Cross-Language Taint Analysis: Generating Caller-Sensitive Native Code Specification for Java

Shuangxiang Kan, Yuhao Gao, Zexin Zhong, and Yulei Sui

***Abstract*—Cross-language programming is a common practice within the software development industry, offering developers a multitude of advantages such as expressiveness, interoperability, and cross-platform compatibility, for developing large-scale applications. As an important example, JNI (Java Native Interface) programming is widely used in diverse scenarios where Java interacts with code written in other programming languages, such as C or C++. Conventional static analysis based on a single programming language faces challenges when it comes to tracing the flow of values across multiple modules that are coded in different programming languages. In this paper, we introduce CSS, a new *Caller-Sensitive Specification* approach designed to enhance the static taint analysis of Java programs employing JNI to interface with C/C++ code. In contrast to conservative specifications, this approach takes into consideration the calling context of the invoked C/C++ functions (or cross-language context), resulting in more precise and concise specifications for the side effects of native code. Furthermore, CSS specifically enhances the capabilities of Java analyzers, enabling them to perform precise static taint analysis across language boundaries into native code. The experimental results show that CSS can accurately summarize value-flow information and enhance the ability of Java monolingual static analyzers for cross-language taint flow tracking.**

***Index Terms*—Static analysis, taint analysis, cross-language program, caller-sensitive specification.**

I. Introduction

CROSS-LANGUAGE programming, involving the use of multiple programming languages within a single software project, has become increasingly relevant in today's diverse computing environment. Java, a widely-used language, employs the Java Native Interface (JNI) for this purpose. JNI allows Java to interact with C and C++ code, enabling access to functionalities like those in C/C++ libraries, such as OpenCV. However, JNI's use introduces the complexity of development [1] and also potential security vulnerabilities [1], [2], [3], [4], [5], [6], [7], [8], [9]. These risks are particularly concerning in the context of sensitive data exchanges, where malicious information flows can infiltrate from an app to native code (or third-party libraries) by exploiting a security vulnerability through JNIs. This allows attackers to bypass specific security checks, enabling activities such as identity theft and unauthorized account access [4], [5], [6], [7], [10].

Static taint analysis is increasingly becoming an important technique to ensure security for multi-module and multi-language frameworks [11], [12], [13], [14]. When undertaking cross-language static analysis, there are typically two strategies to consider. The first strategy involves compiling different programming languages into a singular intermediate representation (IR) [9], [15], [16], [17], [18], [19]. The limitation of this approach is the intrinsic and distinct features among different programming languages, creating a common IR to encompass programs written in distinct languages can be very challenging. Another limitation is the potential loss of information and inconsistencies when using a unified IR for analyzing different languages. In these tools [20], [21], [22], different languages are compiled into a unified IR separately. To accommodate multiple languages, compromises in the unified IR are often made due to separate compilation for distinct semantics of each language. Representing precisely the semantics at the border between two languages is challenging. For example, in JNI programs, Java and C/C++ have static and dynamic binding modes and distinct naming conventions. In addition, C/C++ consistently represents all Java objects as *jobject* without distinguishing specific Java object information. Even if a unified IR representing both Java and C/C++ code is used for later analysis, it may still fail to produce the desired results due to the loss of *jobject* information. The unified IR has to preserve all these API behaviors. As high-level languages evolve, modifying the design or implementation of the unified IR is also challenging. Accommodating all possible behaviors from high-level languages by modifying the unified IR is very hard and sometimes cost-ineffective.

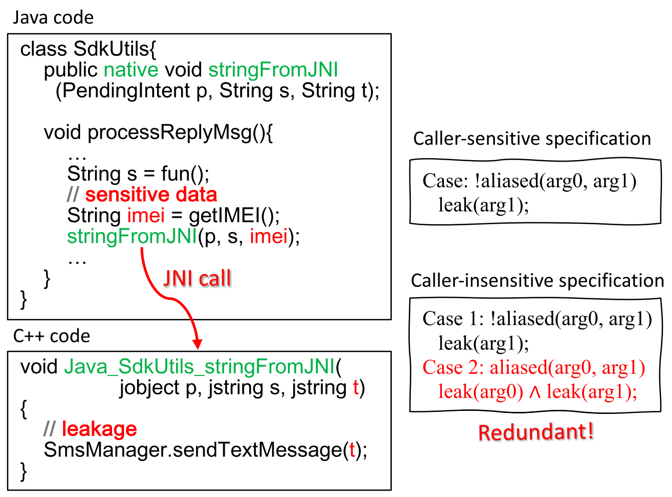
The second strategy utilizes specifications (including summaries and stub functions) to encapsulate the value-flow side-effects of native methods [9], [23], [24]. The specification-based approach represents another way of performing cross-language analysis where the existing single-language analyzers (e.g., Java analyzers) together with their various analysis algorithms can be reused with light or even no modifications. The specification-based approach, which centers on analyzing a summarized or condensed version of a library function rather than the entire function, has demonstrated its practicality in real-world cross-language and cross-module static analysis scenarios. However,

Manuscript received 30 August 2023; revised 28 March 2024; accepted 8 April 2024. Date of publication 27 May 2024; date of current version 14 June 2024. This work was supported in part by the Australian Research under Grant DP210101348 and Grant FT220100391, and in part by the generous Aspire Gift Grant from Google. Recommended for acceptance by T. Hall. *(Corresponding author: Yulei Sui.)*

Shuangxiang Kan and Yulei Sui are with the University of New South Wales, Sydney NSW 2052, Australia (e-mail: shuangxiang.kan@unsw.edu.au; y.sui@unsw.edu.au).

Yuhao Gao and Zexin Zhong are with the University of Technology Sydney, Sydney, NSW 2007, Australia (e-mail: yuhao.gao@student.uts.edu.au; zexin.zhong@student.uts.edu.au).

Digital Object Identifier 10.1109/TSE.2024.3392254


Java code
JNI call
C++ code
Caller-sensitive specification
Caller-insensitive specification
Redundant!

```
class SdkUtils{
    public native void stringFromJNI
      (PendingIntent p, String s, String t);

    void processReplyMsg(){
        ...
        String s = fun();
        // sensitive data
        String imei = getIMEI();
        stringFromJNI(p, s, imei);
        ...
    }
}
```

```
void Java_SdkUtils_stringFromJNI(
                jobject p, jstring s, jstring t)
{
    // leakage
    SmsManager.sendTextMessage(t);
}
```

```
Case: !aliased(arg0, arg1)
  leak(arg1);
```

```
Case 1: !aliased(arg0, arg1)
  leak(arg1);
Case 2: aliased(arg0, arg1)
  leak(arg0) ∧ leak(arg1);
```

Fig. 1. A data leakage JNI example (simplified).

automatically generating highly precise specifications for large-sized programs that involve multiple languages can pose significant challenges. This is due to the fact that specifications heavily depend on the analyzers, such as different value-flow or vulnerability checkers. The current cross-language static analysis approach often entails the creation of specifications either manually or automatically for each language independently, without considering the cross-language calling contexts, like calling context in Java that makes use of native libraries.

To illustrate, Fig. 1 is a simplified Android case of sensitive information leakage within the *Gumen* family in the Android Malware Dataset [25]. This case shows data leakage by transmitting the device's IMEI from the Java layer's *processReplyMsg()* method to the C++ code's *Java_SdkUtils_stringFromJNI()* function (the counterpart of *stringFromJNI()* method in the Java code). A conventional approach to specification generation prioritizes analysis soundness by accounting for all conceivable caller information and calling contexts. Nonetheless, such specifications frequently suffer from redundancy and imprecision. For example, in the initial caller-insensitive specification, the "*Case 2*" branch within the C++ program's specification becomes redundant if it's established that there is only one calling context for *stringFromJNI()* from the Java program callers. A more concise and precise caller-sensitive specification can be created, ensuring that *s* and *t* are always non-aliases. This accurately summarizes all feasible execution scenarios by considering all available code in both the Java and C++ programs. Our insight is that specifications for available source code (written in multiple languages) benefit from cross-language caller-sensitive summarization, which produces more precise and compact specifications than a single-language (e.g., C/C++) based conservative approach. Given that JNI interactions often require C/C++ code to invoke Java methods and access fields while relying on Java class or object information, conducting an independent analysis of C/C++ to over-approximate all possible input to native code can lead to the generation of redundant and imprecise specifications (as detailed in Section II-B1). Note that specifications may need to be updated if the available source code including caller information is changed.

In this paper, we introduce a new approach named *Caller-Sensitive Specification* (CSS) to generate precise and concise specifications for native calls within JNI environments. Given its automated specification extraction approach, CSS aims to improve efficiency and reduce the size of specifications to track value-flows across JNI programs. Unlike the conventional approach of analyzing isolated C/C++ functions, our caller-sensitivity pertains to contextual information (including aliases, method signatures, and tainted flows) across language boundaries, that is when a native callee is called by Java caller methods. The soundness of CSS is based on the assumption that all source code is available. This strategy enables us to identify the data attributes existing at the boundary between Java and C/C++ interactions – for instance, direct and indirect value-flows via parameter passings and returns. Subsequently, utilizing insights provided by the Java analyzer, the C/C++ static analyzer contributes to the extraction of value-flows of the corresponding native C/C++ functions. The outcomes of the C/C++ function's value-flows are conveyed back to the Java analyzer in the form of a specification, namely CSS, facilitating the iterative value-flow analysis.

Incorporating the cross-language context of called native functions into summarization can reduce redundancy with improved analysis efficiency and reduced storage space. Furthermore, our automated CSS approach can be used as a compact program representation for existing single-language static analyzers to aid in cross-language modeling. This compatibility empowers static analyzers to seamlessly execute effective cross-language static analysis with slight code adjustments. Though more precise given caller-sensitive information, the soundness of CSS is based on the availability of both Java and native source code. The specifications interface, capable of accommodating methods from diverse languages, demonstrates its versatility by supporting new languages without necessitating extensive modifications.

In summary, the following are our contributions:

- We introduce CSS, a new caller-sensitive specification approach designed to enhance the static value-flow analysis of JNI programs. This approach, which takes into account the calling context across languages, yields precise and compact specifications for efficient cross-language analysis.
- CSS integrates smoothly with Java static analysis tools specifically for JNI contexts. The incorporation of our proposed caller-sensitive specifications enables these tools to handle Java/Native interactions more efficiently within their existing frameworks.
- We have conducted comprehensive experiments aimed at comparing the value-flow analysis using CSS in JNI programs. Through the application of CSS, existing Java static analyzers demonstrate enhanced performance in tracing value-flow within JNI programs, outperforming current cross-language approaches.

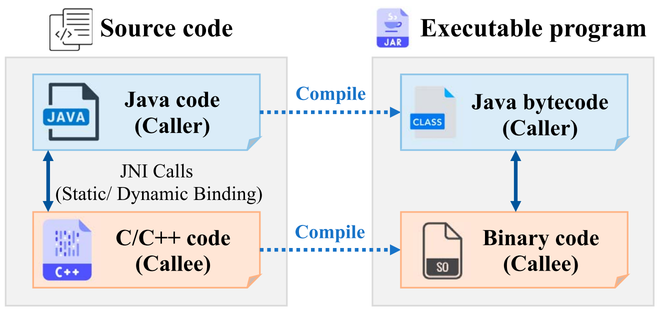


Fig. 2. JNI interoperation.

II. Background and Motivating Example

A. JNI Interoperation

The Java Native Interface (JNI) is a mechanism enabling Java code to interact with native code in languages like C and C++ (In this paper, the terms "native code" and "C/C++ code" are used interchangeably). It facilitates Java methods to call native functions and vice versa. This is particularly useful for accessing platform-specific features or using performance-critical libraries written in native languages.

JNI calls. JNI enables the development of cross-language programs, allowing developers to use Java and C/C++ seamlessly in a simple workflow like Fig. 2. This process involves declaring and calling a "*native*" method in Java, which corresponds to a C++ callee function in C/C++. The Java code is compiled to Java bytecode, while the C/C++ code is compiled to binary code within a dynamic library. The Java Virtual Machine (JVM) facilitates JNI calls, enabling communication between the Java bytecode and binary code during program execution.

Binding modes. There are two binding modes for JNI development: static binding and dynamic binding. Static binding links a native method directly to its implementation during compilation, with a fixed function name format like *Java_PackageName_ClassName_MethodName* (e.g., *Java_SdkUtils_stringFromJNI()* in Fig. 1), determined at compile time. Additional macros and keywords, such as *JNIEXPORT* and *JNICALL*, ensure proper linkage between Java and native code. The *JNIEnv** pointer refers to the *JNIEnv* interface, providing JNI functions to interact with the Java runtime, and the *jobject* or *jclass* is a reference to the invoking Java object or class. On the other hand, dynamic binding allows registering native methods at runtime using any naming scheme through the *RegisterNatives()* function and the *JNI_OnLoad()* function during native library initialization.

JNI functions. JNI provides a set of functions, such as locating classes, accessing methods and fields, and creating objects, that facilitate communication between Java and native code. JNI functions serve as a bridge between the two worlds, enabling seamless interaction and communication. By using these functions, developers can extend the capabilities of Java applications by leveraging platform-specific features and performance-critical native libraries. For example, JNI function *FindClass()* can locate a Java class using its fully qualified name.

B. Motivating Example

Fig. 3 illustrates a JNI example demonstrating how sensitive data propagates between Java (Fig. 3(a)) and C++ (Fig. 3(b)). The interface *void native_c_callee(User, User)*, declared with the "*native*" keyword, serves as the entry point for the Java code to execute the corresponding C++ function. The function *void Java_JavaCaller_native_c_callee (JNIEnv*, jobject, jobject, jobject)* (Lines C2 and C3) represents the C/C++ function implementation of this native method, which will handle the data that passed from the Java side.

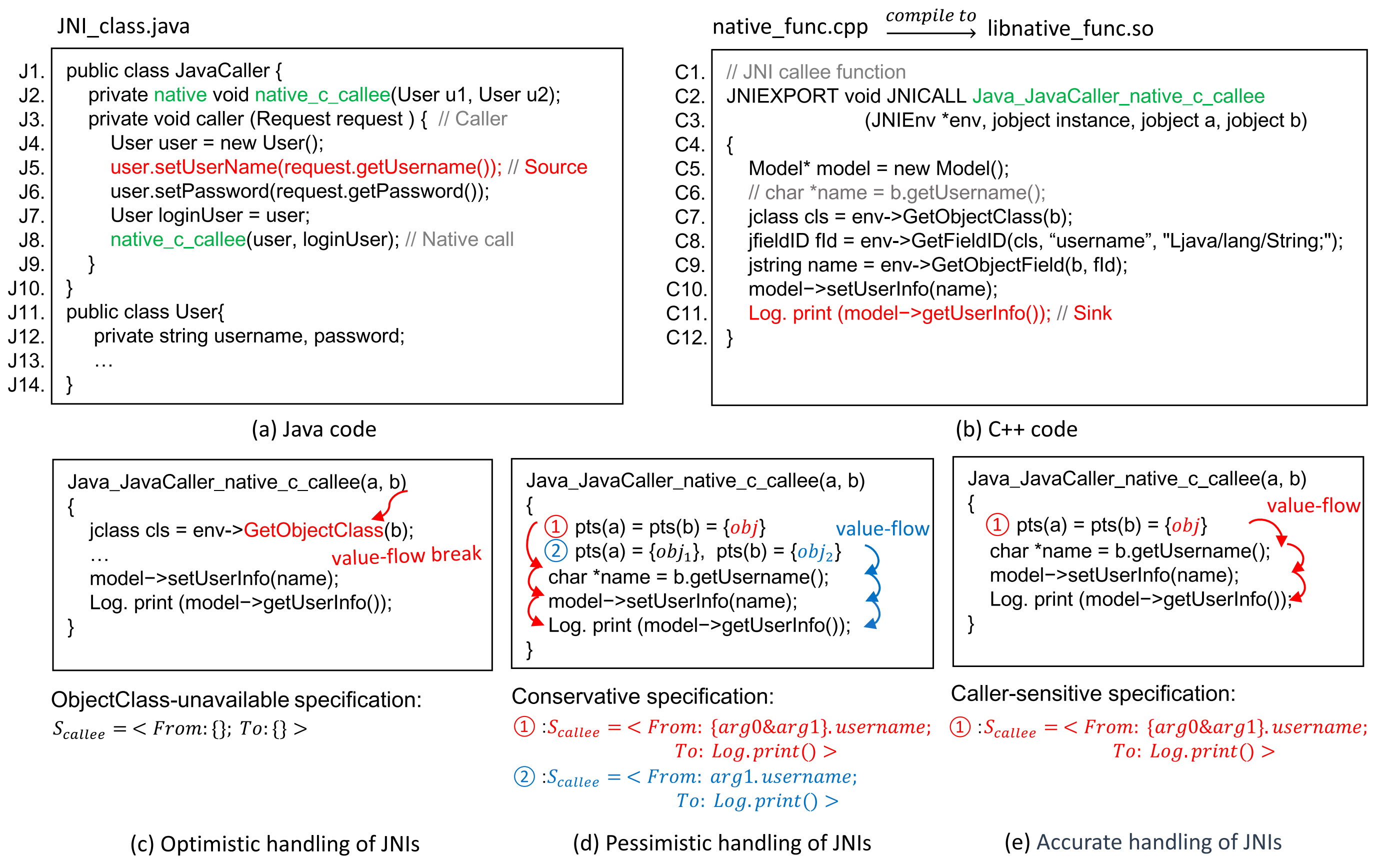
The method *caller()* (Line J3) in the Java side loads the sensitive user data (Line J5) from the request and stores it in an object of *User*. In addition, another object *loginUser* will load and store the object *user* (Line J7). Because the *loginUser* and *user* refer to the same memory, *loginUser.username* and *user.username* are aliased. Finally, the C++ function will be invoked at Line J8 by calling the native method *native_c_callee()* with two aliased arguments *user* and *loginUser*.

In C++ function *void Java_JavaCaller_native_c_callee*(), there exists a potential data leakage of sensitive source data. The *username* of the second parameter, *b*, is obtained through the three JNI functions at Lines C7-9 and then stored into the *userInfo* of object *model* (Line C10), ultimately being leaked to a log file by calling *Log.print()* at Line C11. Since *b.username* and *a.username* are aliased, *a.username* is also leaked when calling *Log.print()*. Next, we will briefly compare CSS and other approaches when building native summarizations to trace sensitive data and detect information leakages across languages.

1) Taint Analysis With Caller-Insensitive Specification (CIS): Conventional Java static analyzers typically conduct a manually defined solution for native library summarizations. These approaches require developers or users to provide value-flow specifications that identify source-to-sink paths of the native functions for the analyzers [26], [27]. However, ensuring the accuracy of manually defined specifications can be challenging because they are often isolated from the cross-language calling context, which is caller-insensitive (Caller-insensitive Specification, CIS), and mainly focuses on analyzing the behaviour of native functions. As a result, the accuracy of such specifications is hard to guarantee. Furthermore, these manually defined specifications either selectively consider certain cases (such as excluding alias situations) or include all possible taint and alias situations.

Optimistic handling of JNIs. If the analysis is conducted exclusively on *Java_JavaCaller_native_c_callee*(), the first challenge is JNI functions. As the C/C++ analyzer encounters *getObjectClass(b)* at Line C7, it faces difficulty in obtaining information about the object *b*, which results in a broken value-flow of the taint analysis at this line. This hinders the creation of a valid source-sink specification, as depicted in the ObjectClass-unavailable specification in Fig. 3(c). This optimistic approach assumes no side effects on unresolved JNI functions, simplifying analysis but risking unsound results.

Pessimistic handling of JNIs. Even if a C/C++ analyzer has access to information about object *b*, in order to maintain soundness, the pessimistic or conservative approach assumes that C/C++ code can execute any action. A conservative


JNI_class.java

```
J1.  public class JavaCaller {
J2.     private native void native_c_callee(User u1, User u2);
J3.     private void caller (Request request ) {  // Caller
J4.         User user = new User();
J5.         user.setUserName(request.getUsername()); // Source
J6.         user.setPassword(request.getPassword());
J7.         User loginUser = user;
J8.         native_c_callee(user, loginUser); // Native call
J9.     }
J10. }
J11. public class User{
J12.     private string username, password;
J13.     …
J14. }
```

(a) Java code

native_func.cpp $\xrightarrow{compile\ to}$ libnative_func.so

```
C1.  // JNI callee function
C2.  JNIEXPORT void JNICALL Java_JavaCaller_native_c_callee
C3.                 (JNIEnv *env, jobject instance, jobject a, jobject b)
C4.  {
C5.     Model* model = new Model();
C6.     // char *name = b.getUsername();
C7.     jclass cls = env->GetObjectClass(b);
C8.     jfieldID fld = env->GetFieldID(cls, “username”, "Ljava/lang/String;");
C9.     jstring name = env->GetObjectField(b, fld);
C10.    model->setUserInfo(name);
C11.    Log. print (model->getUserInfo()); // Sink
C12. }
```

(b) C++ code

```
Java_JavaCaller_native_c_callee(a, b)
{
    jclass cls = env->GetObjectClass(b);
    …                                    value-flow break
    model->setUserInfo(name);
    Log. print (model->getUserInfo());
}
```

ObjectClass-unavailable specification:
$S_{callee} = < From: \{\};\ To: \{\} >$

(c) Optimistic handling of JNIs

```
Java_JavaCaller_native_c_callee(a, b)
{
 ① pts(a) = pts(b) = {obj}             value-flow
 ② pts(a) = {obj1},  pts(b) = {obj2}
  char *name = b.getUsername();
  model->setUserInfo(name);
  Log. print (model->getUserInfo());
}
```

Conservative specification:
① $:S_{callee} = < From:\ \{arg0\&arg1\}.username;\ To:\ Log.print() >$
② $:S_{callee} = < From:\ arg1.username;\ To:\ Log.print() >$

(d) Pessimistic handling of JNIs

```
Java_JavaCaller_native_c_callee(a, b)
{                                       value-flow
  ① pts(a) = pts(b) = {obj}
  char *name = b.getUsername();
  model->setUserInfo(name);
  Log. print (model->getUserInfo());
}
```

Caller-sensitive specification:
① $:S_{callee} = < From:\ \{arg0\&arg1\}.username;\ To:\ Log.print() >$

(e) Accurate handling of JNIs

Fig. 3. An illustrating JNI example with Java and native code.

specification would include the following two alias cases for the function *Java_JavaCaller_native_c_callee()*:

Case 1: alias(arg0, arg1) == true (red value-flow ①)

$S_{callee} = \langle From: \{arg0\&arg1\}.username,\ To: Log.print()\rangle$

Case 2: alias(arg0, arg1) == false (blue value-flow ②)

$S_{callee} = \langle From: arg1.username,\ To: Log.print()\rangle$.

For the conservative specification in Fig. 3(d), *pts(v)* represents the points-to set of variable *v*, Lines C7-9 are simplified to *char *name = b.getUsername();* for easy understanding. However, these redundancy cases in CIS can impact the analysis efficiency if the conditions are complicated in large-size summarizations, and it could also lead to over-tainting and impact the performance of the analysis.

2) Taint Analysis With Caller-Sensitive Specification (CSS): A taint analyzer with CSS typically comprises a Java analyzer and a C/C++ analyzer, each responsible for analyzing the specific programming language. The Java analyzer first performs taint analysis on the Java side and traces the propagation of sensitive data. In this example, the source data of the *username* loaded from the *request* is marked and traced by the Java taint analyzer. Due to *loginUser* and *user* are made to be aliases at Line J7, and thus *loginUser.username* is also tainted. When calling native method *native_c_callee*(), as the Java analyzer lacks traceability to C/C++ code, the taint analysis will be handed over to the C/C++ analyzer. Prior to this, the Java analyzer will generate a caller specification $S_{caller} = \langle taint: \{arg0\ \&arg1\}.username; alias: \{arg0, arg1\}; arg\ Types: \{User, User\}\rangle$ that includes cross-language context information about this native call.

The C/C++ analyzer then takes over the analyzing task using the caller specification S_{caller}. Because S_{caller} contains the object information of *b* (i.e., *User* in the caller), the functionality of three JNI functions can be successfully parsed, leading to the retrieval of *b*'s username. *b.username* is ultimately leaked to the *Log.print()* function. Since S_{caller} indicates that *b.username* and *a.username* are aliased, *a.username* also leaks to the *Log.print()* function. When the analysis on the C/C++ side is completed, a specification representing the taint analysis result in the C/C++ side of the callee is generated in the form of $S_{callee} = \langle From: \{arg0\&arg1\}.username;\ To: Log.print()\rangle$ (Caller-sensitive specification in Fig. 3(e)). This generated specification is then sent back to the Java analyzer for further taint tracing. The above value-flow analysis process reveals that accurate value-flow for C/C++ functions is achieved given the knowledge of cross-language contexts.

3) Taint Analysis With JN-SAF and JuCify: JN-SAF [28] is an Android static analysis framework that handles native method calls. It constructs the call graph from *request.getUsername()* to *Log.print()* and performs taint analysis. While JN-SAF can identify the source-to-sink path in this small

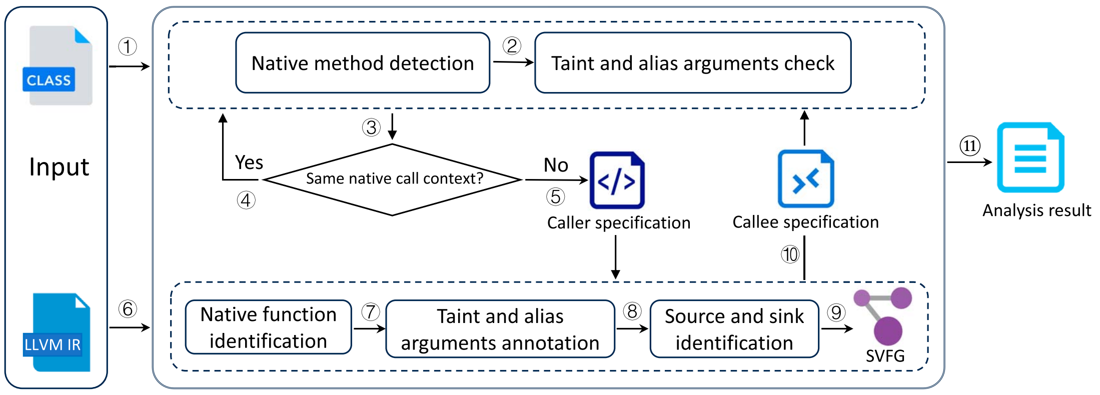

CLASS
Input
LLVM IR
①
⑥
Native method detection
②
Taint and alias arguments check
③
Yes
④
Same native call context?
No
⑤
Caller specification
Callee specification
⑩
Native function identification
⑦
Taint and alias arguments annotation
⑧
Source and sink identification
⑨
SVFG
⑪
Analysis result

Fig. 4. The workflow of our cross-language taint analysis with caller-sensitive specifications.

example, its scalability is constrained for larger programs due to path explosion problems with Angr's [29] symbolic execution. Moreover, it may yield inconsistent results when dealing with binary code compiled at different optimization levels [30]. Another Android static analysis framework, JuCify [17], aims to construct a unified model for Android code. However, Jucify is unable to reconstruct the behavior of pure native functions. Consequently, its tainted flow is broken at *setUserInfo()* and *print()*. These limitations affect the accuracy and scalability of cross-language static analysis in these tools.

III. Methodology

A. Workflow

Fig. 4 illustrates our taint analysis workflow with CSS for JNI programs. It starts by analyzing the bytecode to identify tainted data sources. To demonstrate the generalization of our CSS summarization, we use recent popular taint analyzers Flowdroid [13], Tai-e [27] and, WALA [21] for Android and Java. Analyzers are employed to track the flow of tainted data and gather contextual information for specifying native method callers. Upon finding a called native method (Step ①) with tainted arguments (Step ②), it checks whether the same calling context has been previously analyzed (Step ③). If the same calling context analysis has been performed before, the Java static analyzer will reuse the pre-generated native call specification to reduce redundant analysis (Step ④). Otherwise, a caller specification (Step ⑤) is generated to record the current native calling context.

The caller specification contains method signature, and information about tainted and aliased arguments. For callee specification, we use SVF, a C/C++ static analyzer [12], [31] to combine this information with LLVM IR to identify native functions (Step ⑥) and annotate tainted and aliased arguments (Step ⑦). Sources and sinks in native code are identified based on caller specifications and user configurations (Step ⑧).

The Sparse Value-Flow Graph (SVFG) [31] (Step ⑨) represents interprocedural value-flows across C/C++ functions, helping identify paths from native sources to native sinks. A caller-sensitive specification for each callee is generated using SVFG (Step ⑩), including native method signatures, callbacks, native sources of tainted data and sinks. This specification is then sent back to the Java static analyzer to continue the analysis. As illustrated in Fig. 4, the process of generating caller and callee specifications is conducted in an iterative manner until a fixed point is reached (Step ⑪).

B. Caller Specification Extraction on Java Side

We employ alias analysis and taint analysis in a Java taint analyzer (e.g., FlowDroid, Tai-e and, WALA) to construct caller specifications. In these tools, a forward taint analysis is integrated with an on-demand alias analysis. A native callee function can be invoked by multiple callers from Java code with each call involving different arguments being passed to the callee's parameters. Please note that our caller-sensitive approach, unlike the traditional context-sensitive analysis, focuses on language boundaries, where the caller-sensitive contextual information transferred to native code can be computed through any (context-insensitive or context-sensitive) Java alias and taint analysis. We consider the argument information passed from the Java side caller before extracting value-flow specifications for C/C++ side functions. By following this approach, we can generate specific specifications that precisely reflect the program's behavior, as opposed to relying on a single, all-encompassing conservative specification that might fail to accurately capture the caller-sensitive behaviors.

The analysis begins by identifying sources and sinks, after which data from the input sources is labeled starting from the entry points. Throughout this process, each callsite is examined to determine if the method being called is a native method (indicated by the "*native*" keyword in its declaration). If the method is native and its arguments contain source data when being invoked, a three-tuple specification $S_{caller} = (S_e, D, A)$ (Fig. 5) is generated to capture the calling context of this native method.

Specifically, S_e contains the signature of the called native method, including fully qualified method m (package name, class name, and method name), the return type r and the list of n parameter types P. In order to identify tainted arguments and aliases, D and A are introduced. D represents a list of arguments indicating which argument a_i is tainted or not

$\mathbb{P}\ (parameter) = \{p_i \mid p_i \in \mathbb{T}, 0 \le i < n\}$
$\mathbb{S}\ (signature) = \{(m, r, P) \mid m \in \mathbb{M}, r \in \mathbb{T}, P \in \mathbb{P}\}$
$\mathbb{D}\ (taintArgs) = \{a_i \mid taint(a_i) = true, 0 \le i < n\}$
$\mathbb{A}\ (aliasArgs) = \{(a_j, a_k) \mid alias(a_j, a_k) = true, 0 \le j \ne k < n\}$
$S_{caller} = \{(S_e, D, A) \mid S_e \in \mathbb{S}, D \in \mathbb{D}, A \in \mathbb{A}\}$

Fig. 5. Three-tuple caller specification.

$(taint(a_i) = true$ or $false)$. A represents a list of pairs of arguments, where each pair (a_j, a_k) represents an alias relationship between argument a_j and argument a_k $(alias(a_j, a_k) = true$ or $false)$. If tainted data and aliases occur at a deeper level within the arguments, such as $a_j.f$ and $a_k.g$, we mark the specific level of the taint and alias relationships in fields D and A.

When marking tainted parameters, a recursive access path may arise. For example, when a parameter's field points back to the parameter itself (*a.f* = *a*), the length of the access path can potentially become infinite (*a.f, a.f.f, a.f.f...f*). We use *k*-limiting [32] to address this issue. We set a fixed value *k*, signifying the maximum length for all access paths within our analysis. In cases where the analysis produces an access path that surpasses this predetermined value *k*, it is truncated and annotated with an asterisk. For instance, when *k* = 2, "*o.f.f.f*" is represented as "*o.f.f.**" and all objects pointed to by fields after "*o.f.f*" are considered tainted. Moreover, using *k*-limiting to truncate the access path is sound because *o.f.f.f* is a subset of *o.f.f*, and using *o.f.f* to represent *o.f.f.f* will cause the result of static analysis to be conservative.

But before generating a new S_{caller}, the Java analyzer will check whether the same native call context has been encountered previously. This context involves the call to the same native method and the same aliases and tainted flows. If such a context has occurred before, it reuses the existing native call specification (S_{callee}) and continues the analysis of the Java code. Otherwise, a new S_{caller} is generated for the subsequent C/C++ code analysis.

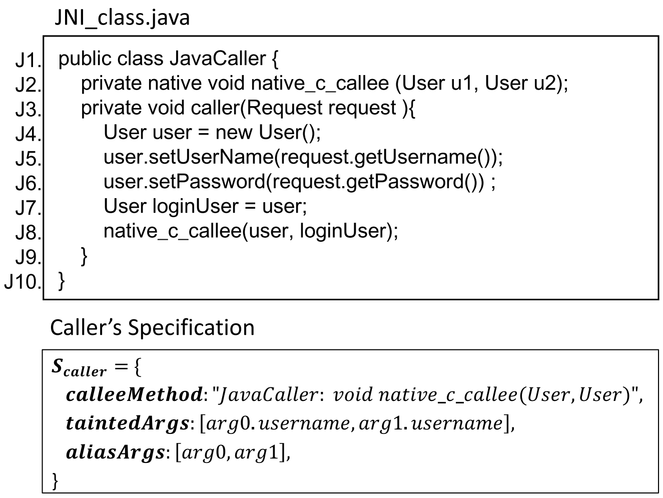
For example, the caller specification S_{caller} of Java method *caller()* is like the format of Fig. 6. *calleeMethod* (Field S_e) records the signature of the invoked native method *native_c_callee()*. *taintedArgs* (Field *D*) and *aliasArgs* (Field *A*) indicate that the *username* of first and second arguments of this native call are both tainted and aliased.

C. Callee's Value-Flow Specification Extraction on C/C++ Side

Once the caller specification S_{caller} is generated, it will be used by C/C++ analyzers to produce the callee specification with source-sink information for native functions. The caller specifications are combined with the LLVM IR compiled by C/C++ code to produce a CSS containing the value-flows of tainted arguments.

Fig. 4 illustrates the five steps involved in generating a caller-sensitive (callee) specification:

- **Native function identification**: Accurate mapping of a native method in Java to its corresponding function in C/C++ is necessary to locate the implementation code of the method, as naming conventions for native methods differ between the two languages.
- **Taint and alias arguments annotation**: This step can assist in more precisely tracking tainted arguments in determined native function and identifying alias relationships among them.
- **Sources and sinks determination in native code**: Native sources and sinks need to be specified to generate interprocedural value-flows in native code.
- **Sparse value-flow graph (SVFG) construction**: Sparse value-flow graph is then constructed to capture the def-use chains of variables for sources and sinks.
- **Caller-sensitive specification generation**: This is accomplished by solving graph reachability on SVFG, which entails traversing from sources to sinks in native code. According to whether the sources can reach the sinks, the corresponding specification is generated.


JNI_class.java

```
public class JavaCaller {
    private native void native_c_callee (User u1, User u2);
    private void caller(Request request ){
        User user = new User();
        user.setUserName(request.getUsername());
        user.setPassword(request.getPassword()) ;
        User loginUser = user;
        native_c_callee(user, loginUser);
    }
}
```

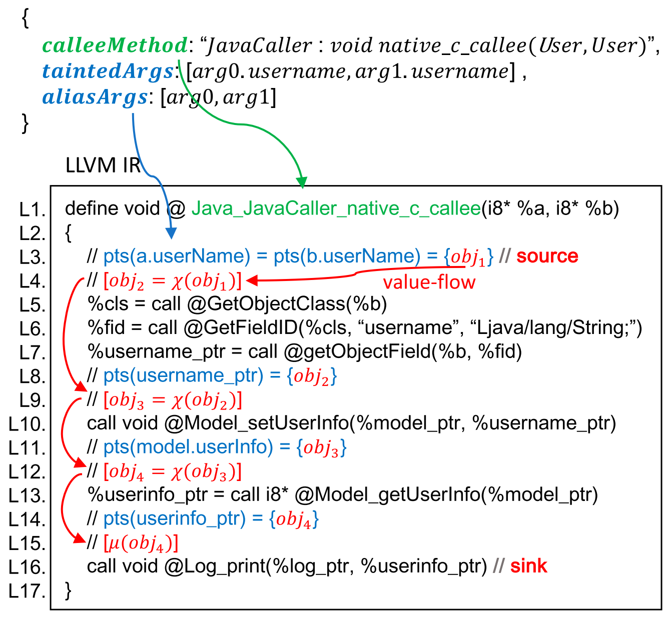
Caller's Specification

```
S_caller = {
  calleeMethod: "JavaCaller: void native_c_callee(User, User)",
  taintedArgs: [arg0.username, arg1.username],
  aliasArgs: [arg0, arg1],
}
```

Fig. 6. Caller's specification extraction.

Before the five steps mentioned earlier, C/C++ code is converted to LLVM IR using *clang* or other tools. In LLVM IR, each instruction is in Static Single Assignment (SSA) form, ensuring that each variable is assigned only once. LLVM IR comprises two types of variables: top-level variables and address-taken variables. Top-level variables, which consist of stack virtual registers (prefixed with "%") and global variables (prefixed with "@"), are explicit and maintain the SSA form. On the other hand, address-taken variables are accessed indirectly through "*Load*" or "*Store*" instructions and encompass stack objects, heap objects, and global objects.

1) Native Function Identification: As described in Section II-A, the naming conventions for native methods differ between Java and C/C++. In Java, native methods are typically declared using the "*native*" keyword, followed by the method signature. To ensure accurate mapping between Java and C/C++, functions that adhere to the JNI naming rules and those registered dynamically through *(env)→RegisterNatives()* within the *JNI_OnLoad()* function in C/C++ code will be searched to find



```
{
  calleeMethod: "JavaCaller : void native_c_callee(User, User)",
  taintedArgs: [arg0.username, arg1.username] ,
  aliasArgs: [arg0, arg1]
}
L1.  define void @ Java_JavaCaller_native_c_callee(i8* %a, i8* %b)
L2.  {
L3.    // pts(a.userName) = pts(b.userName) = {obj_1} // source
L4.    // [obj_2 = χ(obj_1)]
L5.    %cls = call @GetObjectClass(%b)
L6.    %fid = call @GetFieldID(%cls, "username", "Ljava/lang/String;")
L7.    %username_ptr = call @getObjectField(%b, %fid)
L8.    // pts(username_ptr) = {obj_2}
L9.    // [obj_3 = χ(obj_2)]
L10.   call void @Model_setUserInfo(%model_ptr, %username_ptr)
L11.   // pts(model.userInfo) = {obj_3}
L12.   // [obj_4 = χ(obj_3)]
L13.   %userinfo_ptr = call i8* @Model_getUserInfo(%model_ptr)
L14.   // pts(userinfo_ptr) = {obj_4}
L15.   // [μ(obj_4)]
L16.   call void @Log_print(%log_ptr, %userinfo_ptr) // sink
L17. }
```

Fig. 7. A fragment of LLVM IR and its memory SSA form of the motivating example in Fig. 3.

the counterpart of the Java native method (green curved arrow in Fig. 7).

2) Taint and Alias Arguments Annotation: Fields D (taintArgs) and A (aliasArgs) in S_{caller} are used after the C/C++ function is determined. D is used to mark which arguments are "tainted", which can help eliminate the influence of other "untainted" arguments and focus only on the tainted ones. A is used when allocating abstract objects for the C/C++ function arguments. If two arguments are aliased, only one object is allocated for both of them. Conversely, if they are not aliased, then two different objects are allocated for each of the arguments, respectively. For example, in Fig. 7, since *taintedArgs* (Field D) in S_{caller} indicates that the field *username* of first and second arguments are tainted, then the objects pointed by *a.username* and *b.username* are marked as sources and their value-flows will be tracked. In addition, *aliasArgs* (Field A) indicates that parameters a and b are aliased, so only one object obj_1 is assigned to *a.username* and *b.username* (blue curved arrow).

3) Sources and Sinks Determination in Native Code: The parameters indicated in the field D of S_{caller} are marked as sources (or native sources), such as the field *username* of the first and second parameters in Fig. 7. Sinks in native code (or native sinks) need to determine whether the tainted data flows to return value or is leaked to the outside world (data confidentiality is violated). Some sink functions can be configured before the analysis, such as *fprintf()* or *__android_log_print()*, to check whether native sources are used as arguments of these native sink functions. In the motivating example, we are interested in whether the user's *name* or *password* information is written to the logs, so *Log.print()* is marked as a native sink.

4) Sparse Value-Flow Graph (SVFG) Construction: Sparse pointer analysis propagates pointer information from variable definitions to their uses through pre-computed def-use chains. Sparse value-flow graph (SVFG) [31], [33] captures def-use chains and value-flow via assignments for all memory locations represented by both top-level and address-taken pointers. After getting the S_{caller} and LLVM IR, SVFG is constructed to capture the value-flow paths within and across procedural boundaries. During the construction of SVFG, the side-effects of instructions such as "*Load*", "*Copy*", "*Store*", "*Phi*", and "*Call*" are annotated using χ and μ functions. χ and μ functions were used to represent potential defs and uses of a memory object o at stores or loads, where $\mu(o)$ indicates the use of variable o, and $o = \chi(o)$ represents the def and use of o [34].

However, JNI C/C++ functions differ from regular C/C++ functions and require addressing the following three issues:

a) Constructing points-to sets of arguments: Before constructing the SVFG of a determined C/C++ function, one challenge is that the caller on the C/C++ side is absent, as the caller resides on the Java side. Consequently, the points-to set of the arguments is initially empty. To resolve this, it becomes necessary to construct the points-to sets of the arguments based on the fields D and A in the caller specification S_{caller}. When multiple arguments have alias relationships, only one object is created (e.g., obj_1 in Fig. 7). Conversely, for arguments without alias relationships, separate objects are created.

b) JNI type conversion functions modeling: Another difficulty in building the SVFG arises from JNI-provided functions. JNI functions are a set of native programming interfaces provided by the JDK, enabling developers to interact with Java objects, fields, and methods in native code. Type conversion functions are a subset of JNI functions specifically used to convert data between Java and C/C++ data types. Given the distinct data type systems between Java and C/C++, JNI functions play a crucial role in Java and C/C++ communication.

CSS aims to automatically summarize the native application code (C/C++ code) by modelling the behaviour of JNI APIs. These APIs when used in C/C++ have pre-defined semantics. CSS supports modeling of JNI interactions between Java and C in a caller-sensitive manner for precise summarization of native code. When compiling JNI functions into LLVM IR, their source code is often not included. We have abstracted their side-effects to address this challenge in value-flow analysis. This strategy improves value-flow analysis by focusing on user-defined logic and efficiently handling the complexities of JNI functions. For example, we replaced the body of *const char* addr = env→ GetStringUTFChars(env, addr_, NULL)*, which retrieves a pointer to a UTF-8 encoded string representing a Java String object, with a store statement **addr=addr_*, allowing us to handle JNI functions without access to their source code.

c) JNI callbacks to java methods and fields handling: In native code, when callbacks to Java fields and methods are needed, a series of JNI functions are used. These functions contain the complete signatures of the Java fields and methods being called back, such as class names, method names, and, types. If C/C++ analysis can correctly resolve the callbacks based on the signatures of the Java fields and methods, we will parse these callbacks. For example, Lines L5-7 are considered an operation to retrieve the "*username*" field of object b. For

$$\begin{aligned}
&\mathbb{P}\ (parameter) = \{p_i \mid p_i \in \mathbb{T}, 0 \leq i < n\}\\
&\mathbb{S}\ (signature) = \{(m, r, P) \mid m \in \mathbb{M}, r \in \mathbb{T}, P \in \mathbb{P}\}\\
&\mathbb{F}\ (field) = \{(f, p) \mid f \in \mathbb{M}, p \in \mathbb{T}\}\\
&\mathbb{Q}\ (source) = \{a_i \mid taint(a_i) = true, 0 \leq i < n\}\\
&\mathbb{B}\ (sink) = \{(ret \mid S_k) \mid S_k \in \mathbb{S}\}\\
&\mathbb{J}\ (callback) = \{(F_j \mid S_j) \mid F_j \in \mathbb{F}, S_j \in \mathbb{S}\}\\
&S_{callee} = \{(S_e, Q, B, J) \mid S_e \in \mathbb{S}, Q \in \mathbb{Q}, B \in \mathbb{B}, J \in \mathbb{J}\}
\end{aligned}$$

Fig. 8. Four-tuples callee specification.

callbacks with tainted values that the C/C++ analysis cannot resolve, we record the signatures of callback Java fields or methods and mark which field or argument is tainted. This information is then passed to the Java analyzer after native code analysis.

5) Caller-Sensitive Specification Generation: Once the SVFG is built, the value-flow paths of top-level and address-taken variables can be obtained by solving a graph reachability problem on it. For example, the value-flow of address-taken variables obj_1 in Fig. 7 is $L3 \rightarrow L4 \rightarrow L9 \rightarrow L12 \rightarrow L15$ (red curved arrow). Finally, the value-flow passed in the function *Log_print()* through an argument. Then a four-tuple caller-sensitive specification $S_{callee} = (S_e, Q, B, J)$, as shown in Fig. 8, is created to record the tainted value-flow.

S_{callee} is similar to S_{caller}, but fields D and A in S_{caller} are replaced by Q and B in S_{callee}. Additionally, we introduce J to record call-back Java methods S or Java fields F in native code, where F contains field names and types. When S_{callee} is returned to the Java static analyzer for use, S_e is used to identify and locate which native method is called. Q contains the taint arguments flowing to a sink in the native function. B indicates the destination of the tainted arguments in Q. B has two destinations: ret means that it flows back to the Java side through the return value, S_k is the signature of the sink function, indicating that arguments in Q finally flows to the sink function. J contains Java methods and fields that are accessed sequentially in the order they are invoked from the native code callback.

For instance, Fig. 9 shows the CSS of the function $Java\_JavaCaller\_native\_c\_callee()$, which captures the tainted value-flow originating from the field *username* of first and second arguments (*From: [arg0.username, arg1.username]*) and directs it towards the native function *Log.print()* (*To: [Log.print()]*). Because we can deduce from S_{caller}'s parameter information that the role of Lines L5-7 in Fig. 7 is to retrieve the *username* from *b*, so *callbacks* (Field J) in Fig. 9 is empty. Otherwise, we need to record callback information for Java fields.

IV. Evaluation

We present the evaluation of our CSS, aiming to assess its ability to precisely summarize the value-flows of the native code to be used by existing Java analyzers. We conducted three experiments to validate our approach.

Experiment 1: Android benchmark *NativeFlowBench* For this experiment, we utilize the benchmark *Native FlowBench* as introduced in a previous study [28]. This benchmark contains 23 hand-crafted Android JNI applications, offering a wide spectrum of interoperability features. Our analysis employs Flowdroid [13] to conduct value-flow analysis on the Java code and SVF [12], [31] for C/C++ code. This combination of analysis tools allows us to evaluate the effectiveness of CSS in tracking value-flows across both Java and C/C++ components of these Android applications.

Experiment 2: Java benchmark *JavaNativeBench*. To further evaluate the robustness of CSS, we create a set of custom-designed applications *JavaNativeBench* that simulate real-world JNI interactions. These applications are crafted to cover scenarios not addressed in *Experiment 1*. We leverage Tai-e [27], WALA [21], Flowdroid, and, JN-SAF to perform value-flow analysis. In addition, we conduct a comparative experiment between CSS and another summary-based approach to evaluate the quality of CSS.

Experiment 3: Real-world JNI projects. To assess the practical applicability and correctness of CSS within real-world contexts, we examine ten real-world JNI applications sourced from Github. These applications represent diverse Android projects that extensively utilize JNI, providing a comprehensive and robust evaluation platform for our CSS approach. We use FlowDroid for value-flow analysis on the Java components and SVF for the C/C++ components.

Spanning these three experiments, our objective is to demonstrate the effectiveness of the proposed CSS in precisely summarizing value-flows within the native code and proficiently tracing value-flows across JNI programs. Utilizing FlowDroid, Tai-e and, WALA for Android and Java applications separately demonstrates how CSS integrates with various Java static analyzers, demonstrating its ability to improve single-language analyzers for cross-language analysis.

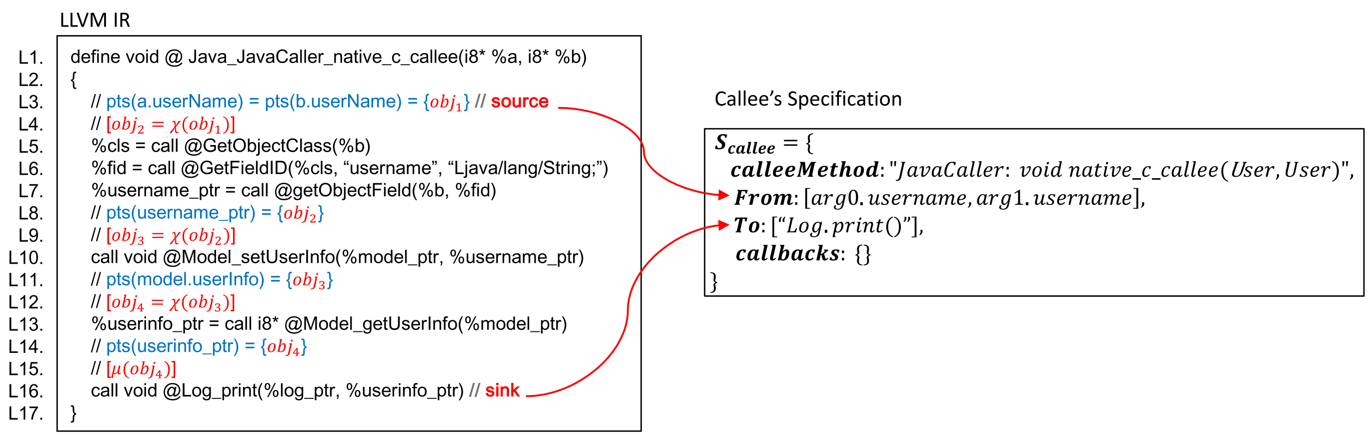
Environment setup. All of our evaluations were performed on 12th Gen Intel(R) Core(TM) i7-12700 with 16GB of RAM. The O.S. is Ubuntu 22.04.3 LTS with Linux 6.2.0-33-generic 64-bit. Before experimenting with each project, sources and sinks will be predetermined, and a manual check for ground truths will be conducted to confirm the existence and number of paths from sources to sinks. Additionally, projects with excessively long running times due to bugs in the baseline tools will be manually terminated to ensure a fair comparison.

The experimental results are publicly available[1].

A. Results on NativeFlowBench Benchmark

In *NativeFlowBench*, all the projects are Android-based. Therefore, we chose a popular Android static analysis tool, Flowdroid [13], as a baseline in this experiment. JN-SAF [28] and JuCify [17] are two tools directly related to cross-language taint analysis. Both can perform source-sink data flow analysis for Android apps and only require specifying sources and sinks for the experimental projects. Additionally, we integrated CSS into Flowdroid to assess whether our proposed approach can

[1]https://drive.google.com/drive/folders/1KCneXc8NY80pEzk1nvjoiB6LB8-D_F3S?usp=sharing



```
define void @ Java_JavaCaller_native_c_callee(i8* %a, i8* %b)
{
   // pts(a.userName) = pts(b.userName) = {obj_1} // source
   // [obj_2 = χ(obj_1)]
   %cls = call @GetObjectClass(%b)
   %fid = call @GetFieldID(%cls, "username", "Ljava/lang/String;")
   %username_ptr = call @getObjectField(%b, %fid)
   // pts(username_ptr) = {obj_2}
   // [obj_3 = χ(obj_2)]
   call void @Model_setUserInfo(%model_ptr, %username_ptr)
   // pts(model.userInfo) = {obj_3}
   // [obj_4 = χ(obj_3)]
   %userinfo_ptr = call i8* @Model_getUserInfo(%model_ptr)
   // pts(userinfo_ptr) = {obj_4}
   // [μ(obj_4)]
   call void @Log_print(%log_ptr, %userinfo_ptr) // sink
}
```

S_{callee} = {
  calleeMethod: "JavaCaller: void native_c_callee(User, User)",
  From: [arg0.username, arg1.username],
  To: ["Log.print()"],
  callbacks: {}
}

Fig. 9. Callee's specification extraction.

TABLE I
RESULTS OF *NATIVEFLOWBENCH* BENCHMARKS. FD, JS, JC, FD-CSS DENOTES THE RESULT OF FLOWDROID, JN-SAF, JUCIFY AND FLOWDROID-CSS

NativeFlowBench	FD	JS	JC	FD-CSS
icc_javatonative	×	✓	×	×
icc_nativetojava	×	✓	×	✓
native_complexdata	×	✓	×	✓
native_complexdata_stringop	×	∗	×	×
native_dynamic_register_multiple	×	✓	×	✓
native_heap_modify	×	✓	×	×
native_leak	×	✓	×	✓
native_leak_array	×	✓	×	✓
native_leak_dynamic_register	×	✓	×	✓
native_method_overloading	×	×	×	✓
native_multiple_interactions	×	✓	×	✓
native_multiple_libraries	×	✓	×	✓
native_noleak	✓	×	✓	✓
native_noleak_array	✓	∗	✓	✓
native_nosource	×	×	×	×
native_pure	×	✓	×	×
native_pure_direct	×	✓	×	×
native_pure_direct_customized	×	✓	×	×
native_set_field_from_arg	×	✓	✓	✓
native_set_field_from_arg_field	×	✓	×	✓
native_set_field_from_native	×	✓	×	×
native_source	×	✓	✓	×
native_source_clean	∗	×	×	✓
Precsion	8.7%	73.9%	17.4%	60.9%

✓: True Positive; ∗ : False Positive; ×: False Negative.

enhance the capabilities of a single-language static analyzer Flowdroid for cross-language data-flow analysis.

Table I presents the results of four different tools on *NativeFlowBench*, namely FlowDroid (abbreviated as FD), JN-SAF (JS), Jucify (JC), and FlowDroid-CSS (FD-CSS).

FlowDroid, primarily focused on analyzing Java and Android applications, encountered challenges when handling native calls. It did not process these native calls if no manual configuration was provided for their source-sink paths, often leading to unsound modeling, as demonstrated by its result (8.7% precision) in Table I.

FlowDroid-CSS, a hybrid approach that uses FlowDroid with CSS summarization of C/C++ code, showed significant improvement over FlowDroid. By leveraging CSS to provide C/C++ value-flow information, FlowDroid-CSS significantly enhances FlowDroid's capability to handle JNI programs, achieving 60.9% precision.

JN-SAF performs relatively well (73.9% precision) compared to some other tools analyzed given that the benchmark was created by the authors of JN-SAF themselves. For cases where entry points not located in native code, the differing performances of FlowDroid-CSS and JN-SAF on certain benchmarks, including *native_method_overloading*, *native_noleak*, *native_noleak_array*, *native_source_clean*, and *native_complexdata_stringop*, primarily depend on the differences in the data flow analysis algorithms of the two tools. For instance, JN-SAF had a false positive on *native_noleak_array* because it cannot distinguish different indexes of a Java array. Another false positive occurred for *native_complexdata_stringop* due to its imprecise string analysis and overapproximated the contents of the string. In contrast, FlowDroid-CSS did not analyze the content of string variables, resulting in false negatives.

The biggest challenge that FlowDroid-CSS faced is scenarios where the entry points and sources are located in native code. This limitation arises from the requirement of caller information provided by the Java side for further value-flow analysis. Without calling context information, subsequent native code value-flow analysis cannot be completed. As a result, FlowDroid-CSS encountered difficulties in handling certain cases, including *icc_javatonative*, *native_heap_modify*, *native_pure*, *native_pure_direct*, *native_pure_direct_customized*, *native_set_field_from_native*, and *native_source*. These scenarios involve native calls initiated by the C/C++ side, making the caller information unavailable for the Java-based analysis, leading to failure in generating correct specifications.

Jucify, another tool capable of handling Android cross-language value-flow analysis, exhibited limitations (17.4% precision), particularly in scenarios involving purely native functions or native leaks. This constraint is due to Jucify's original design, which did not explicitly cater to such scenarios, where native functions are prevalent in many JNI programs.

Note that the current version of JN-SAF is outdated, as it can only analyze NDKs compiled in specific versions. However, the

TABLE II
RESULTS (R) AND TIME(S) (T) OF *JAVANATIVEBENCH* BENCHMARKS

JavaNativeBench	Tai-e				WALA				Flowdroid				JN-SAF	
Name	CIS		CSS		CIS		CSS		CIS		CSS			
	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time
Java2Java	✓	4.09	✓	4.01	✓	2.98	✓	3.04	✓	6.94	✓	6.12	×	1.12
Java2Native	✓	3.98	✓	4.04	✓	3.03	✓	3.12	✓	7.1	✓	6.14	×	1.09
Java2JavaAlias	✓	4.07	✓	3.88	✓	3.04	✓	3.28	✓	6.09	✓	6.28	×	1.04
Java2JavaNoalias	*	4.02	✓	5.37	*	3.11	✓	4.23	*	6.18	✓	8.24	×	1.12
Java2NativeAlias	✓	3.97	✓	4.15	✓	3.02	✓	3.24	✓	6.06	✓	6.31	×	0.99
Java2NativeNoAlias	*	4.15	✓	5.23	*	3.39	✓	4.21	*	6.37	✓	7.87	×	1.21
Java2JavaFieldAlias	×	4.23	✓	4.08	×	3.22	✓	3.31	×	6.16	✓	6.28	×	1.15
Java2JavaFieldNoAlias	×	4.16	✓	5.12	×	3.27	✓	4.52	×	6.28	✓	8.56	×	1.09
Java2NativeFieldAlias	×	4.08	✓	3.98	×	3.23	✓	3.47	×	6.29	✓	6.51	×	1.11
Java2NativeFieldNoAlias	×	4.21	✓	5.23	×	3.43	✓	4.34	×	6.5	✓	8.29	×	1.12

✓: True Positive; $*$: False Positive; ×: False Negative.

authors of JN-SAF did not specify which NDK versions can be analyzed; the problem was also mentioned by the author as per the GitHub issue [30]. Therefore, the Android APKs used in *NativeFlowBench* are provided by JN-SAF.

In summary, the CSS significantly improves FlowDroid's taint analysis, particularly for native calls in Java code. While limitations exist in handling specific JNI scenarios, CSS shows promise in enhancing the effectiveness of value-flow analysis in the presence of native interactions.

B. Results on JavaNativeBench Benchmark

Apart from *NativeFlowBench*, we designed *JavaNative Bench*, specifically targeting scenarios where Java callers invoke C/C++ functions with aliased and tainted arguments. We employ two additional Java static analysis tools, Tai-e [27] and WALA [21], alongside Flowdroid and JN-SAF, to analyze the value-flow. Similar to FlowDroid, Tai-e and WALA focus solely on the analysis of Java code in the absence of native method models. Additionally, we also have a experiment between CSS and another summary-based specification SemanticSpec [35].

1) CIS and CSS: To validate the effectiveness of CSS, we also conducted a comparative experiment using caller-insensitive specifications (CIS). Unlike CSS, CIS does not consider Java caller information and directly analyzes the C/C++ code to extract relevant value-flows. For this experiment, we use a conservative version of CIS that considers all alias relationships among arguments in native functions. The focus was on determining the existence of paths from tainted arguments to sinks in the value-flows.

Table II presents the results and times for the three tools: Tai-e, WALA, and, Flowdroid, respectively, using CIS, CSS, and JN-SAF applied to the *JavaNativeBench* evaluation. Tai-e-CSS, WALA-CSS and, Flowdroid-CSS demonstrated promising results, successfully analyzing all test cases in the *JavaNativeBench*. Additionally, Tai-e-CIS, WALA-CIS and, Flowdroid-CIS achieved success in four cases, while in the remaining six test cases, there were four false negatives

```
    // Java
J1.  public static void main(String[] args){
J2.      Data d1, d2;
J3.      d1.data = source();
J4.      d2.data = "…";
J5.      String s = nativeTransfer(d1, d2);
J6.      sink(s);
J7.  }

    // C
C1.  char* findSubstring(const char* string) {
C2.      const char* subString = "…";
C3.      return strstr(string, subString);
C4.  }
C5.  jstring Java_Main_nativeTransfer(jobject obj1, jobject obj2) {
C6.      jclass cls= GetObjectClass(obj1);
C7.      jfieldID fd = GetFieldID(cls, "data", "Ljava/lang/String;");
C8.      jstring dataString = GetObjectField(obj1, fd);
C9.      char* originalString = GetStringUTFChars(dataString);
C10.     char* modifiedString = findSubstring(originalString);
C11.     jstring resultString = NewStringUTF(modifiedString);
C12.     jclass cls2 = GetObjectClass(obj2);
C13.     … // Operations on the jobject obj2
C14.     return resultString;
C15. }
```

Fig. 10. Code in *Java2JavaFieldNoAlias*.

and two false positives. Tai-e-CSS/WALA-CSS/Flowdroid-CSS and Tai-e-CIS/WALA-CIS/Flowdroid-CIS both indicate that providing value-flow information about native calls can be beneficial for the value-flow analysis of JNI programs.

The failures observed in CIS (Tai-e-CIS, WALA-CIS, and, Flowdroid-CIS) are primarily attributed to two main reasons. First, without access to calling context information, CIS lacks the necessary context to determine whether certain arguments have alias relationships. To ensure the conservativeness of the analysis results, it must consider all alias scenarios, such as in Fig. 10 where *obj1* and *obj2* have the same type *jobject* (Line C5), which may lead to false positives (*Java2JavaNoAlias* and *Java2NativeNoAlias*). However, before conducting taint analysis on C/C++ code, CSS (Tai-e-CSS, WALA-CSS, and, Flowdroid-CSS) already knew the alias and taint relationships between native call parameters. Therefore, it excludes irrelevant scenarios during the taint analysis, making the generated specifications precise and concise, which reduces the incidence of false positives.

The second reason for failures is when dealing with JNI-provided functions, such as *GetObjectClass(obj1)* in Fig. 10 (Line C6), where CIS lacks caller information to determine the specific class information corresponding to the object *obj* in Java. This lack of class or object information hinders the generation of effective specifications for these scenarios. The failures observed in *Java2JavaFieldAlias*, *Java2JavaFieldNoAlias*, *Java2NativeFieldAlias*, and *Java2NativeFieldNoAlias* can also be attributed to these reasons. However, CSS, informed by the caller specification, get the specific java object information represented by *jobject*, allowing the taint analysis to continue.

The reason JN-SAF failed on all 10 test cases is that it cannot analyze the native code compiled with the newer NDK, and JN-SAF did not specify which version of the NDK used to compile their APKs [30]. In Experiment 1, we used the pre-compiled APKs provided by the authors of JN-SAF, not the ones compiled by us because JN-SAF was unable to properly parse the native code we compiled ourselves.

However, a limitation of CSS is that if the same native method is called in different contexts from Java callers (i.e., under different taint and alias arguments), the same native method needs to be reanalyzed and resummarized to address the different contexts. In contrast, for the same context, CSS can reuse pre-generated callee specifications. This situation is evident from the runtime of *Java2JavaNoalias*, *Java2NativeNoAlias*, *Java2JavaFieldNoalias* and *Java2NativeFieldNoalias*, where, in these four cases, there are multiple calls to the same native method under different contexts.

2) Comparison of Different Specifications: Lee et al. [35] also used a summary-based method for JNI analysis aimed at finding interaction bugs. For clarity, we refer to their summary as SemanticSpec. SemanticSpec first extracts semantic summaries from C code, translates them into Java, and integrates them for whole-program analysis.

Table III compares the performance of the static analyzer Tai-e using SemanticSpec and CSS on *JavaNativeBench*. The comparison includes taint analysis results, runtime (excluding the time to generate SemanticSpec and CSS), and specification quality. Specification quality is evaluated based on the following criteria: (1) a *correct* specification contains the intended value-flows without redundancies; (2) a *redundant* specification includes the intended value-flows but also unnecessary cases; (3) an *incorrect* specification lacks the intended value-flows. We categorized them based on their effects on the accuracy of the taint analysis results and the analysis time. Correct specifications produced accurate results with minimal analysis time by capturing intended value-flows without redundancies. Redundant specifications yielded correct results but had longer analysis times due to including unnecessary cases. Incorrect specifications led to erroneous taint analysis results by missing intended value-flows.

TABLE III
COMPARISON OF QUALITY, TAINT ANALYSIS RESULTS AND RUNTIME BETWEEN SEMANTICSPEC AND CSS

JavaNativeBench	SemanticSpec			CSS		
Name	Quality	Result	Time	Quality	Result	Time
Java2Java	O	✓	2.23	O	✓	2.20
Java2Native	O	✓	2.21	O	✓	2.19
Java2JavaAlias	O	✓	2.19	O	✓	2.20
Java2JavaNoalias	O	✓	2.20	O	✓	2.22
Java2NativeAlias	O	✓	2.22	O	✓	2.22
Java2NativeNoAlias	O	✓	2.23	O	✓	2.21
Java2JavaFieldAlias	R,F	×	2.39	O	✓	2.21
Java2JavaFieldNoAlias	R,F	×	2.44	O	✓	2.20
Java2NativeFieldAlias	O	✓	2.23	O	✓	2.22
Java2NativeFieldNoAlias	O	✓	2.21	O	✓	2.21

O: Correct specification; R: Redundant specification; F: Incorrect specification; ✓: True Positive; ×: False Negative.

Like CIS, SemanticSpec extracts C code semantics without caller context and translates it into Java for analysis, potentially retaining redundant information. For example, in *Java2JavaFieldNoAlias* of Fig. 10, SemanticSpec is conservative and hence does not distinguish between tainted *obj1* and untainted *obj2*, leading to specifications that include unnecessary semantics about *obj2*. This lack of caller context can result in redundant specifications and slightly longer analysis times compared to CSS, as seen in *Java2JavaFieldAlias* and *Java2JavaFieldNoAlias* of Table III. Additionally, SemanticSpec generates summaries for the target guest language and struggles with some complex C features and functions. For instance, in the function *findSubstring()* of Fig. 10, it imprecisely models the return value as a new, unrelated string, leading to undetected taint flows in cases like *Java2JavaFieldAlias* and *Java2JavaFieldNoAlias*. CSS generates concise caller-sensitive specifications without unnecessary cases by leveraging the taint and alias information of parameters provided by the caller. The experimental results demonstrate that CSS detects all tainted value flows in our benchmarks.

The results of the above experiments highlight the significance of incorporating CSS in JNI analysis. By leveraging caller specification information, CSS can effectively track alias relationships and handle JNI functions, thereby improving the precision and accuracy of value-flow analysis for JNI programs.

By incorporating CSS into existing Java static analyzers (e.g., FlowDroid, Tai-e and, WALA), the existing established static tools and their algorithms can be leveraged for cross-language analysis. This integration approach enables the static analyzers to take advantage of their existing value-flow analysis techniques in the Java domain and supplement them with the value-flow information from the native (C/C++) world, effectively bridging the gap between the two languages.

C. Results on Real-World JNI Programs

In addition to conducting experiments with *NativeFlowBench* and *JavaNativeBench*, we further tested real JNI programs to validate the accuracy of CSS in extracting value-flow information from native code. For this evaluation, we selected

TABLE IV
VALUE-FLOW ANALYSIS RESULTS OF THREE STATIC ANALYZERS JN-SAF, JUCIFY AND, FLOWDROID-CSS ON TEN ANDROID JNI APPLICATIONS. THE METRICS INCLUDE THE TOTAL NUMBER OF LINES OF CODE IN THOUSANDS (LOC), THE PERCENTAGE OF JAVA CODE (JAVA), THE PERCENTAGE OF C/C++ CODE (C/C++), THE NUMBER OF TRUE POSITIVE RESULTS (RECALL) OUT OF THE TOTAL TRUE POSITIVE RESULTS (TOTAL) IN EACH APPLICATION, AND THE PRECISION OF THE TOOLS (R/T, RECALL-TO-TOTAL). THE TABLE ALSO REPORTS THE TIME TAKEN IN JAVA AND C/C++ SEPARATELY IN SECONDS (TIME) FOR EACH TOOL

					JN-SAF				JuCify				FlowDroid-CSS			
APP	Loc(K)	Java(%)	C/C++(%)	Total	Recall	R/T(%)	Time(s)		Recall	R/T(%)	Time(s)		Recall	R/T(%)	Times	
							Java	C/C++			java	C/C++			Java	C/C++
android-aes-jni	1.3	12.6	86.8	1	0	0	5.7	-	0	0	1.0	194.4	1	100	3.7	0.2
LoggingApp	1.4	36.2	37.3	4	0	0	4.4	-	0	0	4.0	372.21	4	100	3.1	0.4
samba-documents-provider	11	81.2	16.0	1	0	0	14.3	-	1	100	21.0	37.0	1	100	10.9	2.3
base64encode	12	27.1	31.6	2	0	0	14.5	1.3	0	0	2.0	304.4	0	0	8	1.2
AndroidUn7zip	16	4.9	94.8	2	0	0	20.7	-	0	0	8.0	367.9	2	100	20.7	5.6
VoiceChange	26	60.5	23.5	6	0	0	23.8	-	0	0	25.0	444.8	6	100	23.9	6.4
FairEmail	63	55.4	38.5	1	0	0	69.2	-	1	100	22.0	492.7	1	100	28.9	10.2
Log4a	85	38.5	32.3	1	0	0	52.8	-	0	0	32.0	143.3	1	100	41.4	15.4
tracker-control-android	349	23.7	23.7	1	0	0	203.5	-	0	0	21.0	300.4	1	100	40.6	16.7
oboe	160	35.1	61.1	1	0	0	32	-	0	0	12.0	992.5	1	100	55	20.3
Avg	**72.5**	**37.5**	**44.6**	**2**	**0**	**0**	53.9	0.13	**0.2**	**20**	17.9	364.9	**1.8**	**90**	**23.6**	**7.9**

ten JNI-related Android projects from GitHub for our analysis using three tools: JN-SAF, JuCify and, FlowDroid-CSS.

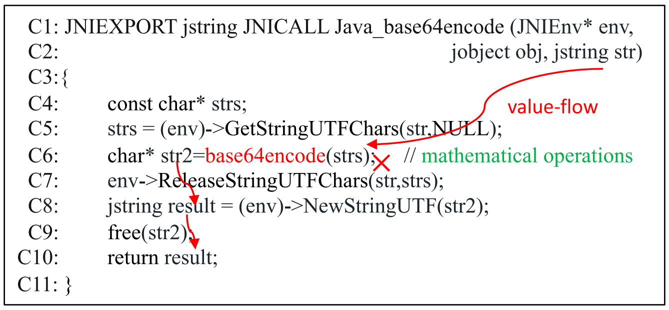
1) Implementation: We selected 10 repositories from around 50 GitHub candidates identified with “JNI” and “NDK” keywords, focusing on three criteria: scale, diversity, and path length. Repositories were classified by size as small (1-15 KLoC), medium (15-100 KLoC), or large (over 100 KLoC), with our choices including 4 small, 4 medium, and 2 large. For diversity, we assigned one repository each to categories like “Security and Encryption”, “Logging”, and “Email” and selected 3 for “File/Data Management” and 2 each for “Audio/Image Processing” and “Privacy/Control”. Path lengths were categorized from short (1-3 hops), medium(4-8 hops) to long (9+ hops), with selections of 4 short, 4 medium, and 2 long paths.

For the selection of sources, we primarily chose functions that involve sensitive data, such as *getDeviceId()*. As for sinks, we selected functions that could potentially leak sensitive data, such as *__android_log_print()*, which writes data to files. We limited the sources and sinks to two scenarios: (1) Sources in Java code, and sinks in native code; (2) Sources in Java code, and sinks in Java code. In the second scenario, the data generated by the source method in Java code flows into the native code and then returns to the Java world. The reason we did not consider the other two scenarios with the sources in native code, namely (1) Sources in native code, the sinks in Java code; (2) Sources in native code, the sinks in native code, is because before processing the value-flow in native code, the Java side needs to provide the caller specifications. In other words, Java is the host language, and native code is the guest language, so the Java side needs to serve as the starting point for source generation in value-flow analysis.

Before the experiments, we manually inspected the code of these Android projects to determine value-flow paths between sources and sinks. With these value-flow paths as ground truth, we can more clearly compare the performance of the JN-SAF, JuCify and FlowDroid-CSS. Table IV summarizes the value-flow analysis results of ten projects. The project characteristics include the name of the application (APP), the total number of lines of code in thousands (Loc), the percentage of code written in Java and C/C++ languages, and the total number of value-flow paths in each application (Total). The table also provides information on the performance of JN-SAF, JuCify and FlowDroid-CSS. For each tool, metrics such as “Recall”, Recall-to-Total (“R/T”) ratio, and execution “Time” (in seconds) are presented. “Recall” measures the effectiveness of the tool in identifying the correct value-flow paths, while The “R/T” ratio indicates the precision of the tool, and the “Time” shows the speed of the tool in processing Java and C/C++ code separately in the application.

2) Performance of JN-SAF, JuCify and FlowDroid-CSS: From the experimental results, it can be seen that value-flow analysis with CSS can correctly detect the source-sink paths in nine out of the ten projects, while JuCify succeeded in only two out of the ten projects. But JN-SAF failed for all projects. The failure of JN-SAF is attributed to its latest version being outdated in terms of analyzing native code. This prevents it from correctly locating or analyzing the.so files and conducting the analysis (The analysis time for C/C++ of JN-SAF in Table IV is “-”). Different NDK versions may employ various optimizations and compilation techniques, which can significantly impact the final results. The applications used in Experiment 3 were compiled by ourselves, and the projects used in Experiment 1 were compiled by the authors of JN-SAF. Moreover, JN-SAF did not specify the versions of NDK that can be analyzed [30].

The main reason for Jucify’s failure is that the value-flow analysis in C/C++ code involves pure native functions. The current JuCify can only reconstruct native calls about Java-C interoperation parts but does not consider pure native functions, meaning it is unable to handle native functions and native leaks. Hence, the scope of JuCify’s value-flow analysis is limited, as it cannot identify value-flow paths where sensitive information is leaked through pure C/C++ functions, which is a common scenario in real-world data breaches. Additionally, JuCify uses Angr for symbolic execution to analyze C/C++ code, but



```
C1: JNIEXPORT jstring JNICALL Java_base64encode (JNIEnv* env,
C2:                                         jobject obj, jstring str)
C3:{
C4:       const char* strs;
C5:       strs = (env)->GetStringUTFChars(str,NULL);
C6:       char* str2=base64encode(strs);   // mathematical operations
C7:       env->ReleaseStringUTFChars(str,strs);
C8:       jstring result = (env)->NewStringUTF(str2);
C9:       free(str2);
C10:      return result;
C11: }
```

Fig. 11. *base64encode* example.

symbolic execution faces the issue of path explosion. The results from Table IV indicate that the time JuCify takes to analyze native code using Angr is at nearly 46 times that of FlowDroid-CSS. This is even when JuCify only focuses on JNI-provided functions and but not the pure C/C++ code. If all logic within the native code were considered, the analysis time for the native code might continue to increase. On the other hand, JuCify cannot accurately reconstruct native code behavior because it uses heuristics to handle native code behavior, rather than tracking value-flow based on the native source code, causing a loss of precision.

As for FlowDroid-CSS, since we use native source code to conduct value-flow analysis, it does not miss the value-flow information of native code like JuCify does. By employing SVFG for sparse value-flow analysis and constructing the def-use chain of variables, it also avoids the path explosion problem encountered when using symbolic execution to analyze native code in Jucify. However, CSS still failed to detect one value-flow path in the project *base64encode*.

In project *base64encode*, a java-to-java value-flow path needs to be detected. The source data *str*, originating from the Java side, passes through the C function *Java_base64encode()* (Fig. 11). Once encoded, *str* returns to the Java side as a return value. However, the *base64encode()* function located at Line C6 performs mathematical and encoding operations on the *str* variable. These operations represent an implicit value-flow originating from the argument and reaching its return value within *base64encode()*, while SVFG's capabilities are limited to constructing explicit value-flows for the string variables (e.g., *str*). Therefore, the value-flow path is disconnected at the *base64encode()* function (red curved arrow), making it impossible to generate a complete java-to-java value-flow path. Tracking implicit flows is a semantic problem [36] and the common approach is to manually write a specification for implicit flows to define the value-flows. For example, we can configure the side-effect of *base64encode()* as a store operation of **str2 = strs*. Then CSS can accurately identify the value-flow path in this project. However, in the experiment, we did not summarize the side-effects of the functions involved between native sources and sinks for the sake of fairness in comparison.

In summary, value-flow analysis with CSS outperforms JuCify in terms of successfully detecting paths with a higher success rate and a lower average analysis time.

V. Limitations

Although the CSS in the experiments improves the accuracy of data propagation tracking in JNI programs by considering cross-language context from Java callers. There are still limitations in our approach. One limitation is as shown by the failed project *base64encode* in Experiment 3. Although there is an implicit value-flow path from the parameter to the return value, this path was not discovered during the experiment due to the involvement of numerous string operations and mathematical computations. While we can summarize the side effects of these functions containing implicit value-flow paths, through manual summarization and advanced string analysis [37].

The second limitation lies in the resummarization of callee specifications when cross-language context changes. Because CSS supports the precise summarization of the behavior of native functions under a cross-language context. If developers update the caller code, which may affect or change the cross-language context, we need to re-summarize the called native functions. This is necessary to generate a new CSS based on the updated caller specification. Subsequently, the newly generated CSS must be incorporated into the native function specifications library. The good side is that our summarization procedure is incremental in nature. This means that our approach involves augmenting the existing CSS with additional information, without the removal of any pre-existing summarization within the CSS. As a result, our approach facilitates incremental summarization, allowing for gradual enhancements in situations where the caller code, such as Java code, undergoes changes.

In Android app development, JNI calls are typically initiated from the Java side, enabling methods in Java code to invoke functions in native code through JNI for various interactions. However, in certain cases, particularly when optimizing performance or implementing low-level functionality, JNI calls can also originate from the C/C++ native code side to invoke Java methods. Value-flow analysis is intractable when the entry point and source are in native code without information from the Java-side caller method, as shown in Experiment 1. In such cases, information from the Java-side caller method is necessary for generating caller-sensitive specifications and conducting accurate value-flow analysis. Without callers' information, it is difficult to identify the correct value-flow paths in the program and generate precise specifications. The absence of caller information can lead to incomplete or inaccurate results in the value-flow analysis, making it difficult to detect potential security vulnerabilities or performance issues. Future work will aim to overcome these limitations to improve the accuracy of value-flow analysis for JNI programs.

VI. Related Work

Being a widely adopted programming language, Java has attracted considerable scholarly interest due to its distinctive features, including its capability for cross-language interaction. Tan et al. [38] carried out an empirical security study on JNI bugs in JDK's native code, providing remedies and insights into these bugs. Li and Tan [39], [40] developed approaches to identify bugs in situations within JNI programs. Kondoh and

Onodera [41] concentrated on four types of JNI-specific errors that native compilers do not capture. Shi et al. [24] utilized specifications to summarize information flow in Android native libraries to support taint analysis in Android apps. However, these studies primarily focus on native code and do not consider the interaction at the boundary between Java and native code.

Tan et al. [15] extend Java Virtual Machine Language (JVML) language for C code to assist static analysis for JNI programs. JN-SAF [28] proposed by Wei et al. first constructs a precise topologically sorted call graph for JNI programs. This framework alternates between Java and C/C++ static analyzers, ensuring accurate value-flow tracking throughout the application. JN-SAF's scalability can be constrained by path explosion problems arising from Angr's symbolic execution. Furthermore, it might yield inconsistent outcomes when handling binary codes with different optimization levels [30]. Samhi et al. [42] aimed at unifying bytecode and native code to facilitate comprehensive static analysis of Android apps. They introduced JuCify as a significant step towards this goal, which generates a native call graph that is merged with the bytecode call graph using links obtained through symbolic execution. However, a key limitation of JuCify it that it is hard to reconstruct native function behavior with high precision, impacting the effectiveness of the analysis in many cross-language context scenarios. Lee et al. [35] also used a summary-based method for JNI analysis, where the approach first extracts semantic summaries from C code and then integrates them with Java code for a whole-program analysis. Our work differs from theirs in that CSS is caller-sensitive, acquiring information from the Java callers to yield a compact and precise summarization. Hence our approach can be seen as a precision-driven enhancement on top of their methods if all source code is available.

Aside from cross-language static analysis of JNI programs, there is research focusing on static analysis involving other languages. Lee et al. [42] introduced HybriDroid for executing static analysis between Java and JavaScript. Monat et al. [16] and Li et al. [8] developed static analyzers for identifying runtime errors and reference count adjustments in Python programs using C extensions. Furr et al. [43], [44], [45] performed type safety checking targets OCaml's FFI to C and JNI. Li et al. [9] compiled Rust and C into LLVM IR and employed abstract interpretation to detect cross-language memory management issues in Rust.

VII. Conclusion

JNI programming makes cross-language software development more flexible and efficient. However, it is challenging to develop static analysis that can accurately analyze cross-language programs, such as JNIs for Java and C/C++ due to complicated semantics for language interaction. We propose a new approach to specification extraction known as caller-sensitive specification. Compared to using conservative specifications, this approach takes the cross-language calling context of native C/C++ functions into account to generate more accurate and compact specifications (side-effects) of the native functions being called from Java. By focusing on the side-effects that are sensitive to the cross-language calling context, the CSS approach can reduce the number of unnecessary specifications. In addition, the CSS can be easily integrated with Java single-language static analysis tools, enabling cross-language analysis with no or light code modifications. The results of the experiment demonstrate that CSS is capable of precisely tracking the value-flows in JNI programs over several recent tools. Considering the challenges in developing an effective static analyzer for JNI programs, the CSS approach presented in this paper offers valuable insights for enhancing static analysis tools in JNI contexts.

Acknowledgment

The authors thank the anonymous reviewers for their feedback on earlier drafts of this paper.

References

[1] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Reading, MA, USA: Addison-Wesley, 1999.

[2] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, vol. 22, 2022, pp. 1–17.

[3] C. Li et al., "Cross-language android permission specification," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.,* 2022, pp. 772–783.

[4] L. Chaoran, "Vulnerability detection in android," Ph.D. dissertation, Swinburne Univ. Technol., Melbourne, VIC, Australia, 2023.

[5] Y. He et al., "A systematic study of android non-SDK (hidden) service API security," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 1609–1623, Mar./Apr. 2023.

[6] A.-D. Schmidt et al., "Smartphone malware evolution revisited: Android next target?" in *Proc. 4th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 1–7.

[7] S. Kumar and S. K. Shukla, "The state of android security," *Cyber Security in India: Education, Research and Training*, pp. 17–22, Singapore: Springer, 2020.

[8] S. Li and G. Tan, "Finding reference-counting errors in Python/C programs with affine analysis," in *Proc. Object-Oriented Program. 28th Eur. Conf. (ECOOP)*, Uppsala, Sweden, vol. 8586, R. E. Jones, Ed., Berlin, Heidelberg: Springer-Verlag, 2014, pp. 80–104, doi: https://doi.org/10.1007/978-3-662-44202-9_4.

[9] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Detecting cross-language memory management issues in rust," in *Proc. Comput. Secur. 27th Eur. Symp. Res. Comput. Secur. (ESORICS)*, Copenhagen, Denmark, vol. 13556, V. Atluri, R. D. Pietro, C. D. Jensen, and W. Meng, Eds., Cham, Switzerland: Springer-Verlag, 2022, pp. 680–700, doi: 10.1007/978-3-031-17143-7_33.

[10] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of android malware behaviors," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2015, pp. 1–15.

[11] Z. Zhong, J. Liu, D. Wu, P. Di, Y. Sui, and A. X. Liu, "Field-based static taint analysis for industrial microservices," in *Proc. 44th Int. Conf. Softw. Eng., Softw. Eng. Pract.*, 2022, pp. 149–150.

[12] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proc. 25th Int. Conf. Compiler Construction*, Barcelona, Spain, A. Zaks and M. V. Hermenegildo, Eds., New York, NY: ACM, 2016, pp. 265–266, doi: 10.1145/2892208.2892235.

[13] S. Arzt et al., "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI '14)*, Edinburgh, United Kingdom, M. F. P. O'Boyle and K. Pingali, Eds., New York, NY: ACM, 2014, pp. 259–269, doi: 10.1145/2594291.2594299.

[14] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of Java enterprise applications: Frameworks and caches, the elephants in the room," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2020, pp. 794–807, doi: 10.1145/3385412.3386026.

[15] G. Tan and G. Morrisett, "Ilea: Inter-language analysis across Java and C," in *Proc. 22nd Annu. ACM SIGPLAN Conf. Object-Oriented

Program., Syst., Lang., Appl. (OOPSLA), Montreal, QC, Canada, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds., New York, NY: ACM, 2007, pp. 39–56, doi: 10.1145/1297027.1297031.

[16] R. Monat, A. Ouadjaout, and A. Miné, "A multilanguage static analysis of Python programs with native C extensions," in *Proc. Static Anal. 28th Int. Symp. (SAS)*, Chicago, IL, USA, vol. 12913, C. Dragoi, S. Mukherjee, and K. S. Namjoshi, Eds., Springer-Verlag, 2021, pp. 323–345, doi: 10.1007/978-3-030-88806-0_16.

[17] J. Samhi et al., "JuCify: A step towards android code unification for enhanced static analysis," in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng., (ICSE)*, Pittsburgh, PA, USA, New York, NY: ACM, 2022, pp. 1232–1244, doi: 10.1145/3510003.3512766.

[18] "JLang: An LLVM backend for the polyglot compiler." GitHub. Accessed: Oct. 23, 2023. [Online]. Available: https://polyglot-compiler.github.io/JLang

[19] S. Buro, R. L. Crole, and I. Mastroeni, "On multi-language abstraction - towards a static analysis of multi-language programs," in *Proc. Static Anal. 27th Int. Symp. (SAS),* Virtual Event, vol. 12389, D. Pichardie and M. Sighireanu, Eds., Springer-Verlag, 2020, pp. 310–332, doi: 10.1007/978-3-030-65474-0_14.

[20] "Facebook," Infer. [Online]. Accessed: Oct. 23, 2023. Available: https://fbinfer.com

[21] Wala. Accessed: Oct. 23, 2023. [Online]. Available: https://github.com/wala/WALA

[22] P. Avgustinov, O. De Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented queries on relational data," in *Proc. 30th Eur. Conf. Object Oriented Program. (ECOOP)*, Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016, pp. 21–225.

[23] H. Zhu, T. Dillig, and I. Dillig, "Automated inference of library specifications for source-sink property verification," in *Proc. Program. Lang. Syst. 11th Asian Symp. (APLAS)*, Melbourne, VIC, Australia, vol. 8301, C. Shan, Ed., Cham, Switzerland: Springer-Verlag, 2013, pp. 290–306, doi: 10.1007/978-3-319-03542-0_21.

[24] C. Shi, C. C.-C. Cheng, and Y. Guan, "LibDroid: Summarizing information flow of android native libraries via static analysis," *Forensic Sci. Int., Digit. Investigation*, vol. 42, 2022, Art. no. 301405.

[25] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Proc. Detection Intrusions Malware, Vulnerability Assessment 14th Int. Conf. (DIMVA)*, Bonn, Germany, vol. 10327, M. Polychronakis and M. Meier, Eds., Springer-Verlag, 2017, pp. 252–276, doi: 10.1007/978-3-319-60876-1_12.

[26] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in DroidSafe," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, California, USA, Reston, VA, USA: The Internet Society, 2015. Accessed: Oct. 23, 2023. [Online]. Available: https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

[27] T. Tan and Y. Li, "Tai-e: A static analysis framework for Java by harnessing the best designs of classics," 2022, *arXiv:2208.00337*.

[28] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.,* Toronto, ON, Canada, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., New York, NY: ACM, 2018, pp. 1137–1150, doi: 10.1145/3243734.3243835.

[29] Y. Shoshitaishvili et al., "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2016, pp. 138–157.

[30] arguslab, "Nativeflowbench github issue," GitHub. Accessed: Oct. 23, 2023. [Online]. Available: https://github.com/arguslab/NativeFlowBench/issues/1

[31] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Trans. Softw. Eng.*, vol. 40, no. 2, pp. 107–122, 2014, doi: 10.1109/TSE.2014.2302311.

[32] N. D. Jones and S. S. Muchnick, "Flow analysis and optimization of LISP-like structures," in *Proc. 6th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1979, pp. 244–256.

[33] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, Minneapolis, MN, USA, New York, NY: ACM, 2012, pp. 254–264, doi: 10.1145/2338965.2336784.

[34] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich, "Effective representation of aliases and indirect memory operations in SSA form," in *Proc. Int. Conf. Compiler Construction*, Berlin, Heidelberg: Springer-Verlag, 1996, pp. 253–267.

[35] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng., (ASE)*, Melbourne, Australia, Piscataway, NJ, USA: IEEE Press, 2020, pp. 127–137, doi: 10.1145/3324884.3416558.

[36] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'Em, can't live without 'Em," in *Proc. Inf. Syst. Secur., 4th Int. Conf. (ICISS)*, Hyderabad, India, vol. 5352, R. Sekar and A. K. Pujari, Eds., Springer-Verlag, 2008, pp. 56–70, doi:10.1007/978-3-540-89862-7_4.

[37] G. Costantini, P. Ferrara, and A. Cortesi, "Static analysis of string values," in *Proc. Formal Methods Softw. Eng., 13th Int. Conf. Formal Eng. Methods (ICFEM)*, Durham, U.K. Springer-Verlag, 2011, pp. 505–521.

[38] G. Tan and J. Croft, "An empirical security study of the native code in the JDK," in *Proc. 17th USENIX Secur. Symp.*, San Jose, CA, USA, P. C. van Oorschot, Ed., San Jose, CA, USA: USENIX Association, 2008, pp. 365–378. [Online]. Available: http://www.usenix.org/events/sec08/tech/full_papers/tan_g/tan_g.pdf

[39] S. Li and G. Tan, "Exception analysis in the Java native interface," *Sci. Comput. Program.*, vol. 89, no. PART C, pp. 273–297, Sep. 2014, doi: 10.1016/j.scico.2014.01.018.

[40] S. Li and G. Tan, "JET: Exception checking in the Java native interface," in *Proc. 26th Annu. ACM SIGPLAN Conf. Object Oriented Program., Syst., Lang., Appl. (OOPSLA)* Portland, OR, USA, C. V. Lopes and K. Fisher, Eds., New York, NY: ACM, 2011, pp. 345–358, doi: 10.1145/2048066.2048095.

[41] G. Kondoh and T. Onodera, "Finding bugs in Java native interface programs," in *Proc. ACM/SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, Seattle, WA, USA, B. G. Ryder and A. Zeller, Eds., New York, NY: ACM, 2008, pp. 109–118, doi: 10.1145/1390630.1390645.

[42] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for android hybrid applications," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Singapore, D. Lo, S. Apel, and S. Khurshid, Eds., New York, NY: ACM, 2016, pp. 250–261, doi: 10.1145/2970276.2970368.

[43] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, Chicago, IL, USA, V. Sarkar and M. W. Hall, Eds., New York, NY: ACM, 2005, pp. 62–72, doi: 10.1145/1065010.1065019.

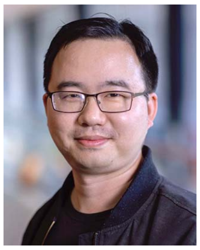
[44] M. Furr and J. S. Foster, "Polymorphic type inference for the JNI," in *Program. Lang. Syst., 15th Eur. Symp. Program. (ESOP)* Vienna, Austria, vol. 3924, P. Sestoft, Ed., Springer-Verlag, 2006, pp. 309–324, doi: 10.1007/11693024_21.

[45] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 4, pp. 18: 1–18:63, 2008, doi: 10.1145/1377492.1377493.

Shuangxiang Kan received the M.Sc. degree from Soochow University, in 2021. He is currently working toward the Ph.D. degree with the University of New South Wales. His research interests include static and dynamic program analysis and program verification.

Yuhao Gao is currently working toward the Ph.D. degree with the Faculty of Engineering and Information Technology, University of Technology Sydney. His research interests include mobile app analysis and program analysis, mainly underground mobile app analysis.

Zexin Zhong is pursuing a Ph.D. degree in static program analysis, cyber security, and micro-services security, under Professor Yulei Sui at the University of Technology Sydney, Australia. Zexin's work focuses on improving static taint analysis for software security. He has also interned at Ant Group, where he developed compositional taint Analysis for large-scale enterprise systems, crucial for protecting user data security.



Yulei Sui is a Scientia Associate Professor at the University of New South Wales, specializing in program analysis, secure software engineering, and machine learning. His research aims to enhance software reliability and security through static analysis and verification frameworks. Currently, he focuses on integrating programming languages, natural languages, and machine learning for trustworthy machine learning and software bug detection using data mining and deep learning.