Interactive Cross-Language Pointer Analysis For Resolving Native Code in Java Programs

Chenxi Zhang[†], Yufei Liang[†], Tian Tan[†*], Chang Xu[†], Shuangxiang Kan[§], Yulei Sui[§], Yue Li[†*]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, China
[§]University of New South Wales, Australia
{dz1733024,602024330013}@smail.nju.edu.cn, {tiantan, changxu}@nju.edu.cn
{shuangxiang.kan, y.sui}@unsw.edu.au, yueli@nju.edu.cn

***Abstract*—Java offers the Java Native Interface (JNI), which allows programs running in the Java Virtual Machine to invoke and be manipulated by native applications and libraries written in other languages, typically C. While JNI mechanism significantly enhances the Java platform's capabilities, it also presents challenges for static analysis of Java programs due to the complex behaviors introduced by native code. Therefore, effectively resolving the interactions between Java and native code is crucial for static analysis. In this paper, we introduce JNIFER, the first interactive cross-language pointer analysis for resolving native code in Java programs. JNIFER integrates both Java and C pointer analyses, equipped with advanced native call and JNI function analyses, enabling the simultaneous analysis of both Java and native code. During the analysis of cross-language interactions, the two analyzers interact with each other, constructing cross-language points-to relations and call graphs, thereby approximating the runtime behavior at the interaction sites. Our evaluation shows that JNIFER outperforms state-of-the-art approaches in terms of soundness while maintaining high precision and comparable efficiency, as evidenced by extensive experiments on OpenJDK and real-world Java applications.**

***Index Terms*—Java Native Interface, Native Code, Pointer Analysis, Cross-Language Analysis**

I. Introduction

In Java development, native method calls play a crucial role in leveraging existing native libraries, facilitating direct interaction with the underlying operating system, optimizing application performance, etc. Unlike ordinary Java methods, which are declared and implemented in Java, native methods are declared in Java but implemented in other languages (primarily C code, referred to as native code). The execution of native methods involves not only intra-language communication within Java and C but also complex cross-language interactions. These interactions can be categorized into two types: first, invoking C functions from Java via native method calls (denoted Java $\rightarrow$ C); second, using Java Native Interface (JNI) functions in C to call Java methods, create Java objects, access Java fields, and more (denoted C $\rightarrow$ Java).

While native code is highly useful in development, it poses significant challenges for static analysis [1]. The key to resolving native code lies in inferring behaviors at cross-language interaction sites, which involves identifying the target native function of a native method call for J $\rightarrow$ C interactions and analyzing the behaviors of various JNI function calls for C $\rightarrow$ J interactions. These tasks are complex due to the intricate nature of cross-language interactions and language differences between Java and C code [2]. Despite these challenges, resolving native code is valuable for static analysis, providing more complete information about program behaviors and benefiting client analyses such as security analysis [3]–[5] and bug detection [6]–[9].

To comprehensively and precisely resolve native code in Java, one of the most straightforward and effective approaches is to utilize pointer analysis, a fundamental technique underlying virtually all other analyses [10]. Despite the development of numerous pointer analysis methods over the past 40 years for C and Java individually [16]–[22], no interactive pointer analysis exists to analyze Java and C simultaneously. The significant syntax and semantic differences between the languages, along with their distinct pointer analysis algorithms, complicate this approach and make it highly challenging. Consequently, existing state-of-the-art works, JN-Sum [11] and Native-Scanner [12], circumvent cross-language pointer analysis to resolve native code in Java. Specifically, JN-Sum focuses on a limited subset of C language constructs, transforming them into Java code, and then analyzing the Java code. However, by focusing on only partial C behaviors, this method fails to resolve many interactions between C and Java, missing numerous native code behaviors. Additionally, the language transformation process is complex, significantly affecting the analysis's robustness and making it difficult to adopt in practice. Conversely, Native-Scanner extracts string constants from C code and, when matched with signatures in Java code, conservatively assumes those Java methods are reachable to mimic the side effects of native code. This highly conservative approach results in significant imprecision.

In this paper, we tackle the above challenge directly, by introducing JNIFER, the first interactive cross-language pointer analysis to effectively resolve Java native code.

a) Method: JNIFER integrates both Java and C pointer analyses, allowing the results of one to be recognized and incorporated into the other. This facilitates the resolution of native calls (Java$\rightarrow$C) and JNI function calls (C$\rightarrow$Java) simultaneously. To achieve this, we propose a cross-language model that uniformly represents the results of both Java and

* Corresponding authors.

C pointer analyses. Based on this model, we introduce two analyzers to on-the-fly resolve the cross-language interactions of Java→C and C→Java respectively, by utilizing pointer analysis while adhering strictly to the JNI specification. To maximally unleash the power of pointer analysis, JNIFER instantiates its methodology by incorporating Tai-e [13] and SVF [14], two state-of-the-art pointer analysis frameworks, as its pointer analysis components for Java and C, respectively.

b) Results: We conducted extensive experiments to compare JNIFER with state-of-the-art tools, JN-Sum and Native-Scanner, to examine its effectiveness. These tools were used to analyze OpenJDK, which extensively uses native code, as well as all real-world Java applications used in recent research [11], [12]. The results indicate that JNIFER significantly outperforms existing state-of-the-art methods:

- *Soundness:* We measured soundness by comparing how many dynamic cross-language interactions in our experiments could be recalled by the analyzers. JNIFER achieved an average recall of 91.6%, significantly higher than JN-Sum (34.6%) and Native-Scanner (62.9%).
- *Precision:* We randomly selected various interaction sites and examined the analysis results of the tools. JNIFER maintained excellent precision at 99.1%, whereas Native-Scanner's precision was only 25.0%. Although JN-Sum reached a precision of 100%, it failed to resolve a significant number of cross-language interaction sites.
- *Efficiency:* JNIFER proved to be the fastest in our experiments, achieving a speedup of 2.4x over JN-Sum and 1.2x over Native-Scanner.

We will release JNIFER as an open-source tool and submit an artifact to AEC for reproducing all experimental results.

II. Motivating Example

In this section, we use an example (simplified from real-world native code in OpenJDK) to motivate our methodology. First, we introduce the basic concepts of native code and the example (Section II-A). Next, we use the example to illustrate the limitations of existing works (Section II-B). Finally, we present our insight (Section II-C).

A. Introduction to Native Code and The Example

To use native code, a developer must declare a native method using the `native` keyword. When this method is invoked, the JVM resolves the target native function written in C (through name matching or runtime registration, details of which are omitted here) and enters the function. This type of interaction, where a native method call in Java leads to a native function, is termed a J → C interaction.

Within a native function, a developer can invoke JNI functions provided by the JVM to call Java methods, access Java fields, and interact with Java objects. This type of interaction, where a JNI function call in C operates on Java elements, is termed a C → J interaction.

Fig. 1 presents an example simplified from real-world native code in OpenJDK, where starting characters J and C denote the line numbers for Java and C code, respectively. The example includes seven cross-language interactions, marked by circled numbers, ordered by their occurrence at runtime. Specifically, ①②④ are J → C interactions, while ③⑤⑥⑦ are C → J interactions. Due to the diversity of C → J interactions, their side effects are explained in comments above the JNI function calls to aid understanding.

In this example, the Java code consists of two classes, `Image` and `ColorModel`. `Image` declares three native methods (J13-J15) and calls them in `initImage`. The right half of Fig. 1 shows the corresponding native functions. Note that according to [2], the first argument of a native function must be `JNIEnv`, which is omitted here for simplicity.

To fully understand such a program, it is essential to resolve all cross-language interactions to obtain complete program behaviors, including control and data flows. For example, at ③, the native function `Java_Image_initNative` calls `SetIntField` to update the Java field `image.rgb` with the value of `rgb`. This indicates a data flow from `rgb` (in C) to `image.rgb` (in Java). However, if the C code retrieves an untrusted value and sets it to a security-sensitive field in Java via `SetField`, it could introduce a security vulnerability. If the static analyzer fails to resolve the cross-language interaction, such a vulnerability may be missed.

B. Limitations of Existing Works

Since the control and data flows of Java programs with native code span both Java and C sides, a natural approach to analyzing such programs is to perform cross-language analysis. However, this is challenging due to the intricate nature of cross-language interactions and the differences between Java and C. Existing works attempt to circumvent this difficulty in resolving native code, but they introduce their own limitations.

Native-Scanner [12] resolves native code in a highly conservative manner, resulting in significant imprecision. It extracts string constants from native code to construct partial Java method signatures, assuming that any Java method matching these partial signatures will be called from C.

For example, for line C22 of Fig. 1, Native-Scanner would extract partial signature `<init>()V`, and consider all no-argument methods with a return type of `void` to be potential targets invoked by native code. Additionally, it does not support field access analysis in native code, thus missing the data flow from ③ to `image.rgb`. Even if it were to handle field accesses, its methodology would likely introduce a substantial number of false positives.

JN-Sum [11] adopts a different approach to analyzing Java programs with native code by transforming C code (related to JNI function calls) into Java code and feeding it to the Java analysis. While this approach avoids the challenges of interactive cross-language analysis, it introduces the complexity of code transformation between C and Java, requiring comprehensive handling of both languages, significant workload, and potentially leading to soundness and robustness issues.

As a compromise, JN-Sum handles only a subset of C features and performs cursory code transformation, resulting in

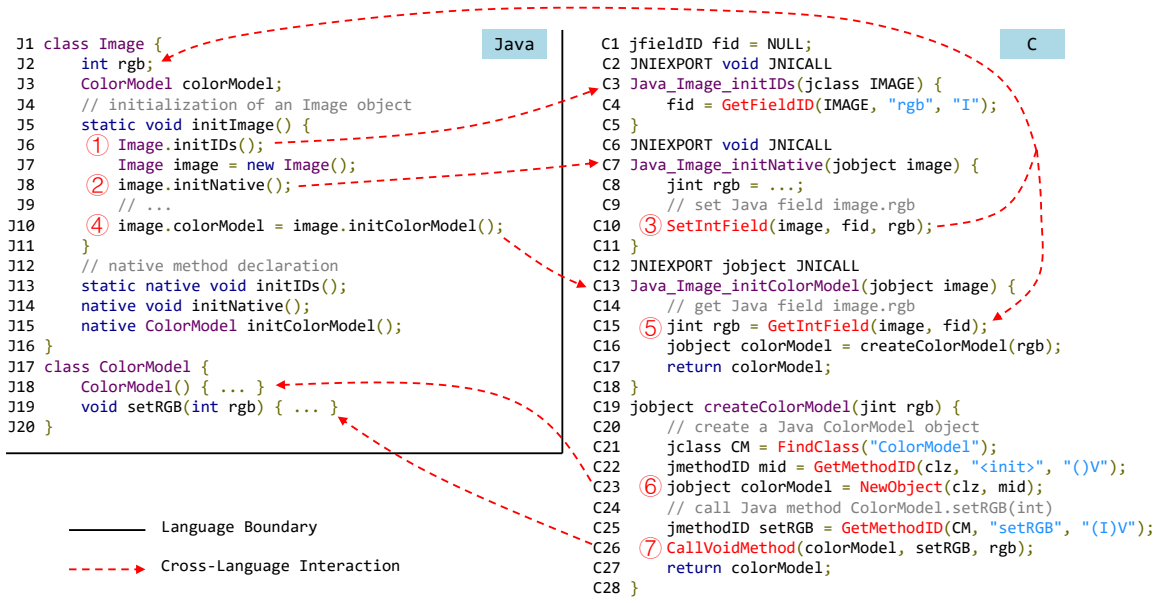

Fig. 1. The motivating example showing interactions between Java and native code. The function names colored in red are JNI functions.

various errors in the transformed Java code. For example, JN-Sum transforms the native function `Java_Image_initIDs` into a static method and the function call at C4 to `GetFieldID(this, "rgb", "I")`. However, the generated Java code is erroneous because there is no `this` variable in a Java static method. This error further prevents JN-Sum from resolving interactions ③ and ⑤, both of which rely on the field ID obtained from C4.

In summary, current state-of-the-art methods have limitations and fail to effectively resolve cross-language interactions, including those in the example shown in Fig. 1.

C. Our Insight

Although performing interactive cross-language pointer analysis is challenging, we believe it is an effective approach for resolving native code and cross-language interactions. This method can adequately collect program information from each cross-language interaction site in both Java and C code, accurately propagate this information to the other language, and achieve effective results.

Therefore, we propose JNIFER, the first interactive cross-language pointer analysis, to resolve native code. JNIFER integrates Java pointer analysis and C pointer analysis, equipped with an innovative cross-language model, enabling the two analyses to interact and collectively resolve cross-language interactions. JNIFER can avoid the imprecision issues of NativeScanner, as well as the complexity and subsequent soundness and robustness issues of JN-Sum, making it more effective at resolving native code compared to existing approaches. For instance, when analyzing ③ in Fig. 1, JNIFER collects information about `image`, `fid`, and `rgb` via C pointer analysis, resolves its target Java field, and propagates it to Java pointer analysis, resulting in an accurate analysis of this interaction site.

III. Methodology of JNIFER

We present an overview of JNIFER and its key components.

A. Overview

Fig. 2 provides an overview of JNIFER, which takes a program containing both Java and native (C) code as input, and outputs the resolved cross-language interactions within the program. The key innovation of JNIFER is its interactive cross-language pointer analysis. JNIFER consists of a Java pointer analysis and a C pointer analysis, which collect program information from both Java and C code. Building on these analyses, we introduce a new JNI analysis, divided into Java and C parts. Both parts run simultaneously with the pointer analyses and interact with each other by exchanging requests and replies to resolve the behaviors at cross-language interaction sites.

As shown in Fig. 2, JNI analysis is the core of JNIFER, consisting of three components. Native call analysis resolves native method calls from Java code (J $\rightarrow$ C interactions), while JNI function analysis resolves JNI function calls from C code (C $\rightarrow$ J interactions). Both analyses are divided into a Java part and a C part, each communicating with their respective pointer analysis. To enable interaction between these two parts, they must be aware of relevant program information (e.g., objects and pointers) from their counterpart. Therefore, we propose a cross-language model to support this communication. Below, we introduce these components in detail.

B. Cross-Language Model

To enable interactive cross-language pointer analysis, we designed a novel model for exchanging program information between the Java and C parts of JNIFER. The core idea of our model is that the Java and C parts convert program elements into numerical IDs or strings and then transmit them

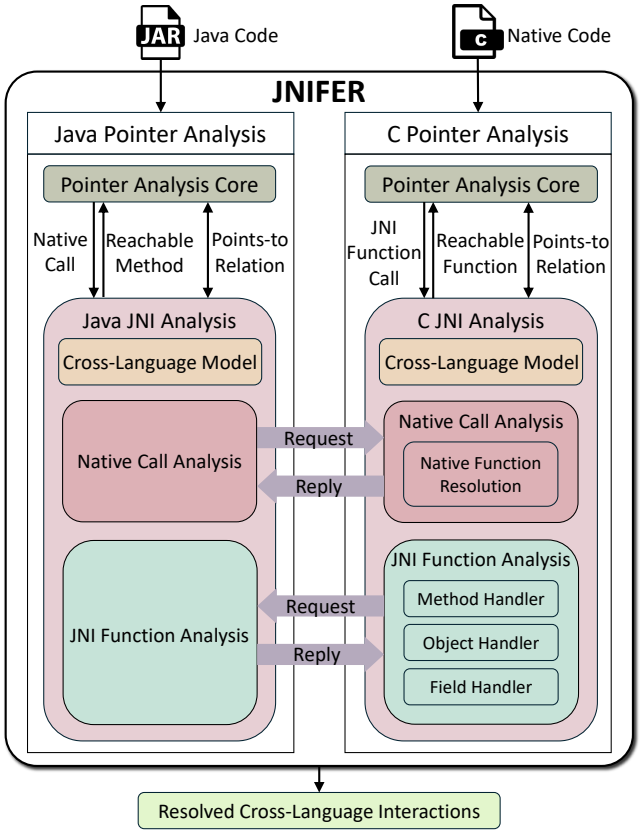

Fig. 2. Overview of JNIFER

to each other via inter-process communication (IPC). Each part maintains mappings between the program elements and the exchanged information. We carefully designed this model to minimize the amount of program information that needs to be exchanged, resulting in a simple yet effective cross-language model. Overall, our model exchanges three types of program elements: Java objects, Java classes/methods/fields, and call sites in both Java and C, as explained below.

a) Java Objects: The Java part and the C part need to exchange Java objects manipulated by native code. Instead of directly exchanging Java objects, JNIFER exchanges the Java pointers that point to the objects. This design reduces the amount of information exchanged, as a single pointer can point to multiple objects. To exchange a Java pointer, the Java part assigns a unique ID to the pointer, and the C part maps this ID to a mock object of type `jobject`, which represents Java references in JNI [2].

The key insight behind this design is as follows: according to the JNI specification, native code can only access Java objects through JNI function calls (e.g., retrieving the type information of a Java object using the `GetObjectClass` function). In JNIFER, the analysis of these JNI function calls is delegated from the C part to the Java part, where the side effects of these calls are analyzed. Consequently, the C part only needs to maintain numerical IDs for the Java pointers referencing these objects. When analyzing JNI function calls, the C part transmits these numerical IDs to the Java part, which then resolves the corresponding Java objects and analyzed the interactions based on JNIFER's JNI function models.

b) Java Classes, Methods, and Fields: We use uniformly formatted string constants to represent the signatures of Java classes, methods, and fields. These signatures are exchanged between the Java and C parts, allowing them to communicate which Java elements they are operating on.

c) Java and C Call Sites: Similar to Java objects, we assign unique IDs to the Java and C call sites for exchange.

C. Native Call Analysis for J → C Interaction

In JNIFER, we designed a comprehensive native call analysis to resolve native method calls in Java, i.e., J → C interaction. This analysis is divided into a Java part and a C part. Briefly, the analysis of a native call (written in Java) aims to accomplish two tasks: first, to identify the corresponding target native function (implemented in C); and second, to analyze the native function (handled by the C part) and feed the side effects back to the Java part. Below, we introduce the working mechanisms of both parts.

a) Java Part: When processing a native method call, since the native method is implemented in C, the Java part cannot analyze it. Instead, the Java part wraps the information at the native call site as a request, including the signature of the native method and the arguments of the call sites, and sends it to the C part, as shown in Fig. 2. For example, when analyzing ④ in Fig. 1, the Java part generates a request containing the method signature `<Image: ColorModel initColorModel()>` and the variable `image`, and sends it to the C part for further processing. Note that such information exchange between the Java part and the C part requires the cross-language model introduced in Section III-B.

The reply from the C part carries information about the return value of the native call, which the Java part uses to complete points-to relations. For example, when analyzing ④, the Java part will use the reply from the C part to fill the points-to set of `image.colorModel`.

b) C Part: The C part of native call analysis is triggered by a request from the Java part. Upon receiving the request, the C part first identifies the native function implementing the native method. This identification is based on the naming convention defined in the JNI specification [2] and the analysis of the JNI function `RegisterNatives`, which developers can use to dynamically register native functions.

After resolving the native function, say f, the C part notifies the C pointer analysis core to mark f as a reachable function and analyze it. If f calls important JNI functions, especially those with side effects on the Java program, then JNI function analysis will come into play as introduced in Section III-D. After analyzing f, the C part processes its return values, wraps them in a reply, and sends them back to the Java part.

For example, for ④ in Fig. 1, the C part resolves `Java_Image_initColorModel` as its target native function based on the naming convention, and analyzes this func-

tion. Finally, it finds that the return value is the object created at ⑥ and wraps it in the reply to the Java part.

D. JNI Function Analysis for C → J Interaction

JNI function analysis is responsible for analyzing JNI function calls, specifically C → J interactions. We focus on the most commonly used JNI functions, which can trigger method calls, object creation, and field accesses in Java programs. These functions are invoked from C code but may have side effects on the Java program. Similar to native call analysis, this analysis is divided into the C part and the Java part. The C part collects method, type, and field information at the JNI function calls, wraps it as a request, and sends it to the Java part, which models the corresponding behaviors in the Java program (method calls, object creation, field accesses). If necessary, the Java part also feeds return values back to the C part. Below, we introduce both parts in more detail.

a) C Part: As shown in Fig. 2, the C part of JNI function analysis comprises three handlers: method, object, and field handlers, each responsible for handling method calls, object creation, and field accesses, respectively. Due to space constraints, we will focus on the object handler here, which is the most complex handler. The details of all three handlers are formalized in Section IV, where you can refer to the specifics of the method and field handlers.

The object handler is invoked when processing a call to the JNI function `NewObject`, which has two key arguments: the type of the Java object to create, and the signature of the constructor for initializing the object. These arguments are typically obtained from the JNI functions `FindClass` and `GetMethodID`, as indicated by their respective arguments. The C part uses C pointer analysis to resolve the arguments of `FindClass` and `GetMethodID`, propagating their results to `NewObject`. In this manner, the object handler resolves the type and constructor signatures of the object to be created, and wraps them as a request to the Java part. For example, when analyzing `NewObject` (⑥) in Fig. 1, the C part resolves its arguments by analyzing `FindClass` and `GetMethodID` at C21 and C22, determining them to be the class `ColorModel` and its no-arg constructor. This information is then wrapped as a request to the Java part.

b) Java Part: Since common JNI functions typically query or manipulate elements of the Java program, which the C part cannot access, JNIFER delegates the detailed analysis of these functions to the Java part. The Java part first retrieves the signature information from the request sent by the C part, and then leverages Java pointer analysis to model the behavior of the JNI functions.

For example, when handling a request for `NewObject`, the Java part uses Java pointer analysis to mock a new object, mark the constructor as a reachable method, and analyze it with the new object. Finally, the Java part wraps the result (including the new object) as a reply to the C part.

IV. Formalism

In this section, we formalize the JNI analysis of JNIFER, introduced in Section III. JNIFER is a sophisticated interactive cross-language pointer analysis, encompassing Java and C pointer analyses along with complicated native call analysis and JNI function analysis. We employ standard Andersen-style analysis [15] for both Java and C pointer analyses. In addition to the method, object, and field handlers described in Section III-D, JNIFER manages various relevant JNI functions, such as `FindClass` and `GetClassObject` for class retrieval, and `GetMethodID` and `GetFieldID` for ID fetching. Due to space constraints, we cannot formalize the entirety of JNIFER in this paper. However, JNIFER will be released as an open-source tool. Here, we first present the domains and notations used (Section IV-A), and then focus on the analysis of native calls (Section IV-B) and common JNI functions (Section IV-C).

Instruction Labels	$i, j \in \mathbb{L} = \mathbb{L}^J \cup \mathbb{L}^C$
Class Types	$t \in \mathbb{T}^J$
Methods	$m \in \mathbb{M}^J$
Method Signatures	$s_m \in \mathbb{S}_m^J$
Variables	$x, y, m_{this}, m_{ret}, arg \in \mathbb{V}^J$
Fields	$f \in \mathbb{F}^J$
Field Signatures	$s_f \in \mathbb{S}_f^J$
Objects	$o_1^t, o_2^t, ... \in \mathbb{O}^J$
Virtual Pointers	$vp_i \in \mathbb{VP}^J$
Pointers	$p_i, p_j \in \mathbb{P}^J = \mathbb{V}^J \cup (\mathbb{O}^J \times \mathbb{F}^J) \cup \mathbb{VP}^J$
Points-to Relation	$pt^J : \mathbb{P}^J \rightarrow \mathcal{P}(\mathbb{O}^J)$

Fig. 3. Domains and Notations (Java Part).

Instruction Labels	$i, j \in \mathbb{L} = \mathbb{L}^J \cup \mathbb{L}^C$
Functions	$fun \in \mathbb{FUN}^C$
Objects	$o_i, o_j \in \mathbb{O}^C$
Variables	$x, fun_{ret}, fid, mid, value \in \mathbb{V}^C$
Pointers	$x \in \mathbb{P}^C = \mathbb{V}^C \cup \mathbb{O}^C$
Points-to Relation	$pt^C : \mathbb{P}^C \rightarrow \mathcal{P}(\mathbb{O}^C \cup \mathbb{P}^J)$

Fig. 4. Domains and Notations (C Part).

A. Domains and Notations

Figs. 3 and 4 illustrate the domains and notations for Java and C analysis, respectively. We use superscripts J and C to distinguish between Java and C domains. Most domains are self-explanatory, but we introduce some special ones below.

In the Java domains, we extend pointers with special virtual pointers, $\mathbb{VP}^J$, which represent mock variables pointing to newly created objects by `NewObject`. We use m_{this} and m_{ret} to denote the *this* and return variables of method m, and arg_k to represent the k-th argument of a method call.

In the C domains, we extend the points-to relation pt^C to allow a C pointer to point to a Java pointer ($\mathbb{P}^J$), supporting the cross-language model described in Section III-B. fun_{ret} refers to the return pointer of a C function, while mid and fid refer to the signatures of a method and field in the Java program.

Table I shows the domains and notations of interaction messages **Request** and **Reply** of the four types of cross-language interactions. We use the tuple $Q_{call}^{J \rightarrow C}[i, s_m, y, arg_1, ..., arg_n]$

TABLE I
DEFINITION OF CROSS-LANGUAGE REQUEST AND REPLY

Cross-Language Interaction			Request	Reply
Native Call Analysis	Native Call		$Q^{J\to C}_{call} = \mathbb{L}^J \times \mathbb{S}^J_m \times (\mathbb{P}^J)^+$	$R^{C\to J}_{call} = \mathbb{L}^J \times \mathcal{P}(\mathbb{P}^J)$
JNI Function Analysis	Method Call		$Q^{C\to J}_{call} = \mathbb{L}^C \times \mathbb{S}^J_m \times (\mathbb{P}^J)^+$	$R^{J\to C}_{call} = \mathbb{L}^C \times \mathbb{P}^J$
	Object Creation		$Q^{C\to J}_{new} = \mathbb{L}^C \times \mathbb{T}^J \times \mathbb{S}^J_m \times (\mathbb{P}^J)^*$	$R^{J\to C}_{new} = \mathbb{L}^C \times \mathbb{VP}^J$
	Field Access	Get	$Q^{C\to J}_{get} = \mathbb{L}^C \times \mathbb{P}^J \times \mathbb{S}^J_f$	$R^{J\to C}_{get} = \mathbb{L}^C \times (\mathbb{O}^J \times \mathbb{F}^J)$
		Set	$Q^{C\to J}_{set} = \mathbb{P}^J \times \mathbb{S}^J_f \times \mathbb{P}^J$	/

$$\frac{i : x = y.m(arg_1, ..., arg_n) \quad m \text{ is native}}{Q^{J\to C}_{call}[i, s_m, y, arg_1, ..., arg_n]}\ [\text{JCALL}] \qquad \frac{Q^{J\to C}_{call}[i, s_m, y, arg_1, ..., arg_n] \quad fun = resolveN(s_m)}{\begin{array}{c} y \in pt^C(fun_{p2}) \\ \forall\ 1 \le k \le n : arg_k \in pt^C(fun_{pk+2}) \\ R^{C\to J}_{call}[i, pt^C(fun_{ret})] \end{array}}\ [\text{CCALLQ}] \qquad \frac{\begin{array}{c} R^{C\to J}_{call}[i, pt^C(fun_{ret})] \\ p_j \in pt^C(fun_{ret}) \\ x = ret(i) \end{array}}{pt^J(p_j) \subseteq pt^J(x)}\ [\text{JCALLR}]$$

Fig. 5. Rules of native call analysis.

$$\frac{i : x = CallMethod(obj, mid, arg_1, ..., arg_n) \quad p_j \in pt^C(obj) \quad s_m \in pt^C(mid) \quad \forall\ 1 \le k \le n : p_k \in pt^C(arg_k)}{Q^{C\to J}_{call}[i, s_m, p_j, p_1, ..., p_n]}\ [\text{CCALL}]$$

$$\frac{\begin{array}{c} Q^{C\to J}_{call}[i, s_m, p_j, p_1, ..., p_n] \\ o_j \in pt^J(p_j) \quad m' = dispatch(o_j, s_m) \end{array}}{\begin{array}{c} o_j \in pt^J(m'_{this}) \quad \forall\ 1 \le k \le n : pt^J(p_k) \subseteq pt^J(m'_{pk}) \\ R^{J\to C}_{call}[i, m'_{ret}] \end{array}}\ [\text{JCALLQ}] \qquad \frac{R^{J\to C}_{call}[i, m'_{ret}] \quad x = ret(i)}{m'_{ret} \in pt^C(x)}\ [\text{CCALLR}]$$

$$\frac{i : x = NewObject(clz, mid, arg_1, ..., arg_n) \quad t \in pt^C(clz) \quad s_m \in pt^C(mid) \quad \forall\ 1 \le k \le n : p_k \in pt^C(arg_k)}{Q^{C\to J}_{new}[i, t, s_m, p_1, ..., p_n]}\ [\text{CNEW}]$$

$$\frac{Q^{C\to J}_{new}[i, t, s_m, p_1, ..., p_n] \quad vp_i = mock(i, t)}{\begin{array}{c} o^t_i \in pt^J(vp_i) \quad R^{J\to C}_{new}[i, vp_i] \quad m = dispatch(o^t_i, s_m) \\ o^t_i \in pt^J(m_{this}) \quad \forall\ 1 \le k \le n : pt^J(p_k) \subseteq pt^J(m_{pk}) \end{array}}\ [\text{JNEWQ}] \qquad \frac{R^{J\to C}_{new}[i, vp_i] \quad x = ret(i)}{vp_i \in pt^C(x)}\ [\text{CNEWR}]$$

$$\frac{i : x = GetField(obj, fid) \quad p_j \in pt^C(obj) \quad s_f \in pt^C(fid)}{Q^{C\to J}_{get}[i, p_j, s_f]}\ [\text{CFIELDGET}] \qquad \frac{Q^{C\to J}_{get}[i, p_j, s_f] \quad o_j \in pt^J(p_j) \quad f = resolveF(s_f)}{R^{J\to C}_{get}[i, o_j.f]}\ [\text{JFIELDGETQ}] \qquad \frac{R^{J\to C}_{get}[i, o_j.f] \quad x = ret(i)}{o_j.f \in pt^C(x)}\ [\text{CFIELDGETR}]$$

$$\frac{SetField(obj, fid, value) \quad p_i \in pt^C(obj) \quad s_f \in pt^C(fid) \quad p_j \in pt^C(value)}{Q^{C\to J}_{set}[p_i, s_f, p_j]}\ [\text{CFIELDSET}] \qquad \frac{Q^{C\to J}_{set}[p_i, s_f, p_j] \quad o_i \in pt^J(p_i) \quad f = resolveF(s_f)}{pt^J(p_j) \subseteq pt^J(o_i.f)}\ [\text{JFIELDSETQ}]$$

Fig. 6. Rules of JNI function analysis.

to refer to a request instance of a native call $i : x = y.m(arg_1, ..., arg_n)$. For example, when analyzing ④ in the Fig. 1, the native call analysis (Java part) generates a request $Q^{J\to C}_{call}$[J10, <Image: ColorModel initColorModel()>, image] and receives a reply $R^{C\to J}_{call}$[J10, vp_{C23}]. A similar representation of a message instance is used for other interactions. Note that the JNI function SetField does not have a return value, and therefore does not generate a reply message.

B. Native Call Analysis

Fig. 5 presents the rules for native call analysis. As described in Section III-C, native call analysis is divided into Java and C parts, with rule names following a specific convention: 1) if a rule belongs to the Java (C) part, its name starts with J (C); 2) if a rule handles a request (reply), its name ends with Q (R).

[JCALL] is applied when processing a native method call, generating a request to the C part. Note that s_m denotes the signature of method m.

[CCALLQ] handles the request in the C part. A helper function $resolveN$, corresponding to "Native Function Resolution" in Fig. 2, is defined to identify the target native function based on the native method signature s_m and the analysis of `RegisterNatives`, as described in Section III-C. The rules for $resolveN$ are omitted here for brevity. After resolving the target native function fun, [CCALLQ] propagates the arguments to fun. According to the JNI specification [2], the first argument of a native function must be the JNI function table `JNIEnv`, thus, the indexing of arguments starts from 2.

Finally, [JCALLR] propagates the return values obtained from the reply, using the helper function $ret(i)$ to retrieve the LHS variable of call site i.

C. JNI Function Analysis

Fig. 6 presents rules of JNI function analysis, in which the rule names follow the convention explained in Section IV-B. Specifically, the rules in the C part correspond to the Method, Object, and Field Handlers in Fig. 2.

In Fig. 6, the top three rules handle the JNI function `CallMethod`. In [JCALLQ], we use the helper function $dispatch(o_j, s_m)$ to identify the actual Java method called from C, based on the type of the receiver object o_j and the method signature s_m, following the dispatch semantics of Java instance methods.

The middle three rules handle `NewObject`. In [JNEWQ], the helper function $mock(i, t)$ creates a virtual variable vp_i of type t for instruction i. This variable is used by Java pointer analysis to point to the object created by the JNI function `NewObject` and is later sent to the C part in a reply message.

The bottom five rules handle the JNI functions `GetField` and `SetField`. We use s_f to denote the signature of field f, which is resolved by the helper function $resolveF(s_f)$ to identify the corresponding Java field. Since `SetField` does not generate a reply message, as shown in Table I, there are no rules required to handle the reply.

V. EVALUATION

In this section, we evaluate the effectiveness of JNIFER by comparing it with state-of-the-art methods in resolving native code behaviors. Specifically, we investigate the following research questions:

- RQ1. How does JNIFER compare to state-of-the-art methods in resolving J $\rightarrow$ C interactions, specifically native method calls?
- RQ2. How does JNIFER compare to state-of-the-art methods in resolving the most commonly used C $\rightarrow$ J interactions, i.e., method calls, object creation, and field accesses?
- RQ3. Is JNIFER efficient?

We first introduce our experimental setup (Section V-A), and then examine the results for RQ1 (Section V-B), RQ2 (Section V-C), and RQ3 (Section V-D).

TABLE II
RECALL AND RESOLUTION FOR J $\rightarrow$ C NATIVE METHOD CALLS.

Bench	#NMethod	JNIFER		JN-Sum		Native-Scanner	
		Recall	#Res.	Recall	#Res.	Recall	#Res.
java.awt	412	**374**	585	342	624	230	609
java.beans	234	**183**	396	164	581	146	581
java.io	159	**134**	150	100	438	120	477
java.lang	196	**149**	170	103	198	134	504
java.math	121	**103**	124	74	433	88	465
java.net	209	**178**	213	131	469	151	508
java.nio	247	**221**	280	190	536	153	512
java.rmi	145	**121**	146	79	150	102	467
java.sec.	136	**117**	150	86	187	99	474
java.sql	105	**82**	105	61	432	73	436
java.text	120	**98**	121	72	432	87	473
java.util	234	**211**	445	174	593	152	611
sun.awt	242	**219**	453	184	596	156	576
sun.java2d	385	**362**	562	314	624	220	594
sun.jvm.	97	**72**	95	55	432	68	436
sun.man.	142	**115**	141	73	437	104	486
sun.misc	193	**170**	194	136	467	146	491
sun.net	213	**187**	227	145	484	150	469
sun.nio	142	**121**	163	96	460	108	462
sun.pisces	195	**178**	405	145	570	126	557
sun.ref.	101	**77**	99	59	432	73	437
sun.rmi	126	**93**	116	76	435	90	469
sun.sec.	160	**133**	179	104	456	119	489
sun.text	109	**86**	108	66	432	78	436
sun.tools	103	**82**	111	62	432	74	437
sun.util	115	**94**	114	70	433	84	463
aspectj	103	**80**	109	CRASH		74	487
lucene	101	**78**	158	63	465	73	513
tomcat-n.	91	**71**	186	54	489	65	509
log4j	91	**74**	150	55	445	66	474
Sum	5027	**4263 (84.8%)**	6455	3333 (66.3%)	13162	3409 (67.8%)	14902

A. Experimental Setup

JNIFER consists of a Java component and a C component for analyzing Java programs and C (native) code, which we implemented on Tai-e [13] and SVF [14], respectively. We run the experiments on a 64-bit machine with an Intel i7-12700KF 3.6GHz CPU and 128 GB of RAM. Below, we describe several key setups in our experiments.

a) The Compared Analyzers: We compared JNIFER with two state-of-the-art native code analyzers, JN-Sum [11] and Native-Scanner [12]. The original JN-Sum has issues that cause it to crash on all our benchmarks. To enable it to analyze more benchmarks, we made two modifications. First, we modified it to quietly ignore unsupported JNI functions to prevent crashes. Second, we removed erroneous statements from Java source code generated by JN-Sum (i.e., the summary), which often contained errors such as assigning a `Throwable` object to an `int` variable. All modifications and fixes will be provided in the artifact. For Native-Scanner, we utilize its artifact version to run our experiments.

b) Benchmarks: For a thorough and practical evaluation, we consider OpenJDK and real-world Java applications as our benchmarks. OpenJDK extensively utilizes native code, presenting numerous cross-language behaviors and various JNI function usages. We choose OpenJDK 8, which is a widely-

TABLE III
RECALL AND RESOLUTION FOR C → J METHOD CALLS AND OBJECT CREATION.

Bench	Method Call							Object Creation						
	#JMethod	JNIFER		JN-Sum		Native-Scanner		#ObjType	JNIFER		JN-Sum		Native-Scanner	
		Recall	#Res.	Recall	#Res.	Recall	#Res.		Recall	#Res.	Recall	#Res.	Recall	#Res.
java.awt	60	**60**	131	15	26	51	7963	30	**30**	58	5	9	28	2884
java.beans	42	39	64	8	23	**40**	7773	22	20	28	1	8	**22**	2757
java.io	10	**10**	18	3	12	**10**	7828	3	**3**	7	1	4	**3**	2798
java.lang	14	**14**	22	4	9	11	8298	5	**5**	9	1	3	2	2910
java.math	7	**7**	17	2	12	**7**	7750	1	**1**	7	**1**	4	**1**	2739
java.net	18	**18**	33	7	17	17	7961	10	**10**	18	5	8	9	2891
java.nio	13	**13**	24	4	13	12	8141	6	**6**	13	2	6	5	2874
java.rmi	13	**13**	26	3	10	12	7931	5	**5**	13	1	4	4	2820
java.sec.	11	**11**	21	3	10	10	7909	4	**4**	10	1	4	3	2851
java.sql	7	**7**	17	2	12	**7**	7675	1	**1**	7	**1**	4	**1**	2679
java.text	8	**8**	17	2	12	**8**	7773	2	**2**	7	1	4	**2**	2752
java.util	38	**38**	61	8	23	36	8413	21	**21**	27	1	8	20	2999
sun.awt	42	**42**	97	8	26	38	7741	22	**22**	41	1	9	21	2741
sun.java2d	47	**47**	124	11	27	42	7899	25	**25**	54	3	9	24	2842
sun.jvm.	7	**7**	17	2	12	**7**	7667	1	**1**	7	**1**	4	**1**	2676
sun.man.	10	**10**	26	3	12	**10**	8291	3	**3**	13	1	4	**3**	2910
sun.misc	15	**15**	26	4	13	14	7809	7	**7**	13	2	5	6	2764
sun.net	19	**19**	29	7	14	**19**	8038	10	**10**	15	4	5	**10**	2924
sun.nio	9	**9**	18	3	12	**9**	8142	2	**2**	7	1	4	**2**	2861
sun.pisces	37	**37**	61	7	23	35	7707	20	**20**	28	1	8	**20**	2713
sun.ref.	7	**7**	17	2	12	**7**	7672	1	**1**	7	**1**	4	**1**	2680
sun.rmi	10	9	19	3	12	**10**	7857	3	2	8	1	4	**3**	2774
sun.sec.	13	**13**	24	3	12	11	8317	5	**5**	11	1	4	3	2961
sun.text	8	**8**	17	2	12	**8**	7688	2	**2**	7	1	4	**2**	2694
sun.tools	7	**7**	17	2	12	**7**	7678	1	**1**	7	**1**	4	**1**	2681
sun.util	7	**7**	17	2	12	**7**	7713	1	**1**	7	**1**	4	**1**	2709
aspectj	7	**7**	24	CRASH		**7**	10797	1	**1**	7	CRASH		**1**	3533
lucene	15	**11**	25	4	14	**11**	12209	5	**3**	12	2	5	**3**	4267
tomcat-n.	17	**13**	25	4	13	**13**	7973	7	**5**	12	2	5	**5**	2745
log4j	10	**10**	24	3	12	**10**	8654	3	**3**	11	1	4	**3**	3084
Sum	528	**516 (97.7%)**	1058	131 (24.8%)	429	486 (92.0%)	245267	229	**222 (96.9%)**	471	46 (20.1%)	152	210 (91.7%)	86513

used JDK version that serves as the foundation for many Java applications. Additionally, our benchmarks include all real-world Java applications previously used in related research [11], [12], namely aspectj-1.6.9, lucene-4.3.0, tomcat-native-1.2.23, and log4j-1.2.16.

c) Ground Truth: To evaluate the effectiveness of JNIFER against other analyzers in terms of soundness and precision, we need to establish a ground truth for cross-language interactions within the benchmarks. This ground truth represents all actual cross-language behaviors at each interaction site. However, due to the complexity of our benchmarks, obtaining their complete cross-language interactions is unrealistic. Therefore, we use runtime instrumentation and sampling manual studies to establish the ground truth for soundness and precision, as explained below.

To measure the soundness of the analyzers, we perform recall experiments on our benchmarks by executing them, recording runtime cross-language behaviors through JVM (Hotspot) instrumentation, and comparing these with the results of static analyzers. For OpenJDK, we run its official test suites, which are organized into 28 directories, each corresponding to a JDK package, such as java.lang and java.io. We randomly selected 100 test cases from each directory (or all available test cases if fewer than 100), except for java.time and sun.invoke, where we encountered compilation issues. For real-world applications, the test suites of aspectj and lucene depend on complex testing frameworks, making it impossible for all three analyzers to reach the application code. Consequently, we manually created test cases for these projects to trigger as many native method calls as possible. For tomcat-native and log4j, we utilized their official test suites.

To establish the ground truth for precision evaluation, we randomly sampled one-third of the benchmarks (10 out of 30). For each selected benchmark, we randomly chose 10 interaction sites (or all available sites if fewer than 10) for each type of cross-language interaction, and then manually determined all possible interactions for each call site by examining the source code.

B. RQ1: Effectiveness of JNIFER for J → C Interactions

Table II presents the results of our recall experiments and the analysis outcomes of the three analyzers for J → C interactions. The column "#NMethod" shows the number of distinct native methods reachable at runtime for each benchmark. For each static analyzer, the column "Recall" shows the number of runtime-recorded reachable native methods resolved by the

analyzer, while the column "#Res" represents the total number of distinct native methods resolved by the analyzer.

JNIFER achieves the highest recall at 84.8%, noticeably outperforming JN-Sum (66.3%) and Native-Scanner (67.8%). Upon investigation, we found that the major reason for JNIFER's superior recall is its better handling of native method dynamic binding. For instance, JNIFER comprehensively analyzes `RegisterNatives`, whereas JN-Sum only supports its limited usage patterns, and Native-Scanner is unable to analyze it, leading to missed native methods called from Java.

To examine precision, we randomly sampled 100 distinct native method call sites and manually inspected their true target native functions (as mentioned in Section V-A). Overall, JNIFER resolved 82 sites, JN-Sum resolved 56 sites, and Native-Scanner resolved 70 sites. We found that all native functions resolved by the three analyzers were true targets, achieving a precision of 100%. However, all call sites successfully resolved by JN-Sum and Native-Scanner could be resolved by simple name matching following the conventions in [2]. In contrast, JNIFER demonstrated a more comprehensive resolution ability by analyzing both name matching and dynamic binding to resolve more call sites.

C. RQ2: Effectiveness of JNIFER for C → J Interactions

In this section, we evaluate how JNIFER performs compared to other analyzers in resolving three common types of C → J interactions: method calls, object creation, and field accesses. As described in Section V-A, we instrumented cross-language interaction sites in native code to obtain true runtime behaviors as ground truth and compared these with the results from the three static analyzers. The results are presented in Tables III and IV. In summary, JNIFER outperforms JN-Sum and Native-Scanner in both soundness and precision for all three types of C → J interactions, with significant soundness advantages over JN-Sum and substantial precision advantages over Native-Scanner. Detailed results are explained below.

a) Method Calls and Object Creation: Table III presents the true cross-language interactions collected via runtime instrumentation and the results of the three static analyzers for method calls (left half) and object creation (right half). The column "#JMethod" indicates the number of distinct Java methods called from native code, and the column "#ObjType" shows the number of distinct types of Java objects created from native code, both collected at runtime in our benchmarks. For each static analyzer, the column "Recall" shows the number of runtime-recorded Java methods and types it resolved, while the column "#Res" indicates the total number of distinct reachable Java methods and created types of Java objects resolved from native code, respectively.

JNIFER achieves the highest recall, with 97.7% for method calls and 96.9% for object creation, by resolving nearly all true Java methods called from native code and all true types of Java objects created from native code in our experiments. Despite we fixed numerous errors in its generated Java summary, JN-Sum still has a low recall of only 24.8% for method calls and 20.1% for object creation. This is because it supports only a subset of the C language and misses some important C features used in native code, such as variadic function calls, leading to an inability to resolve many method IDs in complex real-world native code. Native-Scanner achieved a fairly high recall of 92.0% for method calls and 91.7% for object creation, but at the cost of extremely low precision, as explained below.

TABLE IV
RECALL AND RESOLUTION FOR C → J FIELD ACCESSES.

Bench	#Field	JNIFER		JN-Sum	
		Recall	#Res.	Recall	#Res.
java.awt	114	**101**	156	14	49
java.beans	21	**15**	51	5	39
java.io	12	**12**	17	5	38
java.lang	11	**10**	15	5	17
java.math	11	**10**	15	5	38
java.net	51	**39**	57	17	47
java.nio	41	**41**	52	23	53
java.rmi	25	**18**	29	5	22
java.sec.	17	**16**	39	6	37
java.sql	11	**10**	15	5	38
java.text	11	**10**	15	5	38
java.util	33	**33**	57	19	39
sun.awt	89	**89**	129	7	50
sun.java2d	114	**107**	175	7	50
sun.jvm.	10	**9**	14	5	38
sun.man.	23	**16**	27	5	38
sun.misc	46	**26**	37	19	46
sun.net	47	**40**	63	16	46
sun.nio	12	**12**	18	5	38
sun.pisces	24	**21**	44	5	39
sun.ref.	11	**10**	15	5	38
sun.rmi	15	**10**	16	5	38
sun.sec.	42	**35**	47	23	42
sun.text	11	**10**	15	5	38
sun.tools	11	**10**	15	5	38
sun.util	11	**10**	15	5	38
aspectj	11	**10**	14	CRASH	
lucene	20	**16**	27	5	38
tomcat-n.	25	**21**	47	5	38
log4j	30	**23**	35	5	38
Sum	910	**790 (86.8%)**	1271	246 (27.0%)	1146

In our study, JNIFER achieved a high precision of 99.0% for method calls and 97.3% for object creation, thanks to its interactive pointer analysis approach. While JN-Sum reached a precision of 100.0% for both types of interaction, it was only able to resolve a limited number of interaction sites with simple patterns. Specifically, among the 100 studied C → J method call sites, JN-Sum resolved only 6 sites, whereas JNIFER resolved all 100 sites.

Native-Scanner's precision is very low, at just 0.012% for method calls and 0.14% for object creation, because of its conservative assumption that every Java method or type matching any signature extracted from native code can be invoked or created from native code.

The loss of JNIFER's precision primarily stems from its flow-insensitive pointer analysis, which leads to imprecise points-to sets. A typical example occurs in OpenJDK's native code (`X11Color.c`), where a pointer (`awt_colormodel`) is reused for class lookups with a JNI object creation call in between. Because flow-insensitive analysis does not dis-

tinguish successive assignments to the same pointer, JNIFER imprecisely determines that the pointer could simultaneously reference multiple classes (`DirectColorModel` and `ComponentColorModel`), resulting in a false positive. This limitation could be alleviated in the future by incorporating Static Single Assignment (SSA).

b) Field Accesses: Table IV presents the evaluation results for Java field accesses (both get and set) from native code. The column "#Field" indicates the number of distinct Java fields accessed from native code at runtime. For each static analyzer, the column "Recall" shows the number of runtime-recorded Java fields it resolves, and the column "#Res" shows the total number of fields resolved as being accessed from native code by the analyzer. Native-Scanner does not support analysis of field accesses in native code, hence it is excluded.

JNIFER significantly outperformed JN-Sum in the recall of resolving field accesses, achieving an average recall of 86.8% compared to 27.0% for JN-Sum. A large number of field accesses are missed by JN-Sum due to its incorrect handling of static native method calls. In static native methods, the first argument should be the Java class, which is frequently used to obtain Java field IDs in native code. However, JN-Sum assumes that the first argument of a native method is the *this* object. This incorrect assumption results in significant recall losses in resolving field accesses.

For precision, JNIFER and JN-Sum resolved 73 and 30 sites, respectively, out of the 100 field access sites studied. We manually verified that the field accesses resolved at each site were true positives. Similar to its performance in resolving method calls and object creation, JN-Sum could only resolve straightforward patterns of field accesses, demonstrating significantly less effectiveness compared to JNIFER.

D. RQ3: Is JNIFER Efficient?

We need to examine the efficiency of JNIFER, JN-Sum, and Native-Scanner with caution, as they employ entirely different approaches for resolving native code. Each tool's underlying analysis architecture is distinct and developed based on different frameworks. Specifically, Native-Scanner analyzes C binaries and extracts strings for Java analysis, while JN-Sum first analyzes C source code and then transforms it into Java source code. In contrast, JNIFER directly analyzes the source code of both Java and C simultaneously.

These differences introduce various factors affecting the efficiency of each tool. Despite these variations, JNIFER exhibits the best performance in analysis speed, with an average analysis time of 273 seconds. This is 1.2 times faster than Native-Scanner's 330 seconds and 2.4 times faster than JN-Sum's 663 seconds.

Out of the 30 benchmarks, JNIFER was faster than both Native-Scanner and JN-Sum in 24 cases. However, JNIFER was slower on three specific benchmarks (i.e., java.awt, java.beans, and sun.java2d), with an average of 958 seconds slower than Native-Scanner and 527 seconds slower than JN-Sum. Additionally, in three other benchmarks (i.e., java.util, sun.awt, and sun.pisces), JNIFER was 119 seconds slower than Native-Scanner but 204 seconds faster than JN-Sum on average. The increased analysis time can be attributed to the extensive use of difficult-to-analyze reflection invocations in these benchmarks, which ultimately slowed down the overall analysis process.

E. Threats to Validity

One potential threat to the validity of our evaluation is that we did not cover all of OpenJDK's official test cases, which may affect the reliability of our results. We mitigate this threat by focusing on the test cases in the "java" and "sun" directories, which represent the most widely used and critical functionalities of OpenJDK. While this approach does not cover every test case, we believe it ensures that our evaluation is representative and reliable.

Another concern is that the ground truth used to evaluate the precision of JNIFER and the compared analyzers was constructed manually, which may introduce errors. Tracking the arguments of certain cross-language interaction sites, which determine the targets of these sites, is particularly challenging because these arguments are passed across multiple functions. We mitigate this threat by manually tracing the call stack upwards and inspecting all callers to identify the source of these arguments, thereby determining the targets of the cross-language interaction sites. Furthermore, a second author reviewed and double-checked the identified targets to ensure the correctness of the ground truth.

VI. Related Work

JNIFER resolves native code in Java programs via interactive cross-language pointer analysis. There are two categories of works that are most closely related to ours. The first category is analyzing cross-language interactions between Java and native code. The second category is JNI analysis clients such as JNI specification violation and vulnerability detection.

a) Cross-Language Interaction Analyses: JN-Sum [11], Native-Scanner [12], and MultiQL [25] offer distinct approaches to resolving cross-language interactions but they all do not comprehensively handle the JNI mechanism. JN-Sum provides an abstract interpretation only applicable to a subset of C, suffering from robustness issues that limit its practical applicability. Native-Scanner identifies Java callbacks in native code by extracting string constants but suffers from high false positives due to its assumption. MultiQL is a declarative static analysis approach based on CodeQL for JNI programs with dataflow analysis. It can not support object creation analysis. Though MultiQL has published its source code, we found that it could not be compiled and executed. After contacting the authors, we confirmed that their tool is currently non-functional. Therefore, we did not include a comparison between JNIFER and MultiQL.

ILEA [26] transforms C source code into an extended Java Virtual Machine Language (JVML). However, this extended JVML does not support the full semantics of C, leading to bytecode that may not preserve the original C code's semantics

and may miss cross-language interactions. Additionally, the ILEA tool is currently not available.

Other works focus on native binary analysis of Android apps. Patrik et al. [27] focus on Dalvik bytecode analysis and binary analysis with program slicing to analyze some taint flows in the native binaries. JSA_{DEC-*} [28] decompiles the native binaries of Android apps to C code and uses JN-Sum to analyze the native code. Some other methods [3], [4], [29] use symbolic execution, which results in scalability issues for large codebases. In contrast, our work provides an interactive cross-language pointer analysis that accurately captures the cross-language interactions.

b) JNI Analysis Clients: Many existing approaches focus on monitoring and instrumenting JNI programs. SafeJNI [30] and Jinn [31] insert detection constraints and instrumentations for different JNI specification rules before and after the language boundary, i.e., native method calls and JNI function calls, to monitor the running status of JNI programs. They can accurately report violations when constraints are breached during execution. Other works use static analysis to detect JNI specification violations. Furr et al. [32] build a type system to infer possible Java types that JNI function call arguments may refer to, checking if they conform to the JNI specification. Li et al. [7]–[9] model how JNI functions affect the JVM's exceptional state and detect missing error checks in JNI programs. Kondoh et al. [6] detect missing error checks, JVM local resource leaks, and improper access to JVM critical regions through typestate analysis.

CSS [5] provides a caller-sensitive cross-language taint analysis for native code, analyzing pre-tainted sources in a demand-driven way. CHERI [33] extends Java's security model to native code using hardware-assisted implementation, detecting vulnerabilities that bypass memory protection and high-level security policies, enforcing memory safety and compartmentalization. NCScope [34] uses hardware-assisted methods to collect execution traces and memory data of Android apps, detecting self-protection and anti-analysis mechanisms and diagnosing memory corruption bugs. NATIDROID [35] maps permissions for Android APIs, focusing on native library permissions, and detects vulnerabilities like permission over-privilege and component hijacking. μDep [36] provides a hybrid analysis framework using lightweight static binary analysis with differential fuzzing to identify dependencies between arguments and return values of native methods, detecting sensitive information flows in Android native code. LibDroid [37] offers a static analysis framework summarizing information flows within Android native binaries to identify potential security vulnerabilities, such as misuse of sensitive information. Gu et al. [38], [39] detect potential JNI global reference exhaustion with a hybrid analysis for Android native code. They analyze call graphs of Android system service code to identify potentially vulnerable APIs and automatically tests the findings to reduce false positives.

VII. Conclusion

We introduced JNIFER, the first interactive cross-language pointer analysis for resolving Java native code. JNIFER effectively integrates Java and C pointer analyses, enabling on-the-fly resolution of native calls (Java→C) and JNI function calls (C→Java). Extensive experimental results demonstrate that JNIFER significantly outperforms state-of-the-art native code analysis tools in terms of soundness, while maintaining high precision and efficiency. We expect that the design methodology of interactive cross-language pointer analysis underlying JNIFER can facilitate the development of other client analyses requiring more sound control and data flow handling between Java and C. Furthermore, the methodology may also inspire interactive pointer analysis between other programming languages, like Python and C.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by National Key R&D Program of China under Grant No. 2023YFB4503804, National Natural Science Foundation of China under Grant No. 62402210, and the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[2] Oracle, "Java native interface specification contents," https://docs.oracle.com/en/java/javase/17/docs/specs/jni/index.html, 2024.

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

[4] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1232–1244.

[5] S. Kan, Y. Gao, Z. Zhong, and Y. Sui, "Cross-language taint analysis: Generating caller-sensitive native code specification for java," *IEEE Transactions on Software Engineering*, 2024.

[6] G. Kondoh and T. Onodera, "Finding bugs in java native interface programs," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 109–118.

[7] S. Li and G. Tan, "Finding bugs in exceptional situations of jni programs," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 442–452.

[8] ——, "Jet: exception checking in the java native interface," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 345–358. [Online]. Available: https://doi.org/10.1145/2048066.2048095

[9] ——, "Exception analysis in the java native interface," *Science of Computer Programming*, vol. 89, pp. 273–297, 2014.

[10] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[11] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 127–137.

[12] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, "Identifying java calls in native code via binary scanning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 388–400.

[13] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1093–1105. [Online]. Available: https://doi.org/10.1145/3597926.3598120

[14] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[15] L. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, 1994.

[16] W. Ma, S. Yang, T. Tan, X. Ma, C. Xu, and Y. Li, "Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 539–564, 2023.

[17] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis, "Making pointer analysis more precise by unleashing the power of selective context sensitivity," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[18] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 42, no. 2, pp. 1–40, 2020.

[19] ——, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[20] ——, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 129–140.

[21] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 2, pp. 1–50, 2019.

[22] Y. Lei and Y. Sui, "Fast and precise handling of positive weight cycles for field-sensitive pointer analysis," in *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings 26*. Springer, 2019, pp. 27–47.

[23] M. Barbar and Y. Sui, "Hash consed points-to sets," in *International Static Analysis Symposium*. Springer, 2021, pp. 25–48.

[24] ——, "Compacting points-to sets through object clustering," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[25] D. Youn, S. Lee, and S. Ryu, "Declarative static analysis for multilingual programs using codeql," *Software: Practice and Experience*, vol. 53, no. 7, pp. 1472–1495, 2023.

[26] G. Tan and G. Morrisett, "Ilea: Inter-language analysis across java and c," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, 2007, pp. 39–56.

[27] P. Lantz and B. Johansson, "Towards bridging the gap between dalvik bytecode and native code during static analysis of android applications," in *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2015, pp. 587–593.

[28] J. Park, S. Lee, J. Hong, and S. Ryu, "Static analysis of jni programs via binary decompilation," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3089–3105, 2023.

[29] L. Borzacchiello, E. Coppa, D. Maiorca, A. Columbu, C. Demetrescu, and G. Giacinto, "Reach me if you can: On native vulnerability reachability in android apps," in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 701–722.

[30] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. C. Wang, "Safe java native interface." in *ISSSE*. Citeseer, 2006.

[31] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley, "Jinn: synthesizing dynamic bug detectors for foreign language interfaces," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 36–49.

[32] M. Furr and J. S. Foster, "Polymorphic type inference for the jni," in *European Symposium on Programming*. Springer, 2006, pp. 309–324.

[33] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son *et al.*, "Cheri jni: Sinking the java security model into the c," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 569–583, 2017.

[34] H. Zhou, S. Wu, X. Luo, T. Wang, Y. Zhou, C. Zhang, and H. Cai, "Ncscope: hardware-assisted analyzer for native code in android apps," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 629–641.

[35] C. Li, X. Chen, R. Sun, J. Xue, S. Wen, M. E. Ahmed, S. Camtepe, and Y. Xiang, "Natidroid: Cross-language android permission specification," *arXiv preprint arXiv:2111.08217*, 2021.

[36] C. Sun, Y. Ma, D. Zeng, G. Tan, S. Ma, and Y. Wu, "μdep: Mutation-based dependency generation for precise taint analysis on android native code," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1461–1475, 2022.

[37] C. Shi, C. C.-C. Cheng, and Y. Guan, "Libdroid: Summarizing information flow of android native libraries via static analysis," *Forensic Science International: Digital Investigation*, vol. 42, p. 301405, 2022.

[38] Y. Gu, K. Sun, P. Su, Q. Li, Y. Lu, L. Ying, and D. Feng, "Jgre: An analysis of jni global reference exhaustion vulnerabilities in android," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 427–438.

[39] Y. He, Y. Zhou, Y. Zhou, Q. Li, K. Sun, Y. Gu, and Y. Jiang, "Jni global references are still vulnerable: Attacks and defenses," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 1, pp. 607–619, 2020.