analysis result.

Figure 2 illustrates the workflow of *SBDA*. It takes the environment method as *EP* and generates a call graph *G* from it. From *G* we apply a topological sort algorithm with the reverse order to get a list of method *MList*, which guarantees the callee method always comes before the caller method. If there is a cycle in the call graph, the algorithm will break the cycle arbitrarily to make sure the topological sort will always hold. For each method M_i in *MList*, we apply a heap manipulation summary generation algorithm to

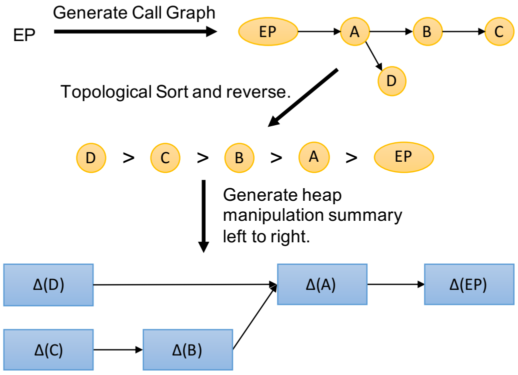

EP
Generate Call Graph
EP
A
B
C
D
Topological Sort and reverse.
D > C > B > A > EP
Generate heap manipulation summary left to right.
Δ(D)
Δ(A)
Δ(EP)
Δ(C)
Δ(B)

Figure 2: SBDA workflow.

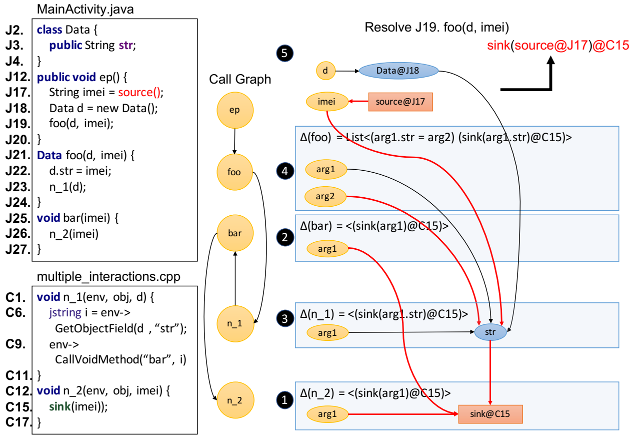
get summary Δ_i. The callee method's summary will propagate to its caller methods until the *EP* is reached.

Heap Manipulation Summary. A summary Δ for a method m is presented by following language:

⟨Δ⟩ ::= '<' ⟨*Rule*⟩* '>'
⟨*Rule*⟩ ::= '(' [⟨*AssignRule*⟩ | ⟨*ActionRule*⟩] ')'
⟨*AssignRule*⟩ ::= ⟨*HeapLoc*⟩ ['=' | '+=' | '-'] ⟨*RHS*⟩
⟨*ActionRule*⟩ ::= ⟨*Action*⟩ '(' ⟨*RHS*⟩ ')' '@' ⟨*Loc*⟩
⟨*RHS*⟩ ::= ⟨*HeapLoc*⟩ | ⟨*Instance*⟩
⟨*Action*⟩ ::= '~' | 'source' | 'sink'
⟨*HeapLoc*⟩ ::= ⟨*HeapBase*⟩ ⟨*Index*⟩
⟨*HeapBase*⟩ ::= 'arg' Digits | 'ret' | ID
⟨*Index*⟩ ::= '.' ID | '[]'
⟨*Instance*⟩ ::= ID '@' ⟨*Loc*⟩
⟨*Loc*⟩ ::= ID

Δ consists of a list of *Rules*. There are two types of *Rule*: *AssignRule* and *ActionRule*. *AssignRule* defines what kind of data propagation happened for the given *HeapLoc* at which *Loc*, whereas *ActionRule* defines what action should take for the *HeapLoc*. *AssignRule* allows three operations: 1) '=' strong update for a *HeapLoc*; 2) '+=' weak update for a *HeapLoc*; 3) '-' kill facts from *RHS*. *ActionRule* has three *Actions*: 1) '~' clear all heap for *RHS*; 2) '*source*' mark an *RHS* as sensitive data; 3). '*sink*' mark an *RHS* as a leaky point. *RHS* consists of *HeapLoc* or *Instance* which represents right-hand-side values. *HeapLoc* is used to represent the heap location which consists of *HeapBase* and *Index*. There are three types of *HeapBase* a callee method could use to create heap manipulating side-effect: the heap of arguments, return value and global variables. Depending on the object type of *HeapBase*, field access or array access can be used to present the *Index*. *Instance* represents the object instance created at particular *Loc*. For example, the *toNative()* method in Figure 1 generates a summary $\Delta(toNative) = \langle(arg1.str = arg2)(sink(arg1.str)@\texttt{C15})\rangle$ where the $arg1.str$ is a *HeapLoc* which means the *str* field of the first argument, and $sink(arg1.str)@\texttt{C15}$ indicates the *str* field of first argument will be leaked at location C15.

Let's take Figure 3 as an example to walkthrough the heap manipulation summary generation process and how we leverage the summary Δ to resolve the dataflow problem for the motivating example. Start from method *ep()* we build a *Call Graph*, and topological sort it in reverse order. We start generating the summary Δ from the leaf function *n_2()*. Native function *n_2()* leaks the first argument thus we generate a summary $\Delta(n_2) = \langle(sink(arg1)@\texttt{C15})\rangle$ and propagate it to Java method *bar()*. *bar()* pass first argument to *n_2()* and the $\Delta(n_2)$ is applied. Therefore, we get summary $\Delta(bar) = \langle(sink(arg1)@\texttt{C15})\rangle$ and propagate it to native function *n_1()*. *n_1()* read *str* field from first argument *d* and invokes method *bar()*. Therefore, $\Delta(bar)$ is applied and we get summary $\Delta(n_1) = \langle(sink(arg1.str)@\texttt{C15})\rangle$. *foo()* puts second argument *imei* into *str* field of first argument *d*, and invokes native function *n_1()*. We apply $\Delta(n_1)$ and then get $\Delta(foo) = \langle(arg1.str = arg2)(sink(arg1.str)@\texttt{C15})\rangle$. Java method *ep()* assigns a sensitive data to variable *imei* at J17 and creates a *Data* instance to *d* at J18. J19 of Java method *ep()* invokes method *foo()*. $\Delta(foo)$ tells us *str* field of variable *d* gets data in variable *imei* which is sensitive, and this *str* field of variable *d* will flow to a leak point at C15. Therefore, we capture the data leakage problem.


MainActivity.java

```
J2.  class Data {
J3.    public String str;
J4.  }
J12. public void ep() {
J17.   String imei = source();
J18.   Data d = new Data();
J19.   foo(d, imei);
J20. }
J21. Data foo(d, imei) {
J22.   d.str = imei;
J23.   n_1(d);
J24. }
J25. void bar(imei) {
J26.   n_2(imei)
J27. }
```

multiple_interactions.cpp

```
C1.  void n_1(env, obj, d) {
C6.    jstring i = env->
         GetObjectField(d , “str”);
C9.    env->
         CallVoidMethod(“bar”, i)
C11. }
C12. void n_2(env, obj, imei) {
C15.   sink(imei));
C17. }
```

Call Graph
ep
foo
bar
n_1
n_2
Resolve J19. foo(d, imei)
sink(source@J17)@C15
5
d
Data@J18
imei
source@J17
4
Δ(foo) = List<(arg1.str = arg2) (sink(arg1.str)@C15)>
arg1
arg2
2
Δ(bar) = <(sink(arg1)@C15)>
arg1
3
Δ(n_1) = <(sink(arg1.str)@C15)>
arg1
str
1
Δ(n_2) = <(sink(arg1)@C15)>
arg1
sink@C15

Figure 3: Heap Manipulation Summary of App "IMEI-leaking": An excerpt.[1]

3.2 Challenge 2: Resolving Native Method Calls

JNI allows two ways to resolve a native method call to a native function:

(1) **Default:** Follow the naming convention in JNI specification [8] to generate corresponding native function name. For example, as Figure 1 illustrated, the corresponding native function name for native method *MainActivity.propagateData()* is *Java_test_multiple_1interactions_MainActivity_propagateData*.
(2) **Dynamic register:** *JNI* allows developer to dynamically register native method signature to native function mapping.

To assist dataflow analysis engine to find native method callee, we propose a *Native Method Mapping* data structure. *Native Method Mapping* is a map where the key is the native method signature and the value is the corresponding native function name and the containing *so* file.

[1] We shortened the method/function names for better presentation. First two arguments of native functions are not counted in the summary as *env* is not presented in Java method and *obj* is "this".

Algorithm 1 Resolve loaded library for class *C*

```
Input: all classes' IR of $A$.
Output: Loaded library for class $C$, $libNameSet$
procedure RESOLVELIBNAMESET($A$, $C$)
    $libNameSet \leftarrow$ empty set
    $loadSigs \leftarrow$ Set("System.load()", "System.loadLibrary()", "Runtime.load()", "Runtime.loadLibrary()")
    for all $class \in A$.getAllReachableClasses($C$) do
        $clinit \leftarrow class$.getStaticInitializer();
        for all $invoke \in clinit$.getInvokeStatements() do
            if $invoke.signature \in loadSigs$ then
                $libNameSet \leftarrow libNameSet :: invoke$.getValueForParameter(1)
    return $libNameSet$;
```

Algorithm 2 Generate Native Method Mapping of APK *A*

```
Input: All classes' IR of $A$.
Output: $A$'s native method to so file map, $n\_map$
procedure GENNATIVEMETHODMAP($A$)
    $n\_map \leftarrow$ empty map
    for all $class \in A$.getClasses() do
        $nativeMethods \leftarrow class$.getNativeMethods();
        if $nativeMethods \neq$ empty then
            $libnames \leftarrow$resolveLibNameSet(A, $class$)        ▷ Invoke Algorithm 1
            for all $name \in libnames$ do
                $nLib \leftarrow$ A.loadNativeLibrary(name);
                for all $method \in nativeMethods$ do
                    $funcName \leftarrow method$.toJNIName();
                    if $funcName \in nLib$.getFunctionNames() then
                        $n\_map(method) \leftarrow (funcName, name)$;
                    else
                        $dynamicMap \leftarrow nLib$ .getDynamicRegisterFunctions();
                        if $method \in dynamicMap$ then
                            $n\_map(method) \leftarrow (dynamicMap(method), name)$;
    return $n\_map$;
```

Algorithm 2 shows the pseudocode for generating *Native Method Mapping n_map* of a given APK *A*. We first visit each *class* in *A*. If *class* defined native methods, we then follow Algorithm 1 to find the possible native function containing *so* files. For each native *method* in the *class*, we generate its native function name *funcName* following the naming convention. We then load each so file, *nLib*, and see if the *funcName* exists in *nLib*. If yes, we add it into the *n_map*. If not, we continue checking the dynamically registered function list for *nLib* and check if the *method* is dynamically registered. If yes, we add it into the *n_map*. However, to obtain the dynamically registered functions for *nLib* is a non-trivial work. We took following approach to compute.

Dynamic Function Register Resolution. As illustrated in Figure 4, *JNI* allows register dynamic function mapping by implementing the *JNI_OnLoad()* method. The *JNINativeMethod* structure contains the mapping information between the native method name, signature and the corresponding native function pointer. `C5-C8` defines an *JNINativeMethod* array *gMethods* to indicate the mapping for native methods *foo()* and *bar()*, then `C16` invokes *RegisterNatives()* with *gMethods* to register.

Dynamic function register resolution procedures:

(1) Dynamic register begins at *JNI_OnLoad()* method, whose first argument is *JavaVM *vm*. Therefore, we first construct a fake pointer to the *JNIInvokeInterface* structure, which has been modeled, and attach the initialized pointer to the first argument (register *R0*) of *JNI_OnLoad()*.

(2) We do the *symbolic execution* from the *JNI_OnLoad()*. In this situation, we need to get the *JNINativeInterface* to make JNI calls. As Figure 4 illustrated, *JNI_OnLoad()* method will first declare an uninitialized *JNIEnv *env* variable. Then it will call *GetEnv()* function from *vm* to initialize the *env* variable. We create a *SimProcedure(GetEnv)* to simulate this behavior. We construct a fake *JNINativeInterface* pointer outside the *GetEnv()* function and then attach to it. Then the *env* variable constructed by *JNI_OnLoad()* can be assigned and continue to propagate.

jni.h

```
C1.  typedef struct {
C2.      const char* name;
C3.      const char* signature;
C4.      void*       fnPtr;
C5.  } JNINativeMethod;
```

main.cpp

```
C5.  static JNINativeMethod gMethods[] = {
C6.      {"foo",   "(Ljava/lang/String;)V",  (void *) native_foo},
C7.      {"bar", "(ILjava/lang/String;)V", (void *) native_bar},
C8.  };
C9.  JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved) {
C10.   JNIEnv *env = NULL;
C11.
C12.   if (vm->GetEnv((void **) &env, JNI_VERSION_1_4) != JNI_OK) {
C13.     return -1;
C14.   }
C15.   ...
C16.   if (env->RegisterNatives(clazz, gMethods, numMethods) < 0) {
C17.     return -1;
C18.   }
C19.   ...
C20. }
```

Figure 4: JNINativeMethod Structure

(3) We hook *SimProcedure(RegisterNatives)* to *JNINativeInterface*'s function pointers table. When the symbolic execution engine executes *SimProcedure(RegisterNatives)*, we can get the memory address of the *gMethods* array. Because each element is accessible at a fixed offset through the *JNINativeMethod* structure. We can resolve each element value of the *gMethods* based on the address and the structure of *JNINativeMethod*.

(4) Each *JNINativeMethod* contains three elements, native method name, native method signature, native function address. We match the native method information from *SBDA* and find its corresponding native function address. Then we can begin *Native Function Summary Builder* from that address.

3.3 Challenge 3: Leveraging Existing Binary Analyzer for Dataflow Analysis

There are a number of existing binary analysis tools [16, 36, 37]. We use *angr* [36] for our work. *angr* is a general binary analysis platform which uses *symbolic execution* technique to recover precise CFG (called *CFGAccurate*) in binary and allows user to perform annotation-based analysis. However, *angr* is not aware of NDK library, JNI function and Java object/method. Therefore, it cannot be directly used to track dataflow in Android binaries.

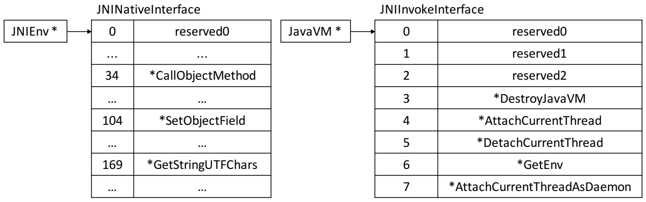
To do NDK/JNI-aware dataflow analysis for Android binary, we leverages *angr*'s *symbolic execution* engine and implements an *Annotation-based Dataflow Analyzer*.

Annotation-based Dataflow Analysis (ADA) leverages *angr*'s *Annotation* and *SimProcedure* features, and is NDK/JNI-aware. *Annotation* is a customizable interface which *angr* uses to allow users to define what kind of data needs to be carried in the state of *symbolic execution* process and what's the propagation rule. *SimProcedure* allows users to replace library function calls with a fake function that models the original library function's effect on the *symbolic execution* state.

Custom Annotations. We design two custom *Annotations* to assist NDK/JNI-aware dataflow analysis:

(1) **SummaryAnnotation:** Native code uses JNI functions to create/inspect/update Java objects, invoke Java methods, catch and throw exceptions, *etc.* What's more, native code has the capability to conduct inter-component communication (ICC) with the aid of JNI functions. Therefore, *NativeDroid* implements *SummaryAnnotation* to capture data related to Java operations in native code.

(2) **TaintAnnotation:** It annotates tainted data with information, such as, taint type (source or sink), taint label, taint locations, *etc.* There are two kinds of source and sink APIs in native world: 1) Linux system calls; 2) JNI functions which invokes Java world methods. We annotate all of them to capture all the possible taint information.



0	reserved0
...	...
34	*CallObjectMethod
...	...
104	*SetObjectField
...	...
169	*GetStringUTFChars
...	...

0	reserved0
1	reserved1
2	reserved2
3	*DestroyJavaVM
4	*AttachCurrentThread
5	*DetachCurrentThread
6	*GetEnv
7	*AttachCurrentThreadAsDaemon

Figure 5: JNINativeInterface and JNIInvokeInterface structures

JNI Function Model. There are two key data structures in JNI, *JNINativeInterface* [4] and *JNIInvokeInterface* [2]. As Figure 5 illustrated, both of them contains a list of function pointers. *JNIEnv ** and *JavaVM ** are the pointers which points to the head of each table.

(1) *JNINativeInterface* provides JNI functions to create/inspect/update Java objects, invoke Java methods, catch and throw exceptions, query Java class information, *etc.* For example, *CallObjectMethod* function is used to call a Java instance method from a native method; *SetObjectField* sets the value of an instance field of an object. As native code of Figure 1 shows, each native function receives an *JNIEnv ** as its first argument, and can invoke JNI functions based on it.

(2) *JNIInvokeInterface* provides JNI functions to create/destroy Java VM, and allocate/discover *JNIEnv*. *EP* of native Activity does not have *JNIEnv ** parameter. Therefore, developer need to use *GetEnv()* function to discover the thread's *JNIEnv **. If the thread has not been created, developer needs to use *AttachCurrentThread()* or *AttachCurrentAsDaemon()* function to attach a thread and allocate *JNINativeInterface*.

Understanding the semantics of the aforementioned JNI functions are essential for *ADA* to do NDK/JNI-aware analysis. Therefore, we need to model each of the JNI functions in *JNINativeInterface* and *JNIInvokeInterface* using the *SimProcedure* technique provided by *angr*. However, the invocation instructions for JNI functions are stripped in released version of Android applications, and the JNI function calls happen through indirect jump in the function pointer table of those two data structures. Therefore, we have to create a fake data structures to imitate *JNINativeInterface* and *JNIInvokeInterface*, and set the corresponding function pointers at each offset to address of our modeled *SimProcedures*.

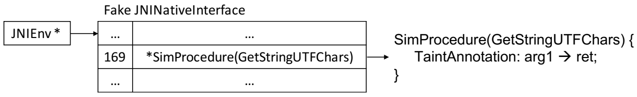
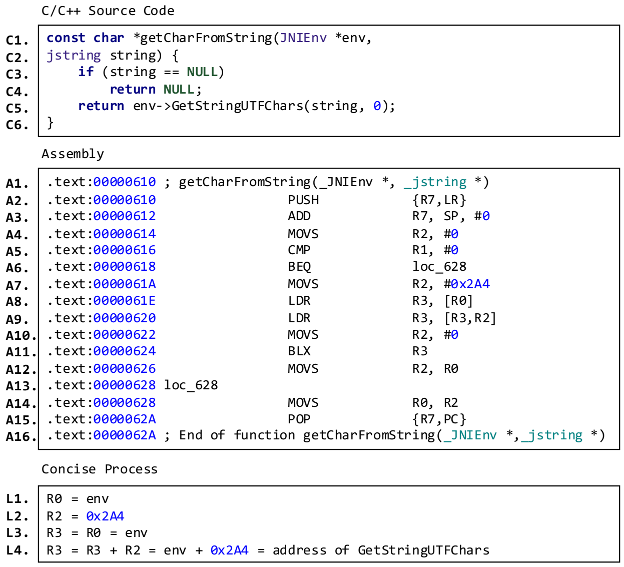

Fake JNINativeInterface
JNIEnv *
...
...
169
*SimProcedure(GetStringUTFChars)
...
...
SimProcedure(GetStringUTFChars) {
TaintAnnotation: arg1 → ret;
}

Figure 6: GetStringUTFChars function model.


C/C++ Source Code

```
C1. const char *getCharFromString(JNIEnv *env,
C2. jstring string) {
C3.     if (string == NULL)
C4.         return NULL;
C5.     return env->GetStringUTFChars(string, 0);
C6. }
```

Assembly

```
A1.  .text:00000610 ; getCharFromString(_JNIEnv *, _jstring *)
A2.  .text:00000610                 PUSH            {R7,LR}
A3.  .text:00000612                 ADD             R7, SP, #0
A4.  .text:00000614                 MOVS            R2, #0
A5.  .text:00000616                 CMP             R1, #0
A6.  .text:00000618                 BEQ             loc_628
A7.  .text:0000061A                 MOVS            R2, #0x2A4
A8.  .text:0000061E                 LDR             R3, [R0]
A9.  .text:00000620                 LDR             R3, [R3,R2]
A10. .text:00000622                 MOVS            R2, #0
A11. .text:00000624                 BLX             R3
A12. .text:00000626                 MOVS            R2, R0
A13. .text:00000628 loc_628
A14. .text:00000628                 MOVS            R0, R2
A15. .text:0000062A                 POP             {R7,PC}
A16. .text:0000062A ; End of function getCharFromString(_JNIEnv *,_jstring *)
```

Concise Process

```
L1. R0 = env
L2. R2 = 0x2A4
L3. R3 = R0 = env
L4. R3 = R3 + R2 = env + 0x2A4 = address of GetStringUTFChars
```

Figure 7: getCharFromString function source code and assembly

Figure 6 illustrates our model of *JNINativeInterface* and its *SimProcedure* table. The model of *GetStringUTFChars* indicates that the TaintAnnotation of the first argument is passed to return value. For example, Figure 7 shows a native function *getCharFromString* that receives an *JNIEnv *env* as its first argument at *C1*. It invokes *GetStringUTFChars()* function from *env* at *C5*. As Figure 6 illustrated, *GetStringUTFChars* is the 170th element of *JNINativeInterface*. Therefore, its offset to *JNIEnv ** is $169 * 4 = 676 = 0x2A4$. As the calling convention prescribed, the first argument of each function is stored in *R0* register. We illustrate the register value update process in the **Concise Process** of Figure 7 which simplifies the procedures showed in **Assembly** code. First, *R0* register is assigned to the value of *env* (a pointer) parameter at *L1*. Second, *R2* is assigned to *0x2A4* at *L2*, which is the offset of *GetStringUTFChars* from *JNIEnv **. Then, *R3* is updated with the value of *R0* at *L3*, which equals the *env* parameter. Finally, add *R2* to *R3* to get the address of *GetStringUTFChars*. *BLX R3* instruction at *A11* will call the *GetStringUTFChars*. When *ADA* executes *A11*, it will call *SimProcedure(GetStringUTFChars)*, which will propagate any *TaintAnnotations* from first argument to the return value.

Java Method Summary. As showed in Figure 1, C9 invokes *CallVoidMethod()* function which will make a Java method call and callee is *MainActivity.toNativeAgain()*. *SBDA* already generated a method summary for *MainActivity.toNativeAgain()*, which is $\Delta(toNativeAgain) = \langle(sink(arg1)@\texttt{C15})\rangle$. The function model *SimProcedure(CallVoidMethod)* takes $\Delta(toNativeAgain)$ and operates on its arguments to properly mark *TaintAnnotations*. For this case, the *data.str* will be marked as leak.

Inter-Component Communication (ICC) Resolution. Native code can make inter-component communication (ICC) by invoking Java *ICC* APIs. *Amandroid* has a comprehensive model for *ICC* [43, 44], thus we apply the same model in function model *SimProcedure(CallVoidMethod)* to capture the possible *ICC* in native code.

3.4 Challenge 4: Handling Native Activity

Android NDK allows the developer to develop Activity in pure native language since Android 2.3 [1]. There are two ways to implement a native Activity [7].

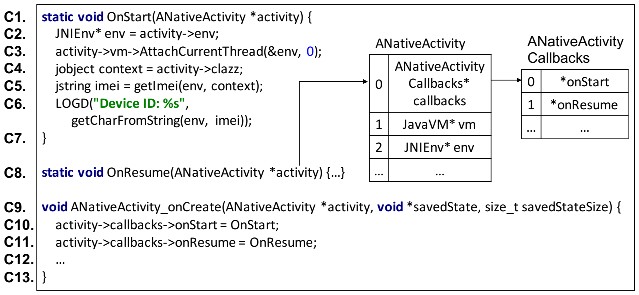
(1) **native_activity.h:** In this way, the app needs to include *native_activity.h* header to implement a native activity. It contains the callback interface and data structures that are required to create a native activity. The default entry point is *ANativeActivity_onCreate* function. NDK allows developers to use a customized function name by specifying in Manifest.

(2) **android_native_app_glue.h:** With include *android_native_app_glue.h*, an app can utilize *android_main* as entry point function to implement a native Activity.

Algorithm 3 Collect Native Activity Info of APK *A*

```
Input: Manifest file and all classes' IR of A.
Output: A's native Activity information, native_activities
procedure COLLECTNATIVEACTIVITYINFO(A)
    native_activities ← empty set
    manifest ← A.getManifest()
    for all compTag ∈ manifest.getComponentTags() do
        compName ← compTag.getAttribute("android:name")
        compClass ← A.getClass(compName)
        if compClass.isChildOfIncluding("android.app.NativeActivity") then
            map ← compTag.getMetaDataMap()
            libs ← empty set
            libName ← map("android.app.lib_name")
            if libName = null then
                libs ← resolveLibNameSet(A, compClass)          ▷ Invoke Algorithm 1
            else
                libs ← libs :: libName
            funcName ← map("android.app.func_name")
            if funcName = null then
                if libs = empty then
                    libs ← A.getAllNativeLibs()
                for all lib ∈ libs do
                    if lib.hasSymbol("android_main") then
                        libName ← lib
                        funcName ← "android_main"
                    else if lib.hasSymbol("ANativeActivity_onCreate") then
                        libName ← lib
                        funcName ← "ANativeActivity_onCreate"
            native_activities ← (compName, libName, funcName)
    return native_activities;
```

There are three important information needed for resolving a native Activity: name, containing *so* file and entry function name. Algorithm 3 shows the pseudocode for collecting these for all native Activities from an app *A*. We first iterate each component *compClass* in the *AndroidManifest.xml* and find the native Activities by check whether *compClass* is or is the child of *"android.app.NativeActivity"*. If *compClass* is a native Activity, we then read its metadata to obtain the *libName*. If did not get *libName*, we then evaluate *compClass*'s static initializer *<clinit>* to find out the argument value for load library method calls, *System.load()*, *System.loadLibrary()*, *Runtime.load()*, and *Runtime.loadLibrary()*. Then assign it to *libName*. We read the *"android.app.func_name"* from *compClass*'s metadata to obtain the *funcName*. If *"android.app.func_name"* does not exist, then the default entry function name is used. We then check if the default name is *"android_main"* (the android_native_app_glue.h case) or *"ANativeActivity_onCreate"* (the native_activity.h case).



```
C1.  static void OnStart(ANativeActivity *activity) {
C2.    JNIEnv* env = activity->env;
C3.    activity->vm->AttachCurrentThread(&env, 0);
C4.    jobject context = activity->clazz;
C5.    jstring imei = getImei(env, context);
C6.    LOGD("Device ID: %s",
          getCharFromString(env,  imei));
C7.  }

C8.  static void OnResume(ANativeActivity  *activity) {…}

C9.  void ANativeActivity_onCreate(ANativeActivity  *activity, void *savedState, size_t savedStateSize) {
C10.   activity->callbacks->onStart = OnStart;
C11.   activity->callbacks->onResume = OnResume;
C12.   …
C13. }
```

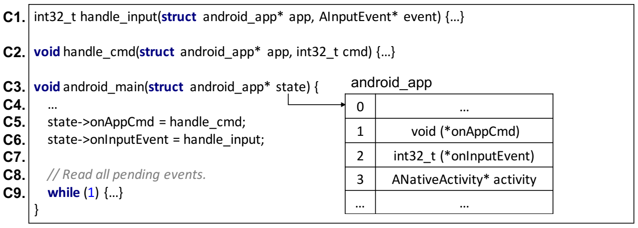
Figure 8: native_activity.h example

native_activity.h. As Figure 8 illustrated, the default *EP* of the native Activity is *ANativeActivity_onCreate* (NDK also allows developers to use a custom *EP*). *ANativeActivity ** is the first parameter whose first structure member is *ANativeActivityCallbacks *callbacks*. *ANativeActivityCallbacks* structure contains the callback functions which will be executed in the native activity lifecycle. However, when we conduct the *ADA* from *EP*, the symbolic execution engine cannot execute those callbacks, as there are no explicit calls.
To comprehensively model this type of native Activity we take a two fold approach:

(1) **Resolve callback function address:** As illustrated in Figure 8, the *ANativeActivity_onCreate* function assigns the callbacks to corresponding index of *ANativeActivityCallbacks* structure. We apply symbolic execution on this *EP* to get addresses of those callbacks and its index in *ANativeActivityCallbacks* structure.
We first construct a fake *ANativeActivityCallbacks* structure. We then construct a fake *ANativeActivity* structure and map the fake *ANativeActivityCallbacks* structure's pointer to the *ANativeActivity* structure. Finally, we assign the pointer to the fake *ANativeActivity* structure to the first argument (*R0* register) of *ANativeActivity_onCreate*. We do the under-constrained symbolic execution from *ANativeActivity_onCreate* function. After the symbolic execution has finished, the elements of *ANativeActivityCallbacks* will be assigned real addresses of those callbacks.

(2) **Explicitly invoke callback functions:** We hook each callback function to *ANativeActivity_onCreate* and apply *ADA* from *ANativeActivity_onCreate* as the *EP*. One challenge here is when native Activity invokes JNI functions. As illustrated in Figure 8, there are no *JNIEnv ** in the *EP*, and the *ANativeActivity* structure's *JNIEnv ** is uninitialized. The developers need to invoke

AttachCurrentThread on *JavaVM ** to assign *env* like in C2 and C3. In *ADA*, we apply *SimProcedure(AttachCurrentThread)* to assign *env* element. After the *env* element is assigned, the *ADA* will be able to correctly resolve JNI functions.



```
C1. int32_t handle_input(struct android_app* app, AInputEvent* event) {...}

C2. void handle_cmd(struct android_app* app, int32_t cmd) {...}

C3. void android_main(struct android_app* state) {
C4.     ...
C5.     state->onAppCmd = handle_cmd;
C6.     state->onInputEvent = handle_input;
C7.
C8.     // Read all pending events.
C9.     while (1) {...}
    }
```

android_app

0	...
1	void (*onAppCmd)
2	int32_t (*onInputEvent)
3	ANativeActivity* activity
...	...

Figure 9: android_native_app_glue.h example

android_native_app_glue.h. As illustrated in Figure 9, *android_main* is the *EP*, and the only argument is the *android_app * state.* There are two important callback function pointers in *android_app* structure, *onAppCmd* and *onInputEvent. onAppCmd* is used for activity lifecycle events and *onInputEvent* is used for input events. Developers need provide their own processing functions to the two callbacks. These callbacks will be triggered when an activity and an input event occur, respectively.

To comprehensively model this native Activity type we apply similar approach as we used to resolve *ANativeActivity_onCreate*. Firstly, We run symbolic execution from *android_main* to resolve the two callbacks value. Then, we hook the two callbacks to *android_main* function and run *ADA*.

4 THE *JN-SAF* FRAMEWORK

JN-SAF consists of *JavaDroid*, *NativeDroid* and *JNI Bridge*. *JavaDroid* is responsible for Dalvik-bytecode (Java world) analysis. It is implemented on top of *Amandroid* [43, 44], which provides various static analysis modules to perform custom analysis of Android apps. However, *Amandroid* does not readily have inter-language analysis capability. Thus, we have to implement the *Summary-based Bottom-up Dataflow Analysis* (SBDA) algorithm as described in Section 3.1. *NativeDroid* is responsible for binary code (native world) analysis, which is built on top of *angr* [36]. *NativeDroid* implements the *ADA* algorithm described in Section 3.3. *JNI Bridge* is the middle layer that assists the control and data communication between *JavaDroid* (implemented in Scala[2]) and *NativeDroid* (implemented in Python). *JNI Bridge* leverages jpy [5], a bi-directional Java-Python bridge to enable *JavaDroid* and *NativeDroid* transfer control and data.

Figure 10 illustrates the pipeline of *JN-SAF* which consists of three major steps: 1) *APK Preprocess*: collects useful information from an app; 2) *Environment Model*: generates environment model for both Java and native components; 3) *Summary-based Bottom-up Dataflow Analysis (SBDA)*: computes information flow for each Android component in a native-aware fashion and apply inter-component analysis to evaluate security problems.

4.1 APK Preprocess

JN-SAF takes an *APK* as the analysis input. It decompiles the *APK* into three parts, *dex* files, *Manifest&Resource* files and *so* files. *JavaDroid* leverages the *DEX2IR* and *Resources Parser* components in *Amandroid* to decompile Dalvik bytecode into *Intermediate Representation* (IR) language *Pilar* [44] and collect component information. *NativeDroid* uses *pyvex* from *angr* to translate binary into *VEX IR* [35].

The *Native Info Analyzer* receives information from *DEX2IR* and *Resources Parser* to compute native world related information:

(1) Generate *Native Method Mapping* following Algorithm 2 described in Section 3.2.
(2) Collect *Native Activity Info* following Algorithm 3 described in Section 3.4.

4.2 Environment Model

Android is an event-based system, and as such no single method can be used as *EP* for the dataflow analysis. To capture all lifecycle and event control-/data-flow of an Android Java component, and to generate *EP* for dataflow analysis, *APK Preprocessor* reuses *Environment Builder* from *Amandroid* to build environment model for each Android Java component as described in [43, 44], and generates an *Environment Method* as the *EP* for each Java component.

We implement *Native Component Environment Builder* following the solution described in Section 3.4 to generate an *Environment Function* as the *EP* for each native Activity component.

The *Environment Method/Function* explicitly invokes the event/lifecycle callbacks as the Android runtime would.

4.3 Summary-based Bottom-up Dataflow Analysis (SBDA)

JN-SAF implements the *Summary-based Bottom-up Dataflow Analysis* (SBDA) algorithm by following the techniques described in Section 3.1. It consists of the following components.

Call Graph Builder. It receives the *environment method/function* from *Environment Model* and uses it as the *EP* to compute a native-aware call graph. Unlike traditional Java call graph building algorithm, our call graph will not stop at native method calls. Instead, it will evaluate the corresponding native function to address possible reflection call from native to Java and add those call target as callee of this native method. The native reflection style call is resolved by following the JNI function model described in Section 3.3.

Bottom-up Summary Propagator. It receives the call graph *CG* from *Call Graph Builder* and applies a topological sort with the reverse order to get a list of method/function *MList.* It iterates the *MList* to send the work order to corresponding *Method/Function Summary Builder* to compute summary Δ, and propagate to their callers.

Java Method Summary Builder. *Amandroid* provides a flow and context-sensitive monotonic dataflow analysis engine [44]. We can leverage this engine to compute the summary for a given method. However, *Amandroid* is not aware of our summary representation and it always does a inter-procedural analysis. We thus

[2] Scala is a JVM-based language.

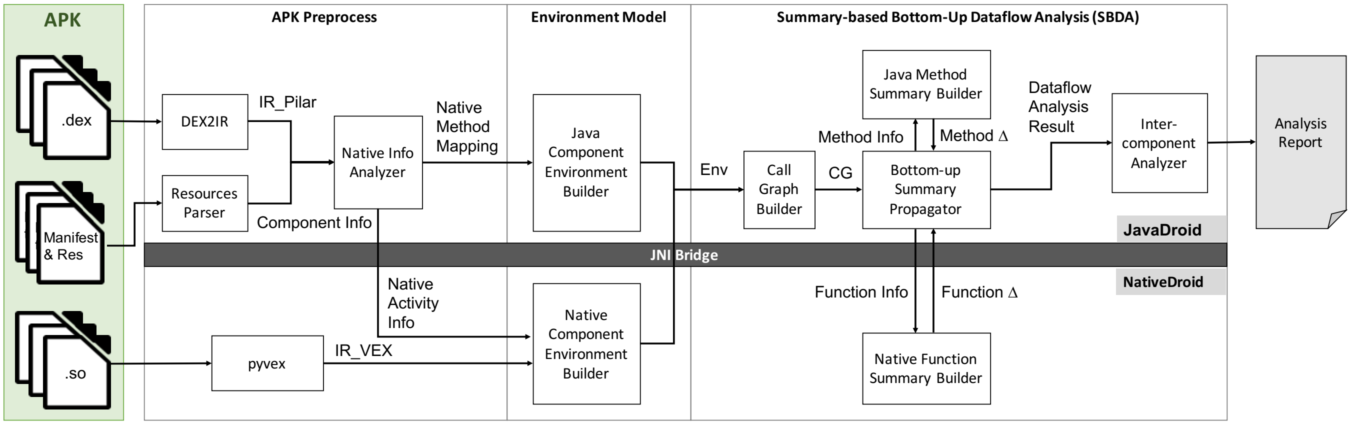


Figure 10: The *JN-SAF* pipeline.

significantly modified its dataflow analysis engine. When the engine reaches a method call, it will not flow the points-to facts into the callee. Instead, it will obtain the summary $\Delta(callee)$ and apply such summary on current points-to facts to imitate the heap manipulation behaviors. When dataflow analysis finishes, we collect the heap manipulation behavior of the current method and generate a summary $\Delta(method)$.

Native Function Summary Builder. Upon receiving a work order with native method signature and its containing *so* file, the *Native Function Summary Builder* first identifies the binary address for the corresponding native function of the native method. Then it applies *ADA* (as described in Section 3.3) to generate Δ starting from such *EP* as follows.

(1) Add *SummaryAnnotation* to each argument including argument index and type information, because from *EP*'s perspective all mutable arguments are considered as *HeapBase*.
(2) Add *SimProcedure* to all JNI functions which might create/delete/manipulate the heap of Java objects. When *ADA* evaluates, those *SimProcedures* will properly update and propagate *SummaryAnnotation*. As an example, native code can construct Java *String* with the aid of JNI function *NewString()* or *NewStringUTF()*, JNI function *SetObjectField()* will set data to a Java object.
(3) When *ADA* encounters any method/function invocation, it will check whether it is a source or sink API. If so *ADA* will add *TaintAnnotation* to proper *HeapLocs*. For method invocation, we will also check with *SBDA* to obtain its Δ and apply it on the arguments *SummaryAnnotations*.
(4) When *ADA* is over, we extract the *SummaryAnnotation* together with *TaintAnnotation* related to each arguments and return node (if the JNI function returns a Java object) to build the summary.

We take *Java_test_multiple_1interactions_MainActivity_propagateData()* function at Figure 1 as an example to walkthrough the native function Δ building process. *Java_test_multiple_1interactions_MainActivity_propagateData()* function receives one argument *data*. We assign *SummaryAnnotation(arg1, test.multiple_interactions.Data)* to *data* and *SummaryAnnotation(arg1.str, 'java.lang.String')* to *data.str*. C6 invokes *GetObjectField()* to read *str* field of *data* to variable *imei*. *SimProcedure(GetObjectField)* get *SummaryAnnotations* from *data.str* and propagate it to variable *imei*. C9 invokes Java method *toNativeAgain()* and pass *imei* as the first argument. *SimProcedure(CallVoidMethod)* obtain $\Delta(toNativeAgain)$ from *SBDA*, and apply on *SummaryAnnotations* of *imei*, we then get *TaintAnnotation(sink(arg1.str), 'C15')*. After finish running *ADA*, we collect the *SummaryAnnotations* and *TaintAnnotations* related to each argument (there are no return value in this case). Finally, we check the heap changes of each *HeapBase* and taint informations to construct the summary $\Delta(propagateImei) = \langle(sink(arg1.str)@C15)\rangle$.

Inter-component Analyzer. Resolving Inter-component communication (ICC) is essential for any Android static analysis tool. *JN-SAF*'s *ICC* resolution is empowered by *Amandroid*'s *Summary Table* (ST) based *ICC* resolution model [44]. The *Inter-component Analyzer* collects *ICC* information from all Java components and native Activity components. Then, it computes *ST* for each component and uses *Amandroid*'s *Component-based Analysis* to address *ICC* dataflow.

5 EVALUATION

We evaluated *JN-SAF* extensively on benchmark and real world apps. Our dataset includes: (1) NativeFlowBench created by us which consists of 22 hand-crafted benchmark apps, each testing one perspective of the inter-language challenges; (2) 100,000 randomly selected popular apps from AndroZoo [11] (ZOO); (3) 24,553 malware apps from the AMD dataset [42] (AMD).

We perform experiments to answer the following research questions (RQ):

> **RQ1:** What is the statistics of native library usage in real world Android apps?
> **RQ2:** How does the running time of *JN-SAF* scale?
> **RQ3:** How does *JN-SAF* perform on Benchmark apps?
> **RQ4:** Is *JN-SAF* capable of discovering crucial security issues to aid in real-world app vetting?

We ran our experiments on a machine with 2.20 GHz, 48-core Xeon, and 256 GB RAM.

5.1 RQ1: What is the statistics of native library usage in real world Android apps?

Table 1: Native library statistics for datasets.

(a) Native library usage.

	ZOO	AMD		ZOO	AMD
Total App [a]	99,910	24,384			
Has Native [b]	39,661	5,365	/ Total App	39.7%	22.0%
Has .so File	35,705	5,164	/ Has Native	90.0%	96.2%
Has Native Method	32,576	3,867	/ Has Native	82.1%	72.1%
Has Native Activity	583	29	/ Has Native	1.5%	0.5%
Total Native Method	4,232,699	112,000	/ Has Native Method	106.7	29.0
Pass Data	3,661,881	90,212	/ Total Native Method	86.5%	80.5%
Pass Object	1,496,911	45,981	/ Pass Data	35.4%	51.0%

[a] We failed to analyze a few apps that use advanced obfuscation.
[b] Has Native = Has .so File ∪ Has Native Method ∪ Has Native Activity.

(b) Architecture.

	ZOO	AMD		ZOO	AMD
Total .so File	235,616	16,116			
ARM	162,356	13,792	/ Total .so File	69.0%	85.6%
ARM 64	10,111	2	/ Total .so File	4.3%	0.01%
X86	37,745	1,149	/ Total .so File	16.0%	7.1%
X86 64	8,511	2	/ Total .so File	3.6%	0.01%
MIPS	9,658	770	/ Total .so File	4.1%	4.8%
MIPS 64	2,477	2	/ Total .so File	1.1%	0.01%
Other	4,758	399	/ Total .so File	2.0%	2.5%

(c) Reflection call.

	ZOO	AMD		ZOO	AMD
Total Reflection Call	7,664[a]	33,497			
Resolved Call	4,744	29,336	/ Total Reflection Call	61.9%	87.6%
Library API Call	2,555	24,249	/ Resolved Call	53.9%	82.7%
App Method Call	2,189	5,087	/ Resolved Call	46.1%	17.3%

[a] Due to time constraint we only finished analyzing 37,781 native functions from *ZOO*.

We collect native library usage on both *ZOO* and *AMD*. As Table 1a indicates, the overall native library usage is reasonably high no matter in benign dataset or malware dataset. *ZOO* has much higher native library usage than *AMD* which means there are many benign use cases for native libraries, so native library existence is not a good indicator for detecting Android malware. We really need to dig into the native library and understand its behavior. We also found cases where an app has native methods but no .so files. This means the .so file is probably downloaded at runtime (in which case no static analyzer will be able to identify). We found native Activity usage in both *ZOO* and *AMD*, which shows the necessity of handle such case.

Table 1b lists the usage of different architectures. Overall, 32 bit architecture has much higher percentage over 64 bit architecture. *ARM* is the most popular architecture for Android. Not surprisingly most of the binaries are in *ARM* architecture.

Native library can invoke Java method through reflection style function calls. We conducted an experiment to study the capability of *NativeDroid* to resolve such calls, and the results are shown in Table 1c. We also studied the distribution of those reflection call targets, and found that the majority of the reflection calls (especially from *AMD*) are targeted to library APIs as oppose to App methods. We experience poor performance on *ZOO* reflection call resolving due to the larger code base and complex logic in market apps as opposed to malware apps. From the obtained reflection call list, we see many interesting library APIs being called, such as *SmsManager.sendDataMessage()*, *ClassLoader.loadClass()*, which might raise red flags.

5.2 RQ2: How does the running time of *JN-SAF* scale?

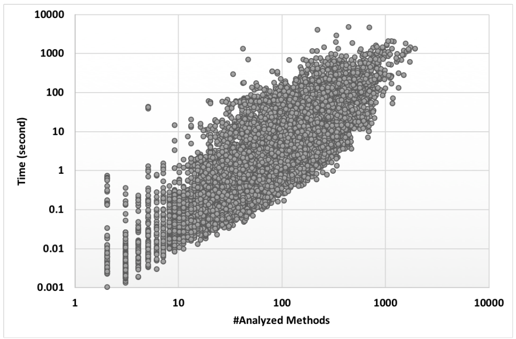
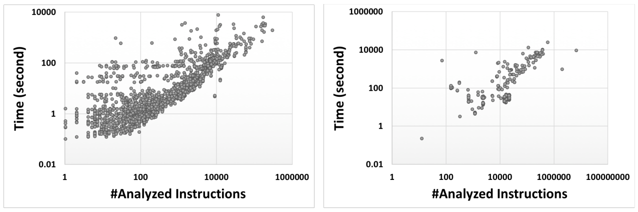


Figure 11: Time to run *SBDA*.



(a) Function Summary Builder **(b) Native Activity Analysis**

Figure 12: Native code analysis performance.

SBDA is the core engine and the most computation-intensive step in *JN-SAF*. Figure 11 presents the time taken to construct *SBDA* for 10,000 randomly picked real-world app components. These components reach 144 methods on average. The average running time for computing the *SBDA* for each component is 42.288 seconds; the minimum is 0.001 seconds whereas the maximum is 86 minutes.

We constructed a separate experiment focused on the running time for native code analysis. Figure 12a illustrates the time taken to build function summary for 2,000 randomly picked real-world app native functions. These native functions reach 4,417 instructions on average. The average running time is 88.982 seconds; the minimum is 0.107 seconds whereas the maximum is 136 minutes. Figure 12b illustrates the time taken to construct native Activity analysis for all 579 native activities (failed to analyze 33 due to path explosion problem). These native activities reach 41,285 instructions on average. The average running time is 570.513 seconds; the minimum is 0.247 seconds whereas the maximum is 438 minutes.

5.3 RQ3: How does *JN-SAF* perform on Benchmark apps?

For evaluation purpose, we designed *NativeFlowBench* since there is no existing benchmark for evaluating inter-language dataflow analysis capability of Android static analysis tools. *NativeFlowBench* contains a set of hand-crafted apps designed to test specific analysis features. Since those apps are hand-crafted, the ground truth is known and we can compute metrics like precision and recall.

Table 2: Results on NativeFlowBench.

App Name	*JN-SAF*	*Amandroid*	*FlowDroid* *IccTA*	*DroidSafe*
Part A: Inter-language Dataflow				
native_source	O	X	X	X
native_nosource				
native_source_clean		*	*	
native_leak	O	X	X	X
native_leak_dynamic_register	O	X	X	X
native_dynamic_register_multiple	O	X	X	X
native_noleak				
native_noleak_array	*			
native_method_overloading				
native_multiple_interactions	O	X	X	X
native_multiple_libraries	O	X	X	X
native_complexdata	O	X	X	X
native_complexdata_stringop	*			
native_heap_modify	O	X	X	X
native_set_field_from_native	OO	XX	XX	XX
native_set_field_from_arg	OO	XX	XX	XX
native_set_field_from_arg_field	OO	XX	XX	XX
Part B: Native Activity Resolve				
native_pure	O	X	X	X
native_pure_direct	O	X	X	X
native_pure_direct_customized	O	X	X	X
Part C: Inter-component Communication				
icc_javatonative	O	X	X	X
icc_nativetojava	O	X	X	X
Sum, Precision and Recall				
O, higher is better	19	0	0	0
*, lower is better	2	1	1	0
X, lower is better	0	19	19	19
Precision p = O/(O + *)	90.5%	0.0%	0.0%	N/A
Recall r = O/(O + X)	100%	0.0%	0.0%	0.0%
F-measure 2pr/(p + r)	95.0%	N/A	N/A	N/A

O = True Positive, * = False Positive, X = False Negative.
We applied *IccTA* for handle part C: Inter-component Communication.

NativeFlowBench contains 22 apps categorized in three parts: Part A focuses on inter-language dataflow analysis challenges: native source and sink finding, native method to native function resolving, JNI library function modeling, native dataflow analysis with Java objects, *etc.* Part B focuses on the native Activity resolving. Part C focuses on inter-component communication between Java and native components. We will make *NativeFlowBench* publicly available. The apps in these testsuites are not crafted to favor a particular tool. They present common scenarios one will find when reasoning about the relevant security issues.

We compare the effectiveness of *JN-SAF* with all other major Android static analysis tools: *Amandroid* [43, 44], *FlowDroid* [12], *IccTA* [23], *DroidSafe* [21]. We run each tool against each of the benchmark apps to check if the tool can report the correct data leak paths, and the detailed comparison is reported in Table 2. The results are shown in terms of True Positive (O), False Positive (*) and False Negative (X), if any. If an app has more than one leakage path, then the result is shown for each of them. Not surprisingly, *JN-SAF* outperforms all other tools as none of the existing Android static analysis tools have inter-language analysis capability. *DroidSafe* is outdated and failed to analyze any of the benchmark apps. *Amandroid* and *FlowDroid* both identified one false path at *native_source_clean.* This is caused by their conservative model for native method calls − if one of the argument is tainted all other arguments will also be considered as tainted. *IccTA* failed to handle the inter-component communication cases due to the lack of native code resolution. *JN-SAF* has false alarm on *native_noleak_array* because *JN-SAF* cannot distinguish different index of an Java array. *JN-SAF* has false alarm on *native_complexdata_stringop* because *JN-SAF* does not do precise string analysis.

5.4 RQ4: Is *JN-SAF* capable of discovering crucial security issues to aid in real-world app vetting?

We evaluated *JN-SAF* on AMD [42] dataset to examine its capacity of real-world app security vetting. AMD is an Android malware ground truth dataset which contains 24,553 samples categorized in 71 malware families. AMD reported 9 malware families that contain native payload [42], and *JN-SAF* is able to detect 8 of them. The missed one is *Lotoor* which is a family of all the rooting tools[3]. We discuss in detail our findings in the following 4 case studies.

5.4.1 Case Study 1: Inter-language Data Leakage

Sensitive information leakage has been a widespread security issue in Android platform. To make detection harder, malware moves the leaky behavior into native world. *JN-SAF* detected two malware families which has such behavior.

Triada obtains the *IMSI* of device in Java layer. Then it passes the *IMSI* to native method *nativeSayTest().* The corresponding native function will then leak *IMSI* by invoking *SmsManager.sendTextMessage(). JN-SAF* detects this issue by generate a summary $\Delta(nativeSayTest) = \langle(sink(arg2)@\texttt{Cx})\rangle$ and feed back to *SBDA*. *SBDA* marks the *IMSI* as source and when *nativeSayTest()* is invoked with such source the leak issue is reported.

Similar to *Triada*, *Gumen* gains the *IMEI* of device in Java layer. Then it propagates the *IMEI* taint source to the third argument of native method *stringFromJNI()*, which leaks *IMEI* by invoking *SmsManager.sendTextMessage(). JN-SAF* utilizes the same detection procedure for detecting *Triada* family. The generated summary is $\Delta(stringFromJNI) = \langle(sink(arg3)@\texttt{Cx})\rangle$.

5.4.2 Case Study 2: Stealthy Command Execution

Malware writers love to use shell command to execute malicious behaviors. For example, *DroidKungFu* is a backdoor malware that try to root device and execute malicious code. It roots the device with the aid of *secbino* program. If the device has not been rooted, it will copy *secbino* to */data/data/pkg/secbino* and *chmod 4755* to get the execution permission. Then it executes *secbino* to get the root privilege and start a service to download other malware apks to install.

JN-SAF detects these behaviors by modeling those Linux programs that can execute shell command, such as, *popen*, *system*, *execv etc.JN-SAF* is able to get the parameters of those system API and know what shell commands are executed.

5.4.3 Case Study 3: Stealthy C&C Communication

Command and Control(C&C) server is frequently used in malware to conceal the malware command and control information generation process into network communication. This process can also move to native world. *JN-SAF* detected a malware familty *Boqx* which hide its C&C communication in the native payload.

[3] Rooting behavior is hard to detect since each rooting method has complex and quite different semantics.

Boqx launches a thread to exec native code in *StatService* class. In the native world, it enables the *WIFI* to ensure the success of communicating with a server. Then it communicates with the server to get the malicious payload and then dynamicly loads these payloads. All these behaviors are completed by native reflection calls. *JN-SAF* models all the JNI functions from *JNINativeInterface* structure. After running *ADA*, we can know what kind of reflection calls are made in the native world.

5.4.4 Case Study 4: Malicious Identity Hiding

Malicious identity such as server *URL* and premium number is important for many malware analysis techniques. *JN-SAF* detects two malware families *Ogel* and *UpdtKiller* that hide those identities in the native world.

Ogel encapsulates its *C&C* server *URL* in native code, and when it starts running it will reads the URL data by invoking a native function *Java_com_googlle_cn_ni_u()*. *Java_com_googlle_cn_ni_u()* uses *NewStringUTF()* to create a Java *String* of its *URL*. *JN-SAF* is able to obtain the value of the C&C server URL. When malware returns the server *URL* from native world to Java world through native method, *NativeDroid* can generate summary that illustrates this process $\Delta(u) = \langle (ret = URL@\texttt{Cx})(source(URL)@\texttt{Cx})\rangle$. Then *JavaDroid* will continue *SBDA* with the summary information.

UpdtKiller executes commands remotely to steal personal information, add artificial SMS messages to the inbox and intercept, auto-reply and block SMS/MMS messages without user's consent. All the sensitive data required by communicating with the remote server, including numbers and *URLs*, are stored in the native code. *UpdtKiller* get these sensitive data via invoking native methods with *Get* prefix, such as, *GetNumber(), GetUrlHost() etc.* These native methods invoke *NewStringUTF* to encapsulates the sensitive data into Java *String* and return to Java world. *NativeDroid* generates summary $\Delta(GetNumber) = \langle (ret = N@\texttt{Cx})(source(N)@\texttt{Cx})\rangle$, and feed back to *JavaDroid*.

6 DISCUSSION

The inter-language related operations such as JNI reflection call construction, dynamic function registration, and Intent value resolution, all require precise resolution of string values. *JN-SAF* does constant string propagation in both *JavaDroid* and *NativeDroid*. If the string is manipulated *JN-SAF* will not be able to construct the precise value. Precise string analysis is expensive and non-trivial in both Java analysis and binary analysis as mentioned in prior research [18, 22, 33]. We leave this for future research.

JavaDroid inherits some limitations from *Amandroid* [44]: 1) It does not handle Java reflection and dynamic class loading in the Java world; 2) The precision and soundness of summary generation depends on the faithfulness of the library API models; 3) It cannot handle fine-grained concurrent execution. *NativeDroid* inherits path explosion issues from *angr* [36]. Control-/Data-flow analysis of *NativeDroid* is mainly based on the symbolic execution engine of *angr*. Path&State explosion are the natural defect of any symbolic execution techniques when encountering large programs as the analysis need to separate all the states for different execution paths. To alleviate explosion problem, *NativeDroid* needs to better constrain the possible execution paths and states which are non-trivial [14]. We will handle these limitations in future work.

To evade detection of static analysis, both Java and native code can be obfuscated with techniques such as string encryption and dynamic code loading. *JN-SAF* currently does not provide a solution for such obfuscation. Anti-obfuscation techniques such as [30] could be applied to improve the detection capability of *JN-SAF*.

7 RELATED WORK

JN-SAF is a static and cross-layer analysis framework that includes analysis for the native world of Android apps. Below we describe three categories of works that are most closely related to ours.

7.1 Android Static Analysis

FlowDroid [12] is a dataflow analysis framework for taint detection of the Android application. *FlowDroid* has an app-level *dummyMain* model to capture Android system events, then uses a flow and context-sensitive *IFDS* [31, 32] algorithm to do taint detection. *FlowDroid* avoids to handle native method invocation and applies a comprehensive model for native method calls.

Epicc [28] leverages *IFDS* on *FlowDroid* to computes Android Intent call parameters. However, it cannot resolve Intent call parameters if it presents in the native code.

IccTA [23] extends *FlowDroid* and uses *IC3* [27] as the Intent resolution engine. *IccTA* is able to track data flows through regular Intent calls and returns. *IccTA* shares the same limitation as *Flowdroid* which does not handle any native method invocations.

DroidSafe [21] is yet another dataflow analysis framework for Android application which tracks Intent communication and RPC calls. *DroidSafe* adopted a flow-insensitive points-to analysis algorithm which aims to handle all possible runtime event ordering. *DroidSafe* does not handle native method call as well.

CHEX [25] is designed to detect component hijacking problem in Android. *CHEX* is built on top of *Wala* [20], it first constructs *app-splits*, each of which is a code segment reachable from an *EP*, then uses the dataflow engine from *Wala* to computes the dataflow summary for each of the *app-split*. The *app-splits* summaries are then linked in all possible permutations to detect possible information flows. *CHEX* does not handle native method call.

SInspector [34] is designed to detect UNIX domain socket misuses. *SInspector* uses *Amandroid* to generate Java layer dataflow and uses *IDA Pro* to capture native dataflow. However, *SInspector* does not track inter-language data flows nor model JNI functions.

Amandroid [43, 44] is a general flow and context-sensitive ICC-aware dataflow analysis framework for security vetting of Android applications. *Amandroid* generates environment model for each Android component and applies a component-based analysis algorithm to capture all possible intra-/inter-component data flows. However, like all other Android static analysis framework, *Amandroid* does not handle native method calls. *JavaDroid* of *JN-SAF* is built on top of *Amandroid*, which leverages many features from *Amandroid* and provides a naive and comprehensive approach to handle native method invocations and inter-language data flows.

7.2 Binary Code Analysis

BitBlaze [37] is a hybrid binary analysis platform, which contains three components: 1) *Vine*: a static analysis component that translates assembly to *IR*, which supports *x86* and *ARMv4* architectures; 2) *TEMU*: It enables whole-system monitoring and dynamic binary instrumentation; 3) *Rudder*: It utilizes *Vine* and *TEMU* to conduct symbolic execution.

BAP [16] is binary analysis platform which supports *x86* and *ARM* architectures. *BAP* re-designs *Vine* to assist its front-end features. After the *IR* translations process finished, *BAP* conducts its back-end analysis in the *IR* granularity.

angr [36] is a binary analysis framework that combines many existing program analysis technique into a single, coherent framework, such as, *Dynamic Symbolic Execution*, *Veritesting*, *Value-Set Analysis (VSA). angr* leverages the *IR* lifter of *Valgrind* [26] to translate assembly to *VEX IR*, With the aid of *VEX IR*, *angr* provides analysis support for many architectures including 32-bit and 64-bit versions of *ARM*, *MIPS*, *PPC*, *x86*. *NativeDroid* of *JN-SAF* is built on top of *angr* and uses its *SimProcedure* and *Annotation* features to model NDK libraries and JNI functions.

7.3 Dynamic&Hybrid Analysis with Native Information Tracking

DroidScope [46] is an Android application dynamic analysis tool that reconstructs *OS* level and *DVM* level information. *DroidScope* collects detailed native and *Dalvik* instruction traces, profile API-level activity, and track information leakage through both the Java and native components using dynamic taint analysis.

NDroid [29] performs dynamic taint analysis based on *QEMU* and tracks information flows through *JNI*. *NDroid* instruments important related JNI functions to resolve information flows, such as JNI entry, JNI exit, object creation. Moreover, It models the system library instead of instrumenting those standard functions to reduce overhead. However, similar to all dynamic analysis systems, *NDroid* has the path coverage issue and it does not track control flows.

TaintART [39] applies dynamic taint tracking by instrumentation the *ART* compiler and runtime. *TaintART* follows *NDroid*'s method to handle JNI calls.

Harvester [30] employs hybrid analysis for extracting runtime values. When encountered with native methods, *Harvester* monitors them as logging points to extract runtime values instead of stepping into the native code to conduct the analysis.

Going Native [9] conducts static analysis to filter apps containing native code firstly and then perform dynamic analysis to study the native code usage of real-world Android apps. Then it generates native code sandboxing security policy.

Malton [45] is a dynamic analysis platform aimed to do malware detection that runs on *ART* runtime. *Malton* conducts multi-layer monitoring including native layer and information flow tracking to provide a comprehensive view of the Android malware behaviors.

DroidNative [10] utilizes specific control flow patterns to reduce the impact of obfuscations and use it as semantic-based signatures to detect malware in ART runtime.

8 CONCLUSION

In this paper, we presented the first Android static analysis framework *JN-SAF* which can track precise control and data flow across language boundary. *JN-SAF* provides a comprehensive model for JNI functions, NDK libraries, and native Activities, which enables dataflow analysis on Android binaries. *JN-SAF* leverages a summary-based bottom-up scheme to do precise and compact inter-language dataflow analysis and provides unified summary representation to integrate Java and binary analysis results. Our experiments result shows that *JN-SAF* can be readily applied to effectively address real-world Android security issues which involve native payload and inter-language communication.

ACKNOWLEDGMENTS

This research was partially supported by the U.S. National Science Foundation under grant no. 1622402 and 1717862, the Chinese National Science Foundation under grant no. 61572115, and the Chinese National Key R&D Plan under grant no. 2016QY04X000. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the above agencies.

REFERENCES

[1] 2018. Android NDK. https://developer.android.com/ndk/. (2018).

[2] 2018. The Invocation API. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/invocation.html. (2018).

[3] 2018. Java Native Interface Specification. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. (2018).

[4] 2018. JNI Functions. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html. (2018).

[5] 2018. jpy - a Python-Java Bridge. https://github.com/bcdev/jpy. (2018).

[6] 2018. Mobile Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide. (2018).

[7] 2018. Native Activities and Applications. https://developer.android.com/ndk/guides/concepts. (2018).

[8] 2018. Resolving Native Method Names. https://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/design.html. (2018).

[9] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Proceedings of the NDSS*.

[10] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and optimizing detection of Android native code malware variants. *computers & security* (2017).

[11] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the ACM MSR*.

[12] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM PLDI*.

[13] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the IEEE ICSE*.

[14] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[15] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-party Components. In *Proceedings of the USENIX Security Symposium*.

[16] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A Binary Analysis Platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

[17] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the ACM Mobisys*.

[18] Aske Christensen, Anders Møller, and Michael Schwartzbach. 2003. Precise analysis of string expressions. *Static Analysis* (2003).

[19] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the ACM PLDI*.

[20] Stephen Fink and Julian Dolby. 2012. WALA–The TJ Watson Libraries for Analysis. http://wala.sf.net/.

[21] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the NDSS*.

[22] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. 2015. String analysis for Java and Android applications. In *Proceedings of the ACM FSE*.

[23] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the IEEE ICSE*.

[24] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *Proceedings of the Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE.

[25] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM CCS*.

[26] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*. ACM.

[27] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the IEEE ICSE*.

[28] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication mapping in Android with Epicc: An Essential Step towards Holistic Security Analysis. In *Proceedings of the USENIX Security Symposium*.

[29] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. 2014. On Tracking Information Flows through JNI in Android Applications. In *Proceedings of the IEEE Dependable Systems and Networks (DSN)*.

[30] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *Proceedings of the NDSS*.

[31] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM POPL*.

[32] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* (1996).

[33] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. 2007. Abstracting Symbolic Execution with String Analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. IEEE.

[34] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z Morley Mao. 2016. The Misuse of Android Unix Domain Sockets and Security Implications. In *Proceedings of the ACM CCS*.

[35] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the NDSS*.

[36] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE S&P*.

[37] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security*. Springer.

[38] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the NDSS*.

[39] Mingshen Sun, Tao Wei, and John Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the ACM CCS*.

[40] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *Proceedings of the NDSS*.

[41] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. 2014. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the SPSM*. 39–50.

[42] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Springer, Bonn, Germany, 252–276.

[43] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM CCS*.

[44] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Transactions on Privacy and Security (TOPS)* (2018).

[45] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *Proceedings of the USENIX Security Symposium*.

[46] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.. In *Proceedings of the USENIX Security Symposium*. 569–584.

[47] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the NDSS*.