NativeSummary: Summarizing Native Binary Code for Inter-language Static Analysis of Android Apps

Jikai Wang
Huazhong University of Science and Technology
Wuhan, China
wangjikai@hust.edu.cn

Haoyu Wang*
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract

With the prosperity of Android app research in the last decade, many static analysis techniques have been proposed. They generally aim to tackle DEX bytecode in Android apps. Beyond DEX bytecode, native code (usually written in C/C++) is prevalent in modern Android apps, whose analysis is usually overlooked by most existing analysis frameworks. Although a few recent works attempted to handle native code, they suffer from scalability and accuracy issues. In this paper, we propose NativeSummary, a novel inter-language static analysis framework for Android apps with high accuracy, scalability, and compatibility. Our key idea is to extract semantic summary of the native binary code, then convert common usage patterns of JNI interface functions into Java bytecode operations, and additionally transform native library function calls to bytecode calls. Along with this effort, we can empower the legacy Java static frameworks with the ability of inter-language data flow analysis without tampering their inherent logic. Extensive evaluation suggests that NativeSummary outperforms SOTA techniques in terms of accuracy, scalability and compatibility. NativeSummary sheds light on the promising direction of inter-language analysis, and thousands of existing app analysis works can be boosted atop NativeSummary with almost no effort.

CCS Concepts

• **Theory of computation** → **Program analysis**.

Keywords

Android, Static Analysis, Mobile Security

ACM Reference Format:
Jikai Wang and Haoyu Wang. 2024. NativeSummary: Summarizing Native Binary Code for Inter-language Static Analysis of Android Apps. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680335

*Corresponding author. The full name of the authors' affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST.

1 Introduction

Android app analysis is a long-lasting hot topic in our research community. Thousands of research studies were focused on Android apps, including malware detection [34], privacy analysis [12] and vulnerability assessment [28], etc. Static analysis is a widely used method [23], and a number of popular frameworks were developed, including FlowDroid [7], IccTA [21], DroidSafe [16], AmanDroid [42], etc. These tools leverage existing Java static analysis techniques to tackle `Dex` bytecode in Android apps, usually taking advantage of `SOOT` framework [37] and its intermediate representation (IR) called `Jimple`. Beyond `Dex` bytecode, developers can use Native Development Kit (`NDK`) to interact with native code, usually written in C/C++, through the Java Native Interface (`JNI`). Recent studies suggest that native code is prevalent in over 60% of Android apps, and it is even more favored by malicious apps [30].

However, the sad fact is that most existing Android app studies overlooked native code, although it poses a great impact on the analysis result. For example, without considering native code, sensitive information flows of Android apps might be incomplete, thus causing false negatives in privacy-related studies. Our community has witnessed the challenge, and several fellow researchers have proposed techniques to handle native code. As the first approach, `JN-SAF` [41] proposed a summary-based bottom-up data flow analysis to handle inter-language data flows. It designed an annotation-based analysis (based on `Angr` [32]) to uncover data flow in native code. `Jucify` [30] uses symbolic execution to extract native-to-bytecode invocations for constructing a unified call graph. Further, μDep [33] views native code as a black box and uses fuzzing techniques to infer the data flow relationship.

Although existing attempts show the promising direction of combining native code and `Dex` bytecode for a unified static analysis, they, however, pose great limitations when applied to real-world scenarios. First, techniques based on symbolic execution suffer from the path explosion problem when analyzing complex real-world Android apps, which inherit scalability issues. Second, using the call graph as a unified model is unable to represent how data is passed through arguments and returned values. There are some widely used operations (e.g., object creation, field access) in Android apps that can directly pass data from native to bytecode without an invocation. *In a nutshell, existing techniques on Android native code analysis suffer from either scalability or accuracy issues.*

This Work. In this paper, we propose NativeSummary, a novel inter-language static analysis framework of Android apps with high *scalability*, *accuracy* and *compatibility*. Our key idea is to use a dedicated binary analyzer to extract semantic summaries of the native code, then convert common usage patterns of JNI interface functions into corresponding Java bytecode operations, and additionally lift native library functions calls to bytecode calls. Built upon this, we can uncover the complete inter-language data flows. For *scalability*, we utilized a fine-tuned abstract interpretation technique in binary analysis, achieving high memory and time efficiency. For *accuracy*, we proposed a novel selective code summarization technique that extracts dataflow summaries from native code to bytecode, which not only makes the call graph more complete but also accurately reflects the dataflows. To achieve *compatibility*, we transform the dataflow summaries into bytecode method bodies and repackage them to APKs. In this way, NativeSummary is compatible with all legacy bytecode-level Android static analysis frameworks, empowering them with the ability of multilingual static analysis *transparently*. Extensive evaluation suggests that NativeSummary outperforms state-of-the-art techniques in terms of accuracy, scalability, and efficiency.

We make the following main research contributions:

- We propose NativeSummary, a novel inter-language Android static analysis framework with high *scalability*, *accuracy* and *compatibility*, based on key techniques including abstract interpretation and selective code summarization.
- We observe that many JNI usage patterns from real-world apps were not covered by existing works, thus we contributed an extended native data flow benchmark.
- We have performed extensive in-the-lab and in-the-wild studies to show the effectiveness and efficiency of NativeSummary. The number of native-to-bytecode call edges and native-related data flows NativeSummary uncovered on real-world apps is two orders of magnitude higher than state-of-the-art. Besides, NativeSummary is much more memory efficient and time efficient than SOTA tools.

2 Background and Related Work

2.1 DEX Bytecode, Native Code, and JNI

Java and Kotlin are the two major programming languages in the Android app development community. The code can be compiled to Java bytecode and then to Dalvik bytecode (DEX). Beyond DEX bytecode, in June 2009, the Android Native Development Kit (NDK) is released. By using the Java Native Interface (JNI), it allows developers to implement Android app using C/C++ or assembly. JNI is a foreign function interface that allows bidirectional communication between Java and C/C++. JNI maps each Java native method declared with the native keyword, to an assembly export function using a special name mangling scheme [18]. Besides, developers are allowed to use `RegisterNatives` to register the mappings without using a specific function name.

2.2 Android Native Code Analysis

Several studies attempted to analyze Android native code. JN-SAF [41] is the first framework that can track dataflows in native code. By extending Amandroid [42], it uses a summary-based bottom-up dataflow analysis for both Java bytecode and native code. Jucify [30] proposed to create a unified inter-language call graph. However, for precise multilingual analysis, the call graph is only a part of the story. Inter-procedural dataflow relationships and some commonly used operations (e.g., object creation, field access) should not be ignored. For binary analysis, JN-SAF implements an annotation-based analysis based on Angr's CFG analysis. Jucify uses symbolic execution (also based on Angr [32]) for binary code analysis, which is shown to suffer from scalability issues.

μDep [33] uses a fuzzing-based approach to infer dataflows in native code. It provides different parameters to each native method and observes the return value, to infer the dataflow relationships. Ruggia et al. [29] proposed ANDani, a native code analysis framework that builds an inter-procedural control flow graph for analyzing Android malware. Lee et al. [20] proposed a source-level static analysis framework for JNI. They use a guest analyzer (C/C++ analyzer) to extract semantic summaries for the host analyzer (bytecode analyzer). A further work [27] extends their framework to binary code via decompilation. They directly expose JNI functions as opaque Java method invocations, so the modification of the existing static analysis framework is still needed.

There are also some attempts to adapt the formal model for multilingual analysis. ILEA [36] can automatically extract models of C code by extending Java Virtual Machine Language (JVML). JNI Light [35] presented a formal operational model for JNI. They adopted a novel memory model to represent both native memory and Java memory, which allows memory to be shared. Some researchers [43] tried to extract library function models from documents, but this is not applicable if it is difficult to obtain the documents.

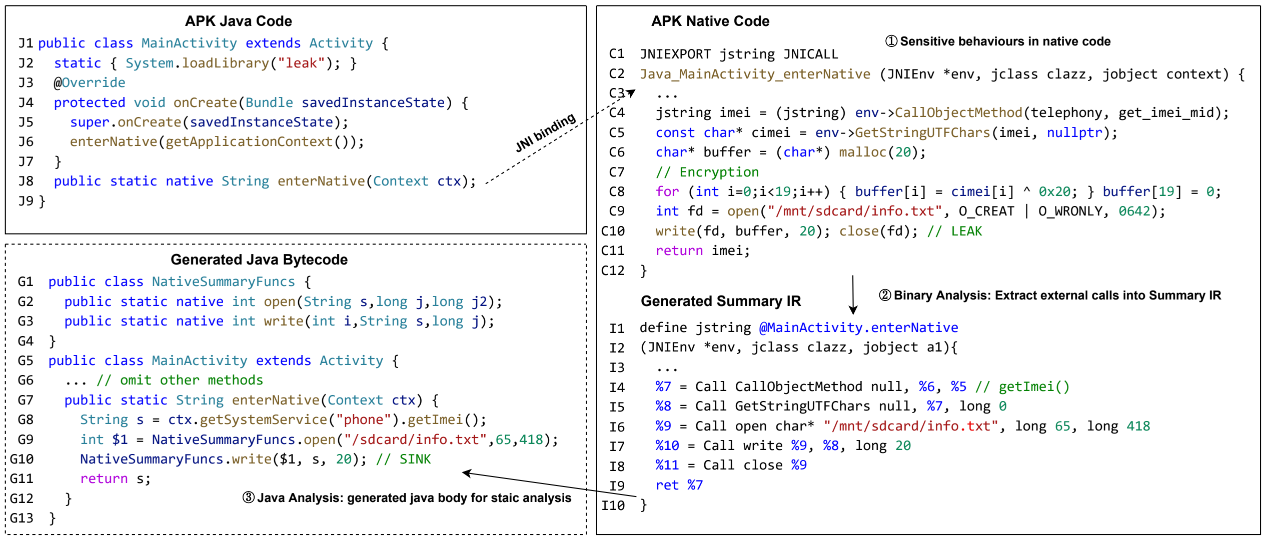
2.3 Binary Code Analysis

Analyzing binary code is considerably more challenging than analyzing DEX bytecode, and the challenges involved are entirely different [31]. Symbolic execution is a popular technique that can be directly applied to binary code, using existing mature tools like Angr [32]. However, it usually suffers from the "path explosion problem", where the number of paths grows exponentially with each branch and quickly becomes intractable [32].

Abstract Interpretation is a static analysis technique for sound over-approximation of all possible runtime states of a program. In abstract interpretation, an abstract domain is selected, and abstract values in that domain are used to over-approximate all possible concrete values in concrete execution. Unlike symbolic execution, which tries to solve constraints to check if each path is feasible, abstract interpretation usually regards all paths in a function are possible. With the development of abstract interpretation techniques, some analyses can scale to millions of lines of code [17]. Besides, by tuning analysis parameters like call-site-sensitivity levels, one can make a balance between precision and scalability.

Value-Set Analysis (VSA) [8] is a well-known attempt that is designed for binary analysis. VSA is a combined analysis of pointer analysis and numeric analysis based on abstract interpretation. Information about numeric values can improve the tracking of pointers, and pointer information can help track numeric values, leading to mutual promotion.

In this work, our binary analysis is based on BinAbsInspector [2], an advanced static binary code analyzer released by Tencent Keen Lab. Similar to VSA, it features an analysis engine that performs abstract interpretation on Ghidra's P-Code IR. Like VSA, BinAbsInspector also divides memory into different regions (global, stack and heap) and attaches a region label to each abstract value. One


APK Java Code
JNI binding
APK Native Code
① Sensitive behaviours in native code
Generated Summary IR
② Binary Analysis: Extract external calls into Summary IR
Generated Java Bytecode
③ Java Analysis: generated java body for staic analysis

```
J1 public class MainActivity extends Activity {
J2   static { System.loadLibrary("leak"); }
J3   @Override
J4   protected void onCreate(Bundle savedInstanceState) {
J5     super.onCreate(savedInstanceState);
J6     enterNative(getApplicationContext());
J7   }
J8   public static native String enterNative(Context ctx);
J9 }
```

```
C1   JNIEXPORT jstring JNICALL
C2   Java_MainActivity_enterNative (JNIEnv *env, jclass clazz, jobject context) {
C3     ...
C4     jstring imei = (jstring) env->CallObjectMethod(telephony, get_imei_mid);
C5     const char* cimei = env->GetStringUTFChars(imei, nullptr);
C6     char* buffer = (char*) malloc(20);
C7     // Encryption
C8     for (int i=0;i<19;i++) { buffer[i] = cimei[i] ^ 0x20; } buffer[19] = 0;
C9     int fd = open("/mnt/sdcard/info.txt", O_CREAT | O_WRONLY, 0642);
C10    write(fd, buffer, 20); close(fd); // LEAK
C11    return imei;
C12  }
```

```
I1   define jstring @MainActivity.enterNative
I2   (JNIEnv *env, jclass clazz, jobject a1){
I3     ...
I4     %7 = Call CallObjectMethod null, %6, %5 // getImei()
I5     %8 = Call GetStringUTFChars null, %7, long 0
I6     %9 = Call open char* "/mnt/sdcard/info.txt", long 65, long 418
I7     %10 = Call write %9, %8, long 20
I8     %11 = Call close %9
I9     ret %7
I10  }
```

```
G1   public class NativeSummaryFuncs {
G2     public static native int open(String s,long j,long j2);
G3     public static native int write(int i,String s,long j);
G4   }
G5   public class MainActivity extends Activity {
G6     ... // omit other methods
G7     public static String enterNative(Context ctx) {
G8       String s = ctx.getSystemService("phone").getImei();
G9       int $1 = NativeSummaryFuncs.open("/sdcard/info.txt",65,418);
G10      NativeSummaryFuncs.write($1, s, 20); // SINK
G11      return s;
G12    }
G13  }
```

Figure 1: The Java code and native code of the example app, the generated Summary IR and the generated Java body from native code.

major difference is that, it uses a simpler abstract domain called *K-Set* (instead of *strided interval* in VSA), that abstracts each location with a set containing a maximum of k possible values, or a special value $\top$ that represents any value.

3 Motivating Example and Key Ideas

We next present a motivating example to underscore the limitations of conventional Android static analysis in addressing native code.

Fig. 1 shows an example that the sensitive behavior is hidden in native methods. Traditional static analyses treat native methods like `enterNative` at J8 merely as black boxes, leading to a failure in flagging sensitive data flows during analysis. Yet, in the realm of native code, sensitive operations occur — the phone's IMEI is accessed at `C4`-`C5`, superficially "encrypted" via a simple xor operation at `C8`, and then stored in a file (indicative of a data leak) at `C10`. This oversight results in false negatives in traditional static taint analysis, especially for malware which often exploits native code to conceal its malicious payload and avoid detection [30].

3.1 Key Ideas

To address the significant oversight in traditional Android static analysis — its exclusion of native code — we have developed a method for converting native code into DEX bytecode. Such a conversion ensures seamless integration with current static analysis frameworks, which primarily function at the DEX bytecode level, thus enabling inter-language static analysis for Android apps.

Prior studies have explored the translation from C/C++ to Java [11]. However, because of significant semantic differences, they convert C/C++ pointers into a complex usage of Java arrays, posing considerable challenges for subsequent bytecode analysis.

Our approach refrains from a comprehensive translation from native code to DEX bytecode. Instead, for each native method, we create a *summary method* within the Java world. Within this summary method, we only translate the interfaces connecting the native method with the outside world, and the dataflows concerning these interfaces' inputs and outputs. This decision is driven by the understanding that *any sensitive behavior triggered by the native code must occur through these interfaces and their parameters.* These interfaces fall into two categories: JNI interface calls and C library calls. JNI interfaces enable native code to interact with the Java world in a way akin to Java reflection, facilitating operations such as invoking Java methods and modifying object fields. In the case of C library calls, it is crucial to also translate them into DEX bytecode calls, because they may also lead to sensitive behaviors such as invoking Android NDK functions for sensitive information.

When summarizing the native method, our translation is twofold. Firstly, for handling interfaces, we translate JNI interface calls by converting the usage of JNI functions into corresponding DEX bytecode operations. For C library calls, our strategy includes identifying equivalent Java APIs or creating stub native methods with the same name, and then translating these C library calls into corresponding DEX bytecode calls. Secondly, for dataflows concerning the interfaces' inputs and outputs, we employ a dedicated binary analyzer to identify these dataflows within the assembly code. We then translate the dataflows to DEX bytecode by directly linking the inputs and outputs among the translated interfaces using assignments and type conversions (detailed in § 4.4).

3.2 Key steps

We next present how NativeSummary analyzes the example app.

(*i*) Binary Analysis

First, we need to perform JNI binding resolution to discover entry points for our analysis. This process discovers, when a native method is invoked, which C or assembly code it will jump to. For example, the `enterNative` method at J8 is bonded to the C export function at `C2` according to the JNI name mangling scheme. The resolution process is further detailed in § 4.3.

Second, we need to prepare JNI-related structures. JNI interface functions are available to native code through structures (e.g., `JNIEnv` at `C2`) containing many function pointers. Thus, we need to create these functions and structures in memory.

Third, abstract interpretation analysis is used to track the dataflow. To capture all possible dataflows, when the native method interacts with the outside through interfaces, according to the required data type, we create a tainted stub value for method arguments (`clazz` and `context` at `C2`), and API return values (e.g., `imei` at `C4`). These special values are propagated during the abstract interpretation.

Finally, we export those interface calls and dataflows among them into *Summary IR* (§ 4.2). During the abstract interpretation, we maintain a list of abstract interface calls. After the analysis, for each call, we query for the possible values of each argument. If the set contains a stub object created previously, then a data flow is discovered. The generated Summary IR for the example app is shown in `I1-I10`. For example, at `C10`, the abstract value of `buffer` is a pointer to heap buffer, and the buffer contains tainted values previously created for `imei` at `C5`. Thus, at `I7`, the second argument directly refers to the value created at `I5`.

***(ii)* Java Analysis**: This module takes the *Summary IR* as input and outputs an APK with the generated method bodies.

First, it performs some preparation analyses. We first resolve JNI ID related APIs, and mark each IR value with the corresponding target (class, method, or field). Due to space limitations, these APIs are omitted in the example code.

Next, it performs Java type inference to assign a Java type for each IR value. We start with values that have a known type, then we propagate the type according to the dataflow. For example, as the target of the `CallObjectMethod` at `I4` is known, and its return type is `String`, we assign value `%7` with the type `String`.

Then, it transforms the Summary IR into Java bytecode. We need to handle JNI interface calls and external library calls. For JNI interface calls, we convert them to corresponding bytecode operations. For example, the `CallObjectMethod` at `C16` can be converted into a Java method invocation (at `G8`). For external calls, we first create corresponding stub methods, and convert them to bytecode calls to its methods. The stub method's signature is inferred by the previous type inference step. For example, the signature of `write` at `G3` is inferred from its usage at `I7`. Because the second argument of `write` used a value with type `String`, in the signature, the type of the second argument is typed as `String`. Then, we create a bytecode call at `G10` to the stub method.

Finally, the generated Java bytecode is repackaged with the original APK file, which can be fed into existing static analysis tools (e.g., FlowDroid [7]). In this example, the taint source is exposed by transforming the `CallObjectMethod` into a corresponding Java call, the dataflow is linked by binary analysis, and the taint sink `write` is exposed by lifting native library functions to bytecode-side static native methods. In this way, we can enhance the existing taint analyzer to successfully detect the inter-language `IMEI` leakage.

4 Design of NativeSummary

4.1 Overview

Fig. 2 shows the overall architecture of NativeSummary, which consists of two main modules: *Binary Analysis Module* and *Java Analysis Module*. *Binary Analysis Module* performs abstract interpretation over native binary code and outputs *Summary IR*. *Java Analysis Module* transforms related JNI interface function calls to corresponding bytecode operations, lifts these external library calls to bytecode calls, and finally repackages the generated bytecode with the original app into a new APK file.

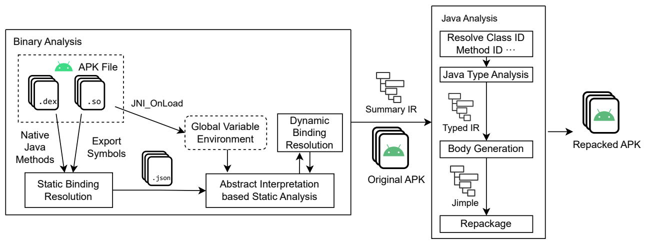


Figure 2: Overview of NativeSummary.

4.2 Summary IR

To convey the dataflows discovered by the *Binary Analysis*, and serve as the basis for the *Java Analysis*, we defined *Summary IR*. There are three types of instructions in *Summary IR*: *Call*, *Ret*, and *Phi*. Interesting function calls are lifted to *Call* instructions, which include JNI interface function calls and external library calls. A *Ret* instruction marks the return value of the analyzed native method. *Phi* instructions are detailed later. Instructions operand can be another instruction or a simple constant.

There are two important differences between the *Summary IR* and a typical compiler IR. First, we only create one IR function for each registered native method. During binary analysis, the analyzed assembly function may call other assembly functions, but all these behaviors are not sensible for the Java world. Besides, having everything inside one method body allows us to freely represent the dataflow relations discovered by the binary analyzer, regardless of complex C features like pointers. Second, *Summary IR* has no control flow. Instructions directly belong to the function. Because abstract interpretation traverses over all concrete executions, and each abstract value is a set of values that over-approximates all possible values it can be. If there are multiple values in the abstract value, we use *Phi* instructions (inspired by [20]) to merge them.

4.3 Binary Analysis

The *Binary Analysis Module* consists of three main steps: (*i*) static binding resolution, (*ii*) dynamic binding resolution and (*iii*) abstract interpretation based static analysis.

4.3.1 Static Binding Resolution. As shown in Fig. 3, the JNI name mangling scheme specifies how a Java-side native method is mapped to a C export function name. During development, a developer can use the `javah` utility that comes with JDK to generate header files that contain C function declarations. When JVM loads the native object, it will find the corresponding Java-side native method and register the mapping. We refer to this process as static binding. To build a robust resolution for static binding, we strictly follow the specification and carefully consider error-prone cases. One common mistake is to forget the escape sequence. In Fig. 3, `"_1"` should be converted into a `"_"` in class name. Another possible mistake is forgetting to handle the signature part, although the function signature part will usually be omitted when there is no overloading.

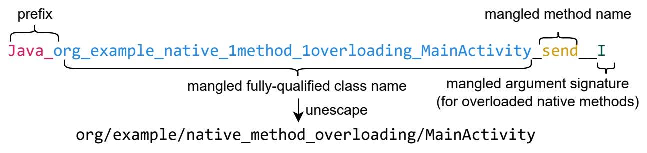

prefix
mangled method name
Java_org_example_native_1method_1overloading_MainActivity_send__I
mangled fully-qualified class name
mangled argument signature
(for overloaded native methods)
unescape
org/example/native_method_overloading/MainActivity

Figure 3: The JNI name mangling scheme [19].

```
jint JNI_OnLoad(JavaVM *vm, void *reserved) {
  env = ...; // omit version check and GetEnv call
  clazz = env->FindClass(className);
  env->RegisterNatives(clazz, gMethods, numMethods);
}
static JNINativeMethod gMethods[] = {
 {"send",    "(Ljava/lang/String;)V",  (void *) native_send},
 {"sendFoo", "(ILjava/lang/String;)V", (void *) native_sendFoo},
 {"sendBar", "(DLjava/lang/String;)V", (void *) native_sendBar},
};
```

Listing 1: Simplified code of JNI dynamic registration.

4.3.2 *Dynamic Binding Resolution.*

A simplified JNI dynamic registration process is shown in Listing 1. The developer can call RegisterNatives with a structure containing dynamic registration information. Usually, the structure is defined as a global variable, so it will be stored in the data segment of the binary.

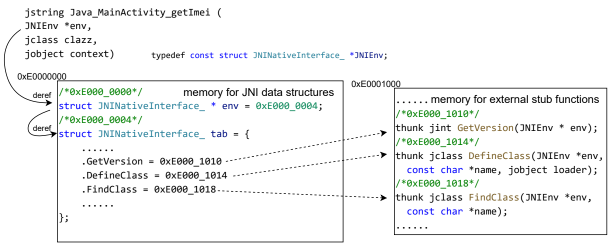
The dynamic registration behavior is discovered by our external function model of RegisterNatives during our binary analysis. Besides statically registered functions, we also apply the binary analysis (detailed in §4.3.3) to JNI_OnLoad and to discover RegisterNatives calls. Then, we obtain possible addresses of the structure by querying the analysis engine for the possible argument values of the RegisterNatives function. By parsing the structure according to its definition, we can get the bindings and add them to the work list of registered native methods. By parsing the provided Java method's signatures, we can set up C function signatures of these newly discovered JNI functions according to the specification.

4.3.3 *Abstract Interpretation based Static Analysis.*

Based on BinAbsInspector [2], we apply a k-call-site sensitive abstract interpretation over binary code. We translate important calls into *Summary IR* according to the k-call-site-sensitive abstraction of our analysis. To be specific, for each native method, we convert each important call with the same call-site and the same k callers' call-sites on the call stack into one bytecode operation. Here we face several challenges: (*i*) how to adapt the static analysis for the JNI environment; (*ii*) how to track different types of dataflows during the use of JNI interface functions; (*iii*) how to further improve the precision when applying abstraction interpretation over assembly. We next depict our solutions for each challenge, respectively.

(*i*) JNI Environment Preparation: To make the abstract interpretation based analysis capable of analyzing JNI codes, we need to set up the JNI-related structures. As shown in Fig. 4, JNI functions are available through function pointers in JNI interface structures (e.g., JNIEnv and JavaVM). To invoke a JNI interface function, we first get the function pointer by accessing the table using the JNIEnv, and then invoke the function pointer.

An example memory mapping for 32-bit ARM is shown in Fig. 4. We map two memory blocks for external stub functions and JNI interface structures, which include JNIEnv and JavaVM. Then we


0xE0000000
deref
deref
memory for JNI data structures
0xE0001000
memory for external stub functions

```
jstring Java_MainActivity_getImei (
JNIEnv *env,
jclass clazz,
jobject context)
```

```
typedef const struct JNINativeInterface_ *JNIEnv;
```

```
/*0xE000_0000*/
struct JNINativeInterface_ * env = 0xE000_0004;
/*0xE000_0004*/
struct JNINativeInterface_ tab = {
    ......
    .GetVersion = 0xE000_1010
    .DefineClass = 0xE000_1014
    .FindClass = 0xE000_1018
    ......
};
```

```
......
/*0xE000_1010*/
thunk jint GetVersion(JNIEnv * env);
/*0xE000_1014*/
thunk jclass DefineClass(JNIEnv *env,
  const char *name, jobject loader);
/*0xE000_1018*/
thunk jclass FindClass(JNIEnv *env,
  const char *name);
......
```

Figure 4: Memory layout for JNI data structures and functions.

create an external thunk function for each JNI interface function and set up each function pointer in the structure to the corresponding address. In this way, when we set the first argument JNIEnv* env to the address of the structure pointer (i.e., JNINativeInterface_-*), the dereference of the pointer and the invocation of JNI interface functions will happen during the static binary analysis the same way as in concrete execution.

(*ii*) Dataflow Tracking: To track the dataflows among JNI calls, we create abstract values according to their types. Before analyzing a native function, we first create corresponding abstract values as its arguments. During the analysis, when our external function models get called, we record the call and create corresponding abstract values as the return value. After the analysis terminates, we can query the abstract value of a specific register at any program point. Calling convention constrains which register should be used to store arguments or return value during a function call, thus we can query the abstract values of each argument to check if it contains any abstract values we created previously for each external call. The return value is resolved in the same way. In this manner, we get complete data flow relationships for each native function.

To create corresponding abstract values for each API, we categorize dataflows during a JNI native method invocation as follows:

- **Opaque JNI Object**: According to the JNI specification, there are pointer-sized opaque types (e.g., jobject and jclass) that should be directly passed to other JNI interface functions without any modification. We leverage Ghidra's address space and allocate an address from a new address space as an opaque pointer to represent it.
- **Integer Type**: BinAbsInspector has a taint tracking mechanism over static analysis. Thus, for the integer type, we directly return a tainted $\top$, and the taint allows us to continue tracking dataflow.
- **Buffer Type**: Native code may call memory allocation functions to get a pointer to the allocated buffer (e.g., malloc). BinAbsInspector has a basic model for these functions, which allocates a new heap region and returns a pointer to the memory buffer within it. Some string-related JNI interfaces like GetStringUTFChars return a buffer that contains the string data. We model these cases in the same way.

Cached IDs: When using the JNI interface functions, there is a common programming pattern that caches method IDs, class IDs and field IDs to improve performance [10]. These IDs can be precomputed and cached to global variables, usually in JNI_OnLoad. During the development of the tool, we discovered multiple real-world usages of this pattern, e.g., in a popular graphics library `libgdx.so` [4]. This brings us a new challenge. To handle this pattern, we preserved the modification to global variables introduced by `JNI_OnLoad` after dynamic register resolution. When other native functions access related global variables, the abstract value stored in the corresponding location will be used. When resolving the abstract value, it will directly refer to the return value of related `Get...ID` call in the `JNI_OnLoad`.

***(iii)* Abstract Interpretation on Assembly**: Applying abstract interpretation to real-world apps directly would face some interprocedural challenges that will dramatically decrease the precision.

C1: Handling Callee Saved Registers. Although the stack grows linearly during concrete execution, in abstraction interpretation, two stack spaces of the same function on the call stack may refer to the same abstract function's local stack. Thus, each function's stack memory is separately stored and its stack pointer is independently maintained. The stack pointer will be switched to the callee's stack pointer during a function call. However, in BinAbsInspector, other registers and the whole memory are directly passed from callsite to the callee.

Callee saved registers are a set of registers defined by calling convention, and they are required to be preserved across a procedure call. But from the perspective of abstract interpretation, it is like that some local variables that are unrelated to the call, are saved on the callee's stack space during the call. This is equivalent to losing one level of call-site-sensitivity. This requires the abstract interpretation to have at least 1-call-site-sensitivity, or else the callee saved registers will get mixed and the analysis result will become extremely imprecise. However, in some simple functions that do not need many registers, the callee saved registers are left untouched, and will continue to be passed to its callees, and more levels of call-site-sensitivity of these values will be lost.

Our mitigation is to simply preserve all callee saved registers at each call site according to the calling convention. We mainly focus on the ARM architecture family and the calling convention for ARM is generally stable (AAPCS and its variants), so in most cases, our solution works well. In very few cases, when some AAPCS variants that support floating-point extensions are used, float-point parameters are passed through additional float-point registers. We did not consider this situation and this dataflow is not tracked.

C2: Passing the Stack Space. Values on the caller's stack space face a similar issue. To return additional return values, a C function will receive a pointer to the caller's stack variable as an argument and write to it. So we need to pass the local values to the callee during the call and propagate the values back when the call ends. However, passing too many functions' local variables on the call stack can cause potential precision loss if the values from different contexts are mixed during the call and passed back when the call ends. We modified BinAbsInspector so that only those local stack values of k-callers on the call stack are passed.

C3: Tail Call Optimization. Further, we also face the challenge of tail call optimization. A tail call appears as an ordinary jump at the end of the function, and is not recognized as a function call, but acts as if the callee's function body is a part of the caller's body. Especially, if the callee is an external function, the call will be missed by BinAbsInspector. We added some code in the processing of jump p-code in BinAbsInspector to handle this case.

Export Result: Finally, the whole analysis result will be converted into *Summary IR*. We maintained a list of external function calls, which include JNI interface functions and other native imported functions (e.g., C library functions). The list is ordered by the time we first reach its callsite during the analysis. After the analysis is completed, we convert these external calls into Summary IR's call instructions. The number of arguments is determined by the function signature or by querying the argument analysis in the decompiler. The actual registers that hold these arguments are obtained according to the calling convention. Then, we obtain the abstract values of the registers by querying the static analysis engine. Finally, the origins of these abstract values are resolved, and if there are multiple origins, we create a "phi" instruction to merge them. The function's return value is also resolved following the same process above, and finally, a Summary IR function is created.

4.4 Java Analysis

This module converts each function in *Summary IR* into a bytecode method body in a best-effort manner, including four steps.

***(i)* ID Resolution**: This step prepares for body generation. There are three kinds of JNI IDs: class ID, method ID, and field ID. Some JNI interfaces (e.g., `FindClass` and `GetMethodID`) return an ID according to the class name or method name string that passes in as an argument. In this step, these calls are resolved and related IR values are mapped to the actual class, method, or field.

***(ii)* Java Type Inference**: In this step, we infer Java type for each IR value and create signatures for native library functions. The native code in JNI functions may call existing C library functions. We convert each library function to a static native method in a special class called `NativeSummaryFuncs`, so we can convert external calls to corresponding Java calls later. Because functions in native code have no signature information, we need to infer the signature of each static method from its usage before creating related bytecode methods. First, values with a determined type (e.g., the arguments and the return value of a Java call operation with a known target) are tagged on IR. Then, the types are bidirectionally flowed on the def-use chain. Types of "phi" instruction operands are merged. If there is a direct conflict between types, we attempt to select a type that is sufficiently generic to represent these types and insert a type conversion. If such a type does not exist, or in cases where there is no type information available, we will choose the Java String type and insert `toString` calls.

***(iii)* Body Generation**: For JNI interface calls, some (listed in Table 2) are converted into Java operations. Native library calls, and other JNI interface calls that do not correspond to bytecode operations, are converted into Java calls to the previously created native methods. To convert "phi" instructions, we will create a local variable for the result and corresponding if-else-if opaque branches and assignments for each operand. So, after these branches, the local variable may be assigned to any values in each branch. `Throw` and `ThrowNew` operations need to be guarded within an opaque branch so that subsequent operations are not dead code.

***(iv)* APK Repacking**: Finally, we inject the created method bodies, remove the "native" modifier, and repackage the DEX into a new APK. We create the method body using the Soot [37] framework, however, Soot reports many errors while loading complex codes

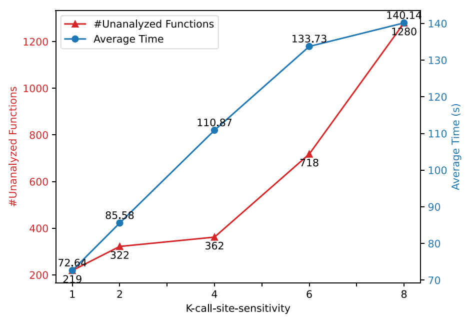


Figure 5: In 3,801 native methods from 100 F-Droid apps, with the increasing of k-call-site-sensitivity, (*i*) the average binary analysis time for each native method and (*ii*) the number of methods that failed to cover within 4 hours.

of real-world apps. To mitigate this issue, and achieve minimum modification to the original code, we first generate a list of modified classes along with the modified DEX file, and then repack the DEX file with the original DEX according to the list. Note that, the generated APK should only be used for static analysis, and we did not expect to run it dynamically.

5 Implementation and Evaluation

Implementation We implement NativeSummary from scratch in around 5*k* SLOC Java and 551 SLOC Python. The static binding resolution is implemented in Python. The Java code includes 2.9K SLOC for *Java Analysis*, and 2.0k SLOC for *Binary Analysis* which only includes our modification to BinAbsInspector [2].
Evaluation We evaluate the *accuracy*, *scalability*, and *efficiency* of NativeSummary, respectively, by answering the following RQs.
RQ1: Can NativeSummary accurately perform inter-language static analysis?
RQ2: Can NativeSummary assist taint analysis in native codes of real-world apps?
RQ3: How efficient is NativeSummary on real world apps?

5.1 Experiment Setup

Our experiments run on a server with two AMD Epyc 7713 CPUs and 256GB RAM, with Ubuntu 22.04.1. We limit the CPU performance to a single core by setting docker's `"--cpus=1"` flag so that the analysis time directly reflects the efficiency.

5.1.1 K-call-site-sensitivity Selection. Call-site sensitivity differentiates different instances of function according to its call context. Increasing the call-site sensitivity value can increase precision when analyzing deep function calls but decrease scalability. We randomly selected 100 open-source Android apps from the F-Droid [13] dataset to measure its impact, and there are 3,801 native methods in total. These apps were evaluated under different configurations of K, i.e., K = 1, 2, 4, 6, and 8. We set the timeout for each native function to 1000 seconds and the total analysis time for each app to 4 hours. We measure the impact of K from three aspects:

- ***The average analysis time*** for each native method rises greatly with the increase of K. Fig. 5 shows that it nearly doubled (140.14s compared with 72.64s) with the increase of K from 1 to 8.
- ***The number of unanalyzed functions*** increases dramatically from 219 (K=1) to 1280 (K=8), due to the timeout of 4 hours.
- ***The number of discovered native-to-java call edges*** decreases with the increase of K. The improvement of analysis precision brings a small number of new call edges (6 for K >= 2, 7 for K >= 4), but the total number of call edges still decreases (compared with K=1, -19 for K=2, -214 for K=4, -317 for K=6, -369 for K=8) because many native methods were not analyzed. Furthermore, for most native methods (3,024, 79.56%), the generated summary is identical from K=1 to K=8. We found that most native methods have only a small amount of reachable code and the function calls are not deep, so the results did not change with the variation of K. For the remaining functions, if K is increased, the analysis spends more time on complex functions such as encryption, compression, etc., leading to an increase in analysis time.

The selection of K: Since increasing the K value does not improve the quality of the summary, we will set the K value to 1 for our subsequent experiments.

5.2 Accuracy (RQ1)

5.2.1 SOTA Selection. We evaluate NativeSummary by comparing it with two most recent works, Jucify [30] and JN-SAF [41]. Note that NativeSummary has high compatibility with existing static analysis frameworks, we have enabled three widely used tools with NativeSummary for comparison, including FlowDroid [7], AppShark [1] and Amandroid [42] (now Argus-SAF). Further, to facilitate the replication of our study, we have also packaged each tool and released them as docker images, so that other researchers can easily run each tool with a single command for comparison.

5.2.2 Benchmarks. We collected the widely used benchmarks, including (*i*) `NativeFlowBench` from JN-SAF [41], and (*ii*) `JucifyBench` from Jucify [30]. Further, we found that these benchmarks did not cover real-world common usage patterns of native code, thus we further contribute a benchmark `NativeFlowExtended` by summarizing native code patterns from real-world apps.
(*i*) NativeFlowBench contains 23 apps. We exclude 4 native activity related apps, as it is reported that native activity (activity that is implemented fully in native code) is very rare in real-world apps [33]. Thus, we have 19 benchmark apps.
(*ii*) JucifyBench contains 11 hand-crafted apps. Jucify classifies native related data leaks into four categories, including 5 "Getter" apps with taint source in native code and taint sink in Java code. The other three categories, "Leaker" (taint source in Java code and sink in native code), "Proxy" (taint source in Java code and sink in Java code), "Delegation" (taint source in native code and sink in native code) each contains 2 apps. Note that in `JucifyBench`, there are no "real" native-specific sinks. The "leak in native" means, the bytecode side leak method is invoked in the native code.
(*iii*) NativeFlowExtended is an extended native code benchmark with 10 apps. Apps in `NativeFlowBench` and `JucifyBench` mainly focus on Java-side manipulation of native code. Actually, native methods of most test cases have exact translations of Java code, which is completely not the case in real-world apps. Based on

our investigation of real-world apps, we found many common JNI dataflow patterns that NativeFlowBench and JucifyBench did not cover, which are added in the benchmark by us. First, we created three benchmark cases that propagate sensitive dataflow using C data structure. Three cases use NIO and direct buffer related JNI interfaces. Two cases use native-specific library functions to leak data. And one case tests the support for caching IDs in global variables[10], one case uses the programming pattern that passes heap pointer as integer handle to the Java code. More detailed introduction can be found in the GitHub repo[5].

5.2.3 Evaluation Result. As shown in Table 1, NativeSummary achieves the best result, across all kinds of implementations. Specifically, `NS/FD` (NativeSummary + FlowDroid) performs the best, achieving 81.4% accuracy. `NS/AS` (NativeSummray + AppShark) achieves similar result. As AppShark does not support ICC taint flows, we skipped the app `icc_nativetojava`. JN-SAF and Jucify perform well in their own benchmarks, but they struggle to detect cases in other benchmarks.

Missing Cases of NativeSummary. In the case of `native_source_clean` (marked with *), IMEI is saved as a member of a data object, and before being printed (i.e., leaked), the corresponding member is reassigned with a constant string in the native method. We used `Jadx` [3], a widely used decompiler to manually investigate the generated method body in the repacked app, and the generated code correctly reflected the corresponding semantics, so we believe that this missing case is caused by the inaccuracy of the backend bytecode analysis frameworks. `native_noleak_array` uses Java array to pass data, and `native_complexdata_stringop` concatenates C string in native code. These features are difficult for static analysis, and only `NS/AM` can handle one case.

NativeFlowExtended In our crafted benchmark from real-world apps, the combination of NativeSummary and AppShark (`NS/AS`) performs the best. JN-SAF and Jucify cannot handle any of these common used patterns. In the case of `native_copy_strdup` and `native_array_elements`, we lift the calls to `strdup`, and `GetByteArrayElements` into Java calls. Flowdroid does not model native methods, while AppShark has a basic black box model that propagates dataflow from argument to the return value, so AppShark is able to detect them. For `native_array_region` and `native_handle`, the complex API usage patterns make the dataflow obscure. For `native_encode`, each source byte is used as the index of the `base64` encoding table, and we normally do not consider the dataflow from array index to the result, so this case is hard to analyze.

It is noticeable that not a single implementation can cover all cases that are correctly detected by at least one tool. This means that the performance of NativeSummary can be further improved if there is a better Java-side taint analysis tool in the future. Nevertheless, we have demonstrated that NativeSummary can empower existing Java static analysis frameworks with the ability to analyze native code. We choose AppShark and FlowDroid as the backend dataflow analysis tools for subsequent experiments.

Table 1: Evaluation result on benchmarks.

◯ = Correct, × = Wrong

App Name	NS+FD	NS+AS	NS+AM	JS	JU
NativeFlowBench (with leak)					
native_source	◯	◯	◯	◯	×
native_leak	◯	◯	◯	◯	×
native_leak_array	◯	◯	◯	◯	×
native_leak_dynamic_register	◯	◯	◯	◯	×
native_dynamic_register_multiple	◯	◯	◯	◯	×
native_multiple_interaction	◯	◯	◯	◯	×
native_multiple_libraries	◯	◯	◯	◯	×
native_complex_data	◯	◯	×	◯	×
native_heap_modify	◯	◯	◯	◯	×
native_set_field_from_native	◯◯	◯◯	××	◯◯	××
native_set_field_from_arg	◯◯	◯◯	◯◯	◯◯	××
native_set_field_from_arg_field	◯◯	◯◯	××	◯◯	××
icc_nativetojava	◯	-	◯	◯	×
native_method_overloading	◯	◯	◯	◯	×
NativeFlowBench (without leak)					
native_nosource	◯	◯	◯	◯	×
native_source_clean*	×	×	×	◯	×
native_noleak	◯	◯	◯	◯	×
native_noleak_array	×	×	×	×	×
native_complexdata_stringop	×	×	◯	×	×
JucifyBench (with leak)					
getter_imei	◯	◯	◯	×	◯
leaker_imei	◯	◯	◯	×	◯
proxy_imei	◯	◯	◯	×	◯
delegation_imei	◯	×	◯	×	◯
proxy_double	◯	◯	◯	×	◯
delegation_proxy	◯	×	◯	×	◯
getter_leaker	◯	◯	◯	×	◯
getter_proxy_leaker	◯	◯	◯	×	◯
getter_imei_deep	◯	◯	◯	×	◯
JucifyBench (without leak)					
getter_string	◯	◯	◯	◯	×
leaker_string	◯	◯	◯	◯	×
NativeFlowExtended					
native_copy	◯	◯	×	×	×
native_copy_strdup	×	◯	◯	×	×
native_encode	×	×	×	×	×
native_file_leak	◯	◯	◯	×	×
native_socket_leak	◯	◯	◯	×	×
native_global_id	◯	◯	◯	×	×
native_handle	×	×	×	×	×
native_array_region	×	×	×	×	×
native_array_elements	×	◯	×	×	×
native_direct_buffer	◯	◯	×	×	×
Sum and Precision					
◯, higher is better	35	34	30	22	9
×, lower is better	8	8	13	21	34
Percentage p = ◯/(◯+×)	81.4%	81.0%	69.8%	51.2%	20.9%

FD = FlowDroid, AS = AppShark, AM = Amandroid, JS = JN-SAF, JU = Jucify, NS = NativeSummary. Some apps contain two leaks.

5.2.4 JNI Interface Function Support. One aspect that can reflect the functional completeness of each tool is the support for JNI interface functions. By inspecting each tool's source code, we summarized their supported JNI interface functions, as shown in Table 2. Beyond method invocation, Jucify can hardly support other functions. As a comparison, our tool supports all the functions.

Answer to RQ1: NativeSummary outperforms SOTA techniques on existing native code benchmarks. It can uncover many common JNI usage patterns which are overlooked by existing efforts, and handle the most number of JNI interface functions.

Table 2: A comparison of supported JNI interface functions.

	Register Natives	Call<Ty> Method	Set/Get <Ty>Field	GetString [UTF]Chars	NewString [UTF]	NewObject	Throw[New]	Get/SetObject ArrayElement	GetObject Class
NativeSummary	O	O	O	O	O	O	O	O	O
JN-SAF	O	O	O	O	O	O	X	O	O
Jucify	O	O	O	X	X	X	X	X	O

Square brackets mean optional. For example, Throw[New] stands for two APIs: Throw and ThrowNew.

Table 3: Real-world app datasets for evaluation.

Dataset	# Apps	# Apps with Native Code	Description
F-Droid [13]	4,023	680	Open source Android apps from f-droid.org
Malradar [40]	4,534	1,575	Malradar malware dataset

Table 4: Results on real-world apps.

Tools	#Total Apps	#Success App	#Total Flows	#Native Methods Analyzed	#Native Methods Succeeded	#Native Edges Created	#Native Related Flows
				In 271 Apps (Intersection of Successful Apps) / In All Apps			
NS+FD	2255	1360	1203775	8521 / 60227	6690 / 44993	247 / 7111	187 / 2239
NS+AS		1567	2483215				425 / 4572
JN-SAF		744	3832	43 / 212	42/177	5 / 85	6 / 26
Jucify		1111	718687	850 / 20462	0 / 2303	0 / 130	2 / 11

FD = FlowDroid, AS = AppShark, NS = NativeSummary

5.3 Scalability (RQ2)

Dataset: We use two real-world datasets of Android apps, as presented in Table 3, including the widely used F-Droid [13] open source app dataset (accessed on 2023-01-18), and MalRadar [40], a most recent Android malware dataset.

Dataset Preprocessing: F-Droid dataset maintains different versions of the same app. After deduplication, There are 4,023 apps left. To identify apps that contain native code, we only choose APKs that contain at least one shared object (.so file) and at least one Java method with `native` modifier. As shown in Table 3, there are 680 apps left in F-Droid and 1,575 apps left in MalRadar.

5.3.1 Experimental Setup. Based on the configuration determined in § 5.1, we limit the maximum memory to 32G by setting docker's `"--memory=32G"` flag. We set a timeout of 2 hours for each app.

Sources and Sinks Selection: Taint sources and sinks have a huge impact on taint analysis [24]. To maximize detected flows, we used sources and sinks from TaintBench [24] (`merged_sources.txt` and `merged_sinks.txt`), which are collected from sources and sinks of various existing Java taint analysis tools, containing about 20k sources and about 10k sinks. Then we convert it to different formats that each tool will accept, and adapt each tool to use it.

Adaption of existing tools: When we evaluate `JN-SAF` on real-world apps, most of the apps (91.32% on F-Droid dataset, 88.51% on Malradar dataset) reported an error saying "loadBinary can not finish within 1 minutes". We manually checked the source code of `JN-SAF` and found that `JN-SAF` hard-coded a 1-minute timeout when decompiling the DEX bytecode. We removed this timeout for fair comparison. `Jucify` invokes `FlowDroid` internally to perform taint analysis. By default, `Jucify` only prints whether there is any flow through native without detailed information, thus we revise it to make it output the flows for comparison.

5.3.2 Result. The result is shown in Table 4. For each tool, we first count the number of apps that can be successfully analyzed, i.e., with valid taint analysis results. Then we count the total number of flows discovered by each tool (`#Total Flows`). To further investigate the performance of binary analysis, we counted the number of native methods being analyzed and succeeded (`#Native Method Analyzed` and `#Native Method Succeeded`) and the number of native-to-java edges created (`#Native Edge Created`) during the process. Finally, by enabling the reconstruction of the taint path, we are able to count `#Native Related Flows`, by searching for native method in the path elements. 271 apps can be successfully analyzed by all these tools, and for a fair comparison, we further show the statistics in these 271 apps separately in Table 4. We next explain the result of each tool.

JN-SAF: JN-SAF can only analyze 744 apps (33%) successfully, and the total number of flows discovered (3,832) is significantly lower than other tools. We found that it is due to the limitation of the backend analysis framework (i.e., Amandroid). As observed by existing studies, Amandroid has limitations in discovering information flow [33, 44] and the number of flows reported by it varies greatly each time it is run independently in the same setup [44]. Because JN-SAF uses an on-demand binary analysis, the number of native methods it can be analyzed is also impacted.

Jucify: Jucify analyzed many native methods (20,462), but the number of native methods that were successfully analyzed is quite low (0 / 2303). After inspecting its source code, we observe that Jucify hard coded a 1,800s timeout in its code for the whole binary analysis. Thus, Jucify's binary analysis has successfully analyzed some native methods internally, but due to the timeout mechanism, these intermediate results are not delivered to the taint analysis. By analyzing its intermediate result, we observe that the actual number of successful native methods is 790 / 16,076. Due to the complexity of the changes involved, we did not make further modifications to it.

NativeSummary: On real-world apps, NativeSummary can still boost different backend data flow analysis tools (FlowDroid and AppShark). As shown in Table 4, NativeSummary can successfully analyze the most number of apps (1,360 / 1,567), and can discover and analyze 3 times more native methods (60,227) than Jucify (20,462) in the whole dataset. Further, our tool can successfully analyze significantly more native methods (44,993), much more than JN-SAF and Jucify, and more than Jucify's hidden number (790 / 16076). Finally, only NativeSummary is capable of finding a non-trivial amount of native-related flows (2,239 / 4,572 in all apps), two orders of magnitude higher than JN-SAF and Jucify.

It is worth mentioning that the discovered new call edges (7,111) consist of 5,217 normal calls (via `Call<ty>Method`, deduplicated by its target) and 1,894 constructor calls due to the support for

`NewObject` and `ThrowNew`. Even if we only consider normal call edges (for fair comparison), the number of calls discovered by NativeSummary is still much more than other tools.

5.3.3 *Further Verification.* We further validate our results from two aspects: call edges and dataflows.

Call Edges: We randomly selected 40 Android apps (20 from the F-Droid and 20 from MalRadar), to check whether the discovered java-to-native bindings and native-to-java edges created are correct. We relied on Jadx [3] to decompile bytecode and Ghidra [25] to decompile assembly. For Java-to-native edges, we are able to confirm the binding is correct, by either checking the binary export symbol or by finding related `RegisterNatives` call and checking the structure pointer passed in. For native-to-Java edges, by checking the log printed by NativeSummary, we can determine the call-site address of the calls to related JNI interface functions, and we check related arguments passed in to ensure the call edges are correct (most of the time, to find a class or method, a string literal is directly passed to FindClass or GetMethodID).

Dataflows: We randomly selected 50 data flows (25 from FlowDroid and 25 from AppShark) from the discovered native-related flows to examine the quality of our semantic summaries. These flows are chosen from the F-Droid open-source app dataset so that we can search for relevant source code to verify the correctness of the data flows. The artifact of this process [6] is available with our tool.

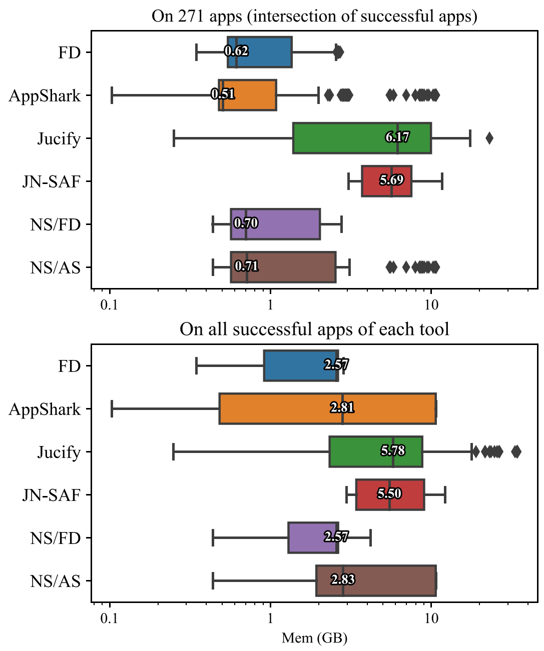
We found that 48 (96%) of the discovered dataflows are correct. Our tool can accurately connect various dataflows in native code, such as logging (21), file operations (14), and computations (4). The behavior of throwing Java exceptions in native code is also correctly reflected in the generated method body. For the two false positives, we identified two reasons: (*i*) the abstract interpretation overestimated the data flow because of complex native computations. (*ii*) the analysis overestimated the argument count of functions with variable arguments (e.g., __android_log_print). Because, if the format string argument is not a constant string, we will over-estimate the argument count to capture potential dataflows.

Answer to RQ2: Extensive evaluation suggests that NativeSummary is the only tool that is ready for analyzing real-world apps. NativeSummary can analyze three times more native methods than the second best-performed tool, and the number of native-to-bytecode call edges and native-related information flows it uncovered is two orders of magnitude higher than SOTA.

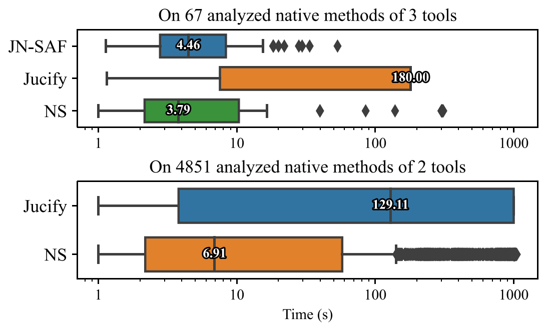
5.4 Efficiency (RQ3)

5.4.1 *Memory Efficiency.* We show the memory consumption on all successful apps that each tool can analyze (Fig. 6a bottom). Further, as shown in RQ2, there are only 271 apps that can be successfully analyzed by all three tools, which will be used for a fair comparison. We also run FlowDroid and AppShark (i.e., without native analysis) under the same setup as NS+FD, as a baseline for comparison.

Result. NativeSummary is much more memory-efficient than Jucify and JN-SAF. Of the 271 apps that can be successfully analyzed by all of the tools, NativeSummary with FlowDroid costs only 1.22 GB memory on average (median 0.70 GB), which is roughly 6X less than JN-SAF (6.09 GB on average and median 5.69 GB) and Jucify (5.89 GB on average and median 6.17 GB). For the whole dataset, the median value (2.57 GB) and average (2.08 GB) of NativeSummary are close to that of FlowDroid (median 2.57 GB, avg 1.96 GB), which are much lower than those of JN-SAF (median 5.50 GB, avg 6.24 GB) and Jucify (median 5.78 GB, avg 6.13 GB). Note that, NativeSummary can analyze much more apps successfully than other tools. When combined with AppShark, NativeSummary can discover more dataflows and edges (in Table 4) while consuming more memory.



(a) Memory consumption (in GB).



(b) Per native method time (in seconds).

Figure 6: Memory statistics (a) of the four tools during the analysis, measured on 271 apps that all tools can succeed (top), and on all apps each tool can successfully analyze (bottom). And native method analysis time (b) on the same set of analyzed native methods.

5.4.2 *Time Efficiency.* JN-SAF and Jucify also launch Angr's analysis starting from each native method as the entry point and we modified each tool to print the time. Fig. 6b shows the distribution of the binary analysis time of each tool on the same set of native methods (the binary loading and initial analysis time are not included). To better show the scalability of each tool and focus on

Table 5: Average time consumption of each step.

Native Summary Repackage 1035.0s				FlowDroid 329.5s
Static Binding Resolution 54.3s	Ghidra Analysis 973.7s		Java Analysis 7.0s	
	Ghidra Load Binary 448.6s	BinAbsInspector Solver 525.0s		

complex native methods, we only consider native methods that all tools spent at least 1s, so many tiny native methods are filtered. Because JN-SAF is unable to analyze many native methods on the dataset due to its on-demand analysis (as shown in RQ2), only 67 native methods are used for comparison in this setting. As shown in Fig. 6b, Jucify spends much more time than other tools, and more than half of the methods analyzed by Jucify have reached the 180s timeout (59.7%), while only 8 native methods (11.9%) reach the 300s timeout by NativeSummary. JN-SAF and NativeSummary are nearly similarly fast, we speculate this is because JN-SAF is implemented based on Angr's `CFGAccurate`, which is a combination of forced execution, symbolic execution, and backward slicing. As a CFG recovery algorithm, it is tuned to run faster than traditional symbolic execution. *These results suggest the scalability issue of existing techniques, while NativeSummary can achieve high time efficiency thanks to the abstract interpretation technique used.*

Breakdown of Execution Time: Table 5 shows the breakdown of average time spent across steps of NativeSummary on RQ2. Note that, the Dynamic Binding Resolution module and the abstract-interpretation-based Static Analysis Module make up the BinAbsInspector Solver component. The Java-level analysis is the most time-efficient. Because the static binding resolution module needs to iterate over all methods in the DEX bytecode to collect native methods, it takes a relatively longer time. Ghidra Load Binary, which is responsible for initial binary analysis (e.g., binary disassembly, function recognition, and CFG construction), takes nearly as much time (448.6s) as the BinAbsInspector Solver.

Answer to RQ3: NativeSummary is much more memory-efficient and time-efficient than SOTA tools. We believe NativeSummary is the only tool that is scalable for analyzing native code in real-world Android apps.

6 Discussion

6.1 Threats to Validity

We observe some cases that our approach could not handle.

- Change native state across native methods. For example, one native method stores data in native global variables or heap memory, and another native method reads it. The reason is that we analyze each native method separately without considering the dependencies between native methods. To the best of our knowledge, this issue is not solved by any existing tools.
- Conversion between pointers and Java numbers. For example, receiving a Java Long (jlong) as a parameter and then converting it to a pointer to a heap object. This is the same type of dataflow as the bench app "native handle" in NativeFlowExtended.
- Complex native computations that our abstract interpretation failed to track the data flow. These operations (e.g., encryption, compression, and string manipulations) often involve memory allocation and manipulation, whose dataflows are hard to track.
- Complex language features like function pointers, C++ virtual functions, and C++ exceptions. The usage of these features would impact the accuracy of our binary analysis.
- Java reflection can be obtained and invoked in native code. In this case, even if the related operation is correctly reflected in the generated bytecode, the actual operation performed is still hard to discover for the Java-side static analysis framework.
- The scalability of binary analysis is still not enough. Table 4 shows that some apps cannot be successfully analyzed in 2 hours. If the analysis time is doubled to 4 hours, then 1902 apps (84.3%) can be successfully analyzed.
- DEX dynamic loading and binary obfuscations. These issues are inherent challenges of Android static analysis. We believe that to better handle obfuscation, applying dedicated deobfuscation techniques is the most viable.
- To handle statically linked library code, an additional step to recognize common native libraries is required, or the analysis time is prolonged and the call is also not lifted to bytecode.
- Our approach does not faithfully reflect the control flow. If a native method only throws an exception under specific conditions, the condition is not captured in the generated body, we just protect this exception within an opaque if statement.

6.2 Enhancing Other Static Analysis Tasks

Besides taint analysis, NativeSummary can discover hidden call edges in native code. This makes calls in native code visible and benefits all analyses requiring a complete call graph (e.g. behavior-based malware detection methods [14, 15, 26, 34]). Additionally, by revealing native code behaviors, NativeSummary can enable previous studies that work on Dex bytecode with the ability to tackle native code. This includes analyses like Android app clone detection methods [9, 22, 38].

7 Conclusion

We propose NativeSummary, a novel inter-language static analysis framework of Android apps with high *scalability*, *accuracy*, and *compatibility*, based on abstract interpretation and selective code summarization. We perform extensive evaluations to show the effectiveness and efficiency of NativeSummary. NativeSummary opens doors for a new direction of Android app research, and many existing app analysis works can be boosted atop NativeSummary.

Data-Availability Statement

NativeSummary and related data are available at [6, 39].

Acknowledgement

This work was supported in part by National Key R&D Program of China (2021YFB2701000), the National Natural Science Foundation of China (grant No.62072046), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the Knowledge Innovation Program of Wuhan-Basic Research, HUST CSE-HongXin Joint Institute for Cyber Security and research grants from OPPO.

References

[1] 2022. AppShark. https://github.com/bytedance/appshark.

[2] 2022. BinAbsInspector. https://github.com/KeenSecurityLab/BinAbsInspector.

[3] 2022. jadx. https://github.com/skylot/jadx.

[4] 2022. libgdx. https://github.com/libgdx/libgdx.

[5] 2023. NativeFlowBenchExtended. https://github.com/NativeSummary/NativeFlowBenchExtended.

[6] 2023. NativeSummary. https://github.com/security-pride/NativeSummary.

[7] 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. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[8] Gogul Balakrishnan and Thomas Reps. 2010. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 1–84.

[9] Jonathan Crussell, Clint Gibler, and Hao Chen. 2014. Andarwin: Scalable detection of android application clones based on semantics. *IEEE Transactions on Mobile Computing* 14, 10 (2014), 2007–2019.

[10] Michael Dawson, Graeme Johnson, and Andrew Low. 2009. Best practices for using the Java Native Interface. *IBM developerWorks* (2009).

[11] Erik D Demaine. 1998. C to Java: converting pointers into references. *Concurrency: Practice and Experience* 10, 11-13 (1998), 851–861.

[12] Fahimeh Ebrahimi, Miroslav Tushev, and Anas Mahmoud. 2021. Mobile app privacy in software engineering research: A systematic mapping study. *Information and Software Technology* 133 (2021), 106466.

[13] F-Droid. 2023. F-Droid - Free and Open Source Android App Repository. https://f-droid.org/ Accessed: 2023-01-18.

[14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 576–587.

[15] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. 45–54.

[16] 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 *NDSS*, Vol. 15. 110.

[17] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 290–299.

[18] Java SE Documentation. 2020. Java Native Interface Specification. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[19] Java SE Documentation. 2020. Java Native Interface Specification Design Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html.

[20] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 127–137.

[21] 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 280–291. https://doi.org/10.1109/ICSE.2015.48

[22] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering* 47, 4 (2019), 676–693.

[23] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[24] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering* 27 (2022), 1–41.

[25] NSA. 2022. Ghidra. https://github.com/NationalSecurityAgency/ghidra.

[26] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2019. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). *ACM Trans. Priv. Secur.* 22, 2, Article 14 (apr 2019), 34 pages. https://doi.org/10.1145/3313391

[27] Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. 2023. Static Analysis of JNI Programs Via Binary Decompilation. *IEEE Transactions on Software Engineering* (2023).

[28] Venkatesh-Prasad Ranganath and Joydeep Mitra. 2020. Are free android app security analysis tools effective in detecting known vulnerabilities? *Empirical Software Engineering* 25 (2020), 178–219.

[29] Antonio Ruggia, Andrea Possemato, Savino Dambra, Alessio Merlo, Simone Aonzo, and Davide Balzarotti. 2022. The Dark Side of Native Code on Android. (2022).

[30] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. JuCify: a step towards Android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*. 1232–1244.

[31] Edward J Schwartz, J Lee, Maverick Woo, and David Brumley. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, Vol. 16.

[32] 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 *IEEE Symposium on Security and Privacy*.

[33] Cong Sun, Yuwan Ma, Dongrui Zeng, Gang Tan, Siqi Ma, and Yafei Wu. 2022. Dep: Mutation-based Dependency Generation for Precise Taint Analysis on Android Native Code. *IEEE Transactions on Dependable and Secure Computing* (2022).

[34] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 1–41.

[35] Gang Tan. 2010. JNI Light: An operational model for the core JNI. In *Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28-December 1, 2010. Proceedings 8*. Springer, 114–130.

[36] Gang Tan and Greg Morrisett. 2007. ILEA: Inter-language analysis across Java and C. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 39–56.

[37] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[38] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/2771783.2771795

[39] Jikai Wang. 2024. *Artifact for "NativeSummary: Summarizing Native Binary Code for Inter-language Static Analysis of Android Apps"*. https://doi.org/10.5281/zenodo.12664081

[40] Liu Wang, Haoyu Wang, Ren He, Ran Tao, Guozhu Meng, Xiapu Luo, and Xuanzhe Liu. 2022. MalRadar: Demystifying Android malware in the new era. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–27.

[41] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1137–1150.

[42] Fengguo Wei, Sankardas Roy, and Xinming Ou. 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)* 21, 3 (2018), 1–32.

[43] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 380–391. https://doi.org/10.1145/2884781.2884881

[44] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. 2021. Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4014–4040.

Received 2024-04-12; accepted 2024-07-03