Challenges of Multilingual Program Specification and Analysis[⋆]

Carlo A. Furia and Abhishek Tiwari

Software Institute, USI Università della Svizzera italiana, Switzerland
bugcounting.net · abhishek.tiwari@usi.ch

Abstract. Multilingual programs, whose implementations are made of different languages, are gaining traction especially in domains, such as web programming, that particularly benefit from the additional flexibility brought by using multiple languages. In this paper, we discuss the impact that the features commonly used in multilingual programming have on our capability of specifying and analyzing them. To this end, we first outline a few broad categories of multilingual programming, according to the mechanisms that are used for inter-language communication. Based on these categories, we describe several instances of multilingual programs, as well as the intricacies that formally reasoning about their behavior would entail. We also summarize the state of the art in multilingual program analysis, including the challenges that remain open. These contributions can help understand the lay of the land in multilingual program specification and analysis, and motivate further work in this area.

1 Introduction

It is increasingly common that modern programming frameworks offer *multilingual* features,[1] where different modules of an application are written using different programming languages. The rise of web development, where an application's front-end and back-end are often developed using different languages, has certainly made multilingual programming commonplace; however, various forms of multilingual programs have been around for much longer (for example, most high-level programming languages offer some form of foreign-function interface).

Regardless of the details of how it is implemented, multilingual programming is likely to add to the challenges of rigorously specifying and analyzing a program's behavior. The programming languages that cooperate in a multilingual program often have (markedly or subtly) different features such as types, memory models, paradigms, and idioms. Even in the optimistic scenario where detailed formal specifications of each component are available, combining them at the language boundary to analyze a multilingual program's overall behavior requires additional technical insights.

This paper aims at drawing a high-level picture of how program specification and analysis are complicated by multilingual features, in order to both better understand the main open challenges, and to motivate further research in this area. To this end, it

⋆ Work partially supported by SNF grant 200021-207919 (LastMile).

[1] Multilingual programs are also sometimes called *polyglots*.

first discusses, in Section 2, various mechanisms for inter-language communication that are commonly used in different kinds of multilingual programs. For each of them, Section 3 outlines the main challenges that they pose to (formally) specifying and analyzing programs using them, and demonstrates them, in a nutshell, on simple examples. The examples are stripped down to be as clear and as succinct as possible; however, they recapitulate challenges and issues that are present, on a much larger scale, in real-world programs. Section 4 and Section 5 bear out this intuition by discussing the essential state of the art of multilingual program analysis, and outlining its open challenges.

2 Dimensions of Multilingual Programming

We organize our discussion of multilingual program analysis along two dimensions. First, Section 2.1 presents a broad categorization of multilingual *programs* according to the *level* of abstraction that their inter-language communication targets. Then, Section 2.2 introduces *layers* of program *specification* (and the corresponding *analyses* that they enable) that are relevant for multilingual programs.

As for the rest of this paper, the dimensions are not meant to be exhaustive or strictly mutually exclusive. Pragmatically, they are useful to illustrate the most common challenges of multilingual program specification on concrete, relevant examples.

2.1 Multilingual Program Levels

One natural way to classify multilingual programs is according to the features multilingual communication relies on. We identify four groups of features, going roughly from higher-level to lower-level. This classification takes the perspective of a programmer developing an application in a host language that includes parts written in a guest language; thus, the "higher-level" the multilingual features are the more transparent and "natural" the combination of functionalities written in the two languages is.

API level. When the host language offers an *API* with features to directly access the guest language's runtime, writing host code that interacts with the guest is straightforward, as it is very similar to using a regular library in the host language. For example, as we show in Section 3.1, Android apps (implemented in Java or Kotlin) may define JavaScript interfaces that give direct access to the app's functionality.

IR level. If the host and the guest languages are deployed on the same runtime, it means their compilers can translate language features into a common intermediate representation (*IR*). This provides an indirect way of communicating between the two source languages, so that a language's code can usually call into code written in the other language. For example, JVM languages such as Java, Kotlin, and Scala all translate into the same bytecode instruction set; hence, a program written in, say, Scala can use compiled Java libraries with hardly any restrictions—as we discuss in Section 3.2's example.

Native level. A *native* interface gives the host language access to native applications—that is, applications that run natively on the host's hardware and operating system. For example, as we show in Section 3.3, JNI (Java Native Interface) offers direct access to C or even assembly code from Java by means of suitable conversion functions and wrappers. While native multilingual programs can be considered a special case of those relying on an API, the expressiveness gap between a high-level language (such as Java) and a system-specific binary is likely to be wider than the gap between high-level languages plugged into a shared API.

System level. The last group of features for multilingual programming is very broad, as it includes all cases of communication using lower-level communication primitives offered by the *system* that is running the program. For example, two programs running on the same operating system can communicate through inter-process communication; even if they run on two different machines, they can still communicate by means of networking primitives.

2.2 Analysis and Specification Layers

Consider the analysis of a multilingual program written in a combination of a host language and a guest language. Conceptually, the interface between host and guest is summarized by a (implicit or explicit, inferred or expressed) *specification*. A certain kind of program *analysis* can cross the inter-language interface only if the interface specification captures the right kind of information. Accordingly, we consider three levels of specification layers—roughly going from less to more detail.

Types. The *types* of a procedure's input and output are a basic constrain on its valid behavior. Thus, they are a natural way of exchanging information across language boundaries—as well as a possible source of subtle misbehavior when converting between similar types of two languages may lose information.

Dataflow. If a dataflow analysis can cross the interface between two languages, it means that it can follow program paths that go from one language and the other. It is well known that dataflow analysis is especially convenient to verify a range of *security* properties such as noninterference, as well as to detect common programming errors such as resource *leaks*.

Effects. Fully specifying the *side effects* of a call (in other words, its *frame*) is a challenge even for individual programming languages [14, 30, 42, 53]. The "mismatch" that often occurs between the semantics of different languages renders fully expressing, and reasoning about, the effects of an inter-language call especially daunting.

3 Examples of Multilingual Analysis and Specification

This section presents several examples of snippets of code from multilingual programs, and discusses the ensuing challenges of analysis and specification on them.

3.1 API Level

As a significant example of multilingual programming through an API, we consider *hybrid* mobile applications ("apps") that use Android's WebView component.[2]{b} This

[2] In the paper, we refer to several online documents by means of URL references: these are marked by superscript letters in blue between curly braces,{a} so that they can be easily distin-

```
1  class Messenger {
2    private String message;
3
4    @JavascriptInterface
5    public String getMessage();
6
7    @JavascriptInterface
8    public String getInfo();
9    @JavascriptInterface
10   public void setInfo(String info);
11
12   @JavascriptInterface
13   public ArrayList<String> asList();
14 }
```

(a) An example of a Java class whose interface methods are made available to JavaScript code through Android's WebView component.

```
15 class MainActivity {
16   protected void onCreate(Bundle state)
17   {
18     WebView wv = new WebView(this);
19     WebSettings wvset = wv.getSettings();
20     // enable JavaScript bridge
21     wvset.setJavaScriptEnabled(true);
22     // add interface object
23     Messenger msg = new Messenger(this);
24     wv.addJavasriptInterface(msg, "msg");
25     // run JavaScript from URL or string
26     wv.loadUrl(/* URL or JS */);
27   }
28 }
```

(b) An example of Android activity that instantiates a WebView component, enables JavaScript communication, and makes an instance of `Messenger` available to the JavaScript runtime.

```
29 // Call function after DOM is loaded
30 document.addEventListener(
31   "DOMContentLoaded", function() {
32     var message = msg.getMessage();
33     // Add message obtained from Java
34     // as a new paragraph in webpage
35     var p = document.createElement("p");
36     p.textContent = message;
37     document.body.appendChild(p);
38 });
```

(c) An example of JavaScript code that reads a message from Android/Java and displays it into the current web page.

```
39 // Information security leak
40 var device = msg.getInfo();
41
42 // Dataflow from JS to Android
43 msg.setInfo(
44   document.getElementById("Search")
45 );
46
47 var lst = msg.asList();
48 // lst is a generic object
```

(d) Snippets of JavaScript code that misuse the Java-JavaScript bridge by modifying an Android app's behavior in unintended ways.

Fig. 1: Code snippets that demonstrate Java-JavaScript hybrid programs written using Android's WebView framework.

component embeds a headless browser within a host Android app; the browser can, among other things, run JavaScript code that accesses the Android runtime through a programmer-defined API.

Figure 1a shows a Java class `Messenger` that marks four methods with the annotation `@JavascriptInterface`.{c} In Figure 1b, the app's main activity initializes the WebView component (line 18), allows it to run JavaScript code (line 21), creates an instance `msg` of the `Messenger` class (line 23), and registers it with the WebView object: after line 24, any JavaScript code that runs in the WebView will have access to a persistent property

guished from regular footnotes, and listed at the end of the paper after the usual bibliographic references.

`msg` of the browser's DOM instance with an interface corresponding to class `Messenger`'s `@JavascriptInterface` methods. Finally, in line 26, the Android activity loads content in the WebView by passing either the URL of a webpage (which may embed JavaScript code), or directly JavaScript code as a string to `loadUrl`. Figure 1c is an example of JavaScript code that could be passed, as a string, to `loadUrl`: a callback that, as soon as the DOM is fully loaded, calls `getMessage()` on the bridged object `msg` and inserts the received string as a new paragraph of the current web page.

The Java-JavaScript bridge offered by Android's WebView component is widely used in mobile apps [36, 61][d], as it provides a convenient way of programming platform-independent web components in JavaScript, and then combining them with Java code to create a full-fledged app. Using the WebView's API-level inter-language communication framework, developers can flexibly define the Java interface that will be visible and accessible to the JavaScript code (Figure 1a), co-designing the components written in the two languages so that they seamlessly communicate. On the other hand, this flexibility may also complicate the analysis and specification of the inter-language program behavior, as we'll discuss henceforth.

Types. Any Java method can be annotated with `@JavascriptInterface`. However, Java's and JavaScript's type systems are quite different: only Java's primitive and string types are reliably converted to their JavaScript counterparts (e.g., numeric, Boolean, and string types). Consequently, multiple studies [9, 28] have shown several instances of type bugs in Java-Javascript communication in hybrid apps. Besides, instances of complex (possibly user-defined) Java reference types appear in JavaScript as generic `object` instances, as outlined in Figure 1d's lines 47–48. Thus, even if the Java-JavaScript communication goes through a user-defined interface with typed signatures, fully analyzing the overall behavior of a hybrid app requires a precise formalization of type conversions between two fundamentally different type systems—in particular, static vs. dynamic. In fact, some JavaScript operations behave slightly differently depending on whether they run on "native" JavaScript objects or on Java-bridged objects; for example, Tiwari et al. [62] noticed that deleting an instance method of a Java interface object has no effect—whereas one can delete individual methods of a regular JavaScript object.

Dataflow. The information flow in Android hybrid apps is *bidirectional*: the JavaScript runtime can not only read information from the Java runtime (line 40 in Figure 1d) but also write it back (line 43 in Figure 1d). This compounds the difficulty of rigorously analyzing the dataflow that cross language boundaries (which is already challenged [43, 44, 54] by JavaScript's highly dynamic nature[e]). For example, many Android techniques to detect information-flow security leaks [28, 62] are based on the idea of detecting flows of sensitive information into public channels. There is evidence [61, 63] that several Android hybrid apps suffer from such information-flow security issues[f] that state-of-the-art analysis techniques would not be capable of detecting automatically.

Effects. Line 43 in Figure 1d also captures a scenario where detecting all the effects of executing some JavaScript code within a WebView component would be challenging. The JavaScript code sets some "information" state component in the Android runtime to the search results in the current web page; since the search results are loaded dynamically, generally specifying how the state will change requires some kind of environ-

```
class JListWrapper[A](val underlying: java.util.List[A])
  extends mutable.AbstractBuffer[A]
  with ...
{
  def length = underlying.size
  def insert(idx: Int, elem: A): Unit = underlying.subList(0, idx).add(elem)
  // ...
```

(a) An excerpt of Scala's `scala.collection.convert.JListWrapper`.

```
// Java list, unmodifiable
val jl: java.util.List[String] = java.util.List.of("a", "b", "c")
// Convert Java List to Scala mutable Buffer using JListWrapper
val sl: mutable.Buffer[String] = jl.asScala
sl.insert(1, "X")
```

(b) A mismatch between Scala and Java types causes an `UnsupportedOperationException`.

Fig. 2: Code snippets that demonstrate Scala data structures wrapping Java ones.

ment specification. Studies [61, 63] have found several examples of apps{g}{h}{i} that abuse this mechanism to inject malware into an Android app, or to simply obfuscate its behavior. Android WebView's threaded model further complicates the synchronization between the Java and JavaScript runtimes: JavaScript code may execute asynchronously with respect to when it is started by an Android call to `loadURL`, and hence it may access a bridged object in different states in different executions [62]. By itself, JavaScript's highly dynamic nature also complicates reasoning about its semantics statically; for example, the JavaScript code running in a WebView component can redefine a method of the JavaScript interface [62].

3.2 IR Level

Scala is a modern, statically typed programming language [39] that runs of the Java Virtual Machine (JVM) and compiles to bytecode (JVM's intermediate representation). Scala has prominent *functional* features, but is actually a multi-paradigm language—also offering a sophisticated support for object orientation. Interoperability with Java is another key feature of Scala, whose programs can seamlessly use libraries written in other JVM languages.

As a significant example of multilingual Java-Scala programs, we consider how Scala collections support Java's widely used `java.util.List` type.[3] To optimize performance, when converting a Java collection to a suitable Scala type, Scala *wraps* the Java objects with a suitable Scala interface—avoiding, as much as possible, copying the collection's content.{j} Figure 2 shows a small excerpt of Scala's wrapper for Java's `List`,{k} as well as an example of Scala code seamlessly using Java lists. We will now

[3] Scala has substantially revised its collections framework with version 2.13 of the language; in this paper, we consider Scala 3, which also uses v. $\geq$2.13 of the collections.

go into the details of this code, and its implications for the correctness of multilingual programs.

Types. Scala collections have been carefully designed to ease interoperability with Java while offering more expressive, functional features on top. However, this also entails that some limitations of Java collections' design leak into Scala—rendering a precise conversion between the two type systems not always possible. As shown in Figure 2a, Java `List` type's wrapper `JListWrapper` extends Scala's `mutable.AbstractBuffer`.{l} This makes perfect sense, since a Java's list interface provides methods "to efficiently insert and remove [...] elements at an arbitrary point in the list".{m} The catch is that Java uses the same `List` type also to denote *unmodifiable* lists; calling modification operations on an unmodifiable list is allowed by the type system but results in an exception at runtime. This shortcoming of the Java collection API can also affect, through the wrappers, Scala collections.

The Scala code in Figure 2b's line 2 creates an instance `jl` of `java.util.List` by calling method `List.of`,{n} which returns an unmodifiable list. Since `jl`'s immutability is unknown statically, line 4 creates a *mutable* wrapper `sl` around `jl`. Henceforth, calling any modification operation—such as `insert` at line 5, which calls `List.add` on the underlying `jl`—results in an `UnsupportedOperationException` at runtime. This example shows that, even for closely interoperable languages such as Scala and Java, precisely reasoning about type conversions at the language interface may be arduous.

Dataflow. Dataflow analyses are naturally defined at the level of an intermediate representation [5, 8]. Thus, combining Java and Scala dataflow analyses at the level of bytecode is, in principle, an effective way of applying such kind of analysis to a multilingual program. Still, designing a static analysis that is effective and sufficiently precise for programs spanning two different programming paradigms remains challenging. For example, the popular IntelliJ IDEA includes a dataflow analysis{o}{p} that can uncover common programming mistakes such as dead code and null dereferencing. This functionality was originally developed for Java as a purely intraprocedural analysis; more recently, it has been extended to Scala code.{q} The Scala dataflow analysis is, however, *inter*procedural, because the IntelliJ developers decided that being able to follow a control flow into function calls is paramount in a functional language like Scala;{r} it is also, arguably, more feasible, by taking advantage of the prevalent usage of immutable data in Scala—as opposed to old-fashioned imperative programming in Java.

Effects. The imperative/functional abstraction gap also affects how we can reason about the precise behavior of a call where callee and caller are programmed in two different languages. A strict functional Scala program consists of a collection of *pure* (side-effect free) functions, which simplifies reasoning about their behavior. Scala's expressive type system can still accommodate meaningful side effects using, for example, an effect system [48, 49]. However, Scala programs often take full advantage of the language's versatility by combining purely functional and imperative features—for instance, to implement parallel and distributed programming features [20, 40]. As a result, fully analyzing the behavior of a combination of imperative and functional features remains a formidable challenge [38, 67].

```java
class Messenger {
 private String message;

 public static native String fromC();
 public static native void toC(String msg);

 // load shared library within static block
 static { System.loadLibrary("Messenger"); }

 public String getMessage()
 { message = fromC(); return message; }

 public void setMessage(String msg)
 { message = msg; toC(message); }
}
```

(a) An example of a Java class some of whose interface methods are implemented as native code.

```c
JNIEXPORT void JNICALL toC (JNIEnv * env, jclass obj, jstring jstr)
{ const char *cstr;
  cstr = (*env)->GetStringUTFChars(env, jstr, NULL);
  if (cstr == NULL) return;

  // Use cstr as a null-terminated C string ...
  // ...

  // Release memory after usage
  (*env)->ReleaseStringUTFChars(env, str, cstr); }

JNIEXPORT jstring JNICALL fromC (JNIEnv * env, jclass obj)
{ char cstr[] = /* Get string from C program */;
  return (*env)->NewStringUTF(env, cstr); }
```

(b) Outline of the C implementations JNI native methods `toC` and `fromC`.

Fig. 3: Code snippets that demonstrate Java Native Interface's (JNI) capabilities.

3.3 Native Level

Java Native Interface (JNI) is a framework that gives Java applications access to native libraries. JNI has been extensively used to implement core JDK functionality [23, 57] such as input/output and device access. In this section, we go over a simple example of Java-C hybrid program using JNI, highlighting the features that affect reasoning about and specifying the multilingual program's behavior.

As mentioned in Section 2.1, a multilingual native interface such as JNI essentially implements a form of API-level communication—but one with a wide inter-language abstraction gap. Figure 3a shows a Java class `Messenger` that includes two method signatures `fromC` and `toC` with modifier **`native`**. This denotes that their implementation is not given in Java, but is to be found in some native library that is available to the JVM where this code will run. Apart from this detail, methods `fromC` and `toC` can be used by the Java program just like any other method—as demonstrated by the other two methods `getMessage` and `setMessage` of the same class `Messenger`.

A fundamental difference between Figure 3's example of Java-C communication, and Figure 1's example of Java-JavaScript communication lies in how the two languages communicate at the interface. In the Java-JavaScript bridge, methods marked with `@JavascriptInterface` are implemented in Java and *callable* from JavaScript; in contrast, in the Java-native JNI framework, methods marked with **`native`** in Java are *implemented* in C (and compiled to native) and then used regularly in Java.

Types. Figure 3b outlines C implementations of Figure 3a's native methods `fromC` and `toC`. Even in such a simple example, the wide gap between Java and C types is apparent. In particular, the C code needs boilerplate code to explicitly:{s} *i*) access the Java runtime through reference `env`, *ii*) retrieve the string object pointed to by `jstr`, *iii*) convert it to a null-terminated C string (using JNI utility function `GetStringUTFChars`). Conversely, JNI function `NewStringUTF` handles the opposite conversion—from a C character array to a Java string. On the one hand, these JNI library functions handle the conversion between corresponding Java and C types, ensuring that no information is lost.{t} On the other hand, the conversion is far from straightforward: even if we could precisely check extended type correctness properties of the C implementations, lifting the results of the analysis to the Java-C interface would require more legwork, and being able to reconcile idiomatic usage in the two languages.

Dataflow. As clear from Figure 3's example, the control flow between Java and C (or any other language compiled to native) is bidirectional, in that the C code can both read (e.g., `toC`) and modify (e.g., `fromC`) data in the Java runtime. Similarly to Figure 1's example of Java-JavaScript API communication, analyzing a bidirectional inter-language dataflow is complex; the lower-level nature of C aggravates the complexity. In particular, there is an array of pitfalls that may affect C programs (such as undefined behavior, memory leaks, and so on) and thus, indirectly, also the Java code that would otherwise not have to worry about these issues (thanks to Java's stricter type system, memory model, and automatic memory management). There is evidence that hybrid Java-native apps do suffer from vulnerabilities that originate in such issues of C [29, 51, 69].

Effects. Java is an object-oriented language, and hence it uses classes as the fundamental unit of *encapsulation*. In contrast, C does not offer comparable modular features beyond functions (and files). In practice, though, C programmers often still find ways of combining data and functions that operate on it in a coherent way—for example, by defining a series of functions that all take a reference to the same **`struct`** as input, as if it were the target object of methods of the same class [64]. It remains that, in order to specify and reason about programs that combine the two languages, one needs a *methodology* that properly captures each language's notion of encapsulation—and then a way of reconciling the two formal models. C's complex memory model, as well as its looser notion of encapsulation, entail that applying specification methodologies designed for object-oriented languages like Java to C may involve more work in terms of annotations and adaptability to analyzing idiomatic C code. For example, the VCC verifier [13] features a sophisticated ownership model to verify low-level, idiomatic C code; the VeriFast verifier [24] uses separation logic and abstract predicates [41] to specify

```
35 output = subprocess.run(
36     "sort -t, -k5 -rg cities.csv | awk -F, '{print $2}'",
37     capture_output=True,
38     shell=True
39 )
40 # retrieve shell output and convert it to UTF string
41 data = output.stdout.decode('utf-8')
42 # print first row of 'data'
43 print(data.splitlines()[0])
```

Fig. 4: A Python script that communicates with shell command line utilities.

both C and Java programs in a similar style—which could be a basis to construct a multilingual methodology.

3.4 System Level

The lowest level of multilingual communication encompasses very many concrete combinations of languages and systems. As a simple example of programs that rely on system-level communication between components implemented in different language, we consider scripts written in Python that interact with Unix command-line utilities through the operating-system shell.

Figure 4 shows a simple Python program that uses standard library `subprocess` to call the shell (line 38) in order to run command-line utilities `sort` and `awk` in combination. Line 36 displays the executed commands: first, `sort -t, -k5 -rg cities.csv` outputs the rows of comma-separated file `cities.csv` (`-t,`) sorted in decreasing numerical order (`-rg`) of their fifth field (`-k5`); this command's output is piped into the command `awk -F, '{print $2}'`, which prints out the second field (`$2`) of every row. File `cities.csv` lists information about various cities in the world; its second column is the city name, and its fifth column is the city's population. Thus, the command returns a sequence of rows with all city names sorted from most to least populated. Therefore, `subprocess.run`'s call on line 35 returns a Python object with information about the outcome of executing the command, such as its exit status and any error that may have occurred. Then, line 41 extracts the command's raw output (attribute `stdout`) and converts it from a byte string to a character string; and line 43 splits the string at line-breaks and prints the first one (i.e., the name of the city with the largest population count).

Even from this simple example, it is clear how the interface between, in this case, Python and the system primitives is low-level, and relies on several conventions to share information implicitly.

Types. The range of types that are available between languages at the system level is usually quite limited. In our example, all the information exchanged between shell and Python program is represented as a string (plus possibly the numeric exit code of the shell process). This means that there are very few checkable guarantees that the data received by the Python program is in a consistent format: for example, line 43 relies on the assumption that `data` was properly encoded as a linebreak-separated lines, without spurious information before the first line of data. Conversely, the Python program risks passing data to the shell that is in an incorrect or inconsistent format; for example, it has to carefully quote or escape characters that have a special meaning in the shell (e.g., `$`).

Dataflow. Precisely analyzing the dataflow of a program that calls out to the operating system is a daunting task. Dynamic analysis, such as a taint analysis [25,37], is the most viable approach, since it relies on concrete executions in a real-world environment—rather than on abstract models of the system, which would be difficult to define accurately. Also techniques that combine static and dynamic analysis, such as dynamic-symbolic execution [11, 19], could be effective to detect information-flow vulnerabilities [27,59] in such hybrid programs—provided they can rely on constraint solvers that support the ubiquitous string type [7, 18].

Effects. Reasoning about the behavior of a high-level program that interacts directly with other system components would require a large amount of manual, expert work, pushing the envelope of formal methods. One approach could be to carefully define stubs of the specific system functionalities that interact with the high-level program (in Figure 4's example, the command-line utilities `sort` and `awk`), and specify them formally; this approach would essentially reduce the analysis problem to a scenario of API communication (as in Section 3.1) with full interface specifications. Alternatively, and more ambitiously, one could first re-implement the required system-level functionality using a verified-by-construction approach such as the CakeML system [26, 58]; in this case, the binary used to execute the system-level part of the multilingual program would behave exactly as in its specification.

4 Multilingual Analysis: State of the Art

In tandem with the growing uptake of multilingual programming frameworks, there is a developing interest in understanding the limitations of "traditional" program analysis techniques on multilingual programs, and whether these programs are more likely harbor certain kinds of defects [1–3]. In this section, we discuss the most significant work in these areas.

A language-*agnostic* multilingual analysis is one that is not overly specific to a certain combination of host and guest languages, but is applicable, in principle, to a selection of multilingual scenarios. In contrast, a language-*specific* multilingual analysis takes into account the specific interaction mechanism of two programming languages within a certain programming framework—for instance, Java and JavaScript in hybrid Android apps. Accordingly, Section 4.1 discusses language-agnostic analyses, whereas Section 4.2 discusses language-specific analyses. Unsurprisingly, the classification in agnostic and specific is somewhat fuzzy, in that even a language-agnostic analysis is typically applicable only to certain kinds of multilingual frameworks (and is demonstrated on specific instances); however, language-agnostic analyses tend to be more abstract than language-specific ones.

For convenience, in this section we loosen up the host/guest terminology we have used, in a more restricted way, elsewhere in the paper. Namely, consider a program where a module H, written in some language ℓ_H, calls into another module G, written in some other language ℓ_G. Then, we call H the host (program); ℓ_H the host language; G the guest (program); and ℓ_G the guest language.

4.1 Language-Agnostic Multilingual Analysis

Lee et al. [29]'s approach works by first summarizing the behavior of interface operations implemented in the guest language in a way that only retains their effects within the host language modules. Then, any calls to these operations in the host are replaced by their summaries, and the host program plus summaries undergoes a standard whole program analysis. This approach's main intuition is that summarizing the effects at the interface with the host is sufficient to capture the key interaction between host and guest. Unfortunately, even such restricted summaries are challenging to construct for realistic, feature-laden programming languages. Besides, the precision of the whole program analysis performed on the host program with summaries may be reduced if the summaries fail to capture the nuances of guest-specific program optimizations.

Prakash et al. [45] propose a language-agnostic way of combining program analyses performed on two different languages. Their approach first analyzes the host and guest programs are analyzed in isolation (ignoring inter-language communication). Then, it builds an inter-language call graph that captures the points-to relation across language boundaries; this information is finally used to merge the host and guest analysis results.

Youn et al. [69] discuss how to apply to multilingual programs the declarative static analyses offered by the CodeQL vulnerability analyzer. The basic idea is to add constraints to a declarative analysis that capture inter-language communication. In particular, they show how to add constraints that approximate the inter-language points-to relation, so that the analysis can propagate the language-specific constraints from host to guest and vice versa. The main limitation of this approach is that it is only applicable to scenarios where exactly the same kind of analysis is available for both host and guest and can be implemented in CodeQL. This also excludes cases where the host and guest analysis have different granularities (e.g. object-sensitive vs. field sensitive).

Most language-agnostic frameworks are focused on *static* analysis, which entails that they can usually only analyze a restricted set of dynamic inter-language features. A recent study [22] suggests that dynamic features are widely used in Android hybrid apps: in particular, the JavaScript code run within a WebView Android component (the same mechanism described in Section 3.1) is often passed dynamically as a string. Another scenario that falls short of the capabilities of language-agnostic static analysis are inter-language communication via Android's intents.{u}{v} In these cases, language specific analyses have to be custom designed to achieve a better analysis precision.

4.2 Language-Specific Multilingual Analysis

Android hybrid apps have received a lot of attention as one of the most significant, and widespread, scenarios of multilingual programming. In Section 3.1 and Section 3.3, we already outlined two pairs of languages that are frequently used in the design of hybrid apps: Java-JavaScript and Java-C/native. Most program analysis work involving such apps focuses on three major areas: *i*) vulnerability detection; *ii*) checking of information flow security properties (e.g. noninterference); *iii*) detection of type-mismatch/crashing bugs. In the rest of this section, we discuss some relevant work in these areas.

Java-JavaScript hybrid apps. Detecting vulnerabilities that originate in hybrid Android apps using the WebView framework has been the object of plenty of recent work [10, 12, 17, 21, 33, 34, 36, 47, 52, 63, 68, 70].

Luo et al.'s work [33] was among the earliest demonstrations that the WebView component can be compromised, on the Java side, by injecting malicious JavaScript code. Zhang et al.'s large-scale study [70] of web-resource manipulation APIs in hybrid apps pointed to other kinds of vulnerabilities, which originate in the cross-principle manipulation of web resources. A key takeaway of this study is that attacks may occur when an app's code manipulates web resources, and app and resources do not originate from the same organization. Hidhaya et al. [21] demonstrated a so-called supplementary event-listener injection attack, where an attacker registers additional event listeners—whose callbacks execute malicious activities—with the HTML elements in a webpage that is then loaded by the WebView via JavaScript injection.

Bifocals [12] is a static analysis tool that can detect and mitigate some WebView vulnerabilities triggered by using JavaScript interfaces that load third-party web pages. The work on Bifocals also demonstrated man-in-the-middle attacks that can expose sensitive data if we allow executing JavaScript from untrusted source. In order to better evaluate the impact of code-injection attacks that target WebView components, BabelView [47] instruments hybrid apps with an attacker model that over-approximates the reach of possible attacks on the app's information flow. Mandal et al.'s static analysis approach [34] is specialized to detect vulnerabilities in Android infotainment apps.

Misusing WebView components may also result in information-flow security bugs. HybriDroid [28] is a framework for detecting such type bugs; it works by performing a taint analysis on an hybrid app's interprocedural control-flow graph that captures the inter-language information flow. This, in turn, is based on a formalization of hybrid communication in Android apps. Following a somewhat similar approach, Bae et al. [9] designed a type system that can detect bugs originating in the misuse of Android hybrid app communication APIs. In order to perform a large-scale study of how Android hybrid apps are programmed, the work on LuDroid [60] applied standard static analysis techniques to detect all flows of information from Android into JavaScript. They also showed that such an analysis is incomplete, as it can miss information-flow security bugs that originate inside JavaScript; and found numerous cases of apps that are susceptible to man-in-the-middle attacks since they load JavaScript code dynamically. In order to extend the range of issues that can be detected, the same authors proposed IWandroid, a technique that tracks sensitive data flows from Java to JavaScript and then to a public sink. To this end, IWandroid implements an IFDE/IDE analysis [50] to summarize Java-JavaScript interface communication. IWandroid can also detect integrity violations where the JavaScript environment modifies Android properties (a form of bidirectional inter-language communication, also discussed in Section 3.1).

Java-C/native hybrid apps. Analyzing hybrid Android apps where UI components are implemented in C (or other language that compiles to native) and connected to the Java host through JNI has also been a popular research topic [4, 6, 16, 29, 46, 51, 55, 56, 66].

JN-SAF [66] is an efficient analysis framework for JNI hybrid Android apps, implementing a summary-based bottom-up dataflow analysis that captures control and dataflow behavior of native components. More precisely, JN-SAF separately analyzes Java bytecode and binary code by symbolic analysis; then it combines the summaries of each analysis (which use a unified abstract heap model) to detect malicious behavior that may involve inter-language communication. One limitation of this approach is that its heap model is specific to the dataflow analysis JN-SAF supports; hence it may not support other, more general analyses of program behavior. To overcome this limitation, JuCify [51] performs an analysis in multiple steps: first, it uses symbolic execution to find calls that cross the bytecode-native boundary; then, it merges the call graph information about the native components into the Java bytecode; finally, it summarizes the semantics of the called native functions as bytecode statements. Underlying JuCify is a unified model of Android hybrid app code execution, which is considerably more detailed than the previous approaches'. Unfortunately, its detail is also a source of complexity that limits its scalability to realistic-size apps.

Other multilingual combinations. A few other program analysis work targets sundry combinations of languages used together, including Python-C, Java-Python, and Lua-C [31,35,45,65,69].

Monat et al. [35] proposed an abstract-interpretation based static analysis of Python programs with C extensions. Their work leverages the Mopsa static analysis framework to combine the Python and C abstract domains; concretely, they combined off-the-shelf value analyses for C and Python programs by means of a custom domain that gives an abstract semantic to operations at the inter-language boundary. PolyCruise [31] is another framework that can analyze Python-C programs. It is based on a dynamic flow analysis to find inter-language dependencies sources and sinks of sensitive data. PolyCruise can also be used *online*, while a program is running, to monitor its behavior and find vulnerabilities that may only occur in operating conditions.

Turcotte et al. [65] proposed a framework for analyzing Lua-C programs based on a simplified model of the guest language (i.e., C). To this end, it combines the host and guest type systems at the inter-language interface. While this type model abstracts away several details of the inter-language semantics, it fully captures statically any possible type errors that may ensue. Furthermore, after type checking determined that the inter-language interface is type safe, one can remove dynamic wrappers in Lua code, which contributes to improving the performance of the program. The approach is fairly general; however, applying it to realistic languages still requires some extra work, as one has to define a fully-typed foreign-function interface between languages (for example, the paper uses Poseidon Lua, a variant of Lua that offers gradual typing).

5 Discussion

In the paper, we discussed several challenges of multilingual program specification and analysis, and summarized some of the research in this area in the last decade or so.

A lot of progress has occurred *bottom-up*: the starting point were empirical studies (or even ad hoc observations) of concrete vulnerabilities that can be found in certain multilingual programs. The realization that a particular misbehavior may occur at the interface between two language runtimes motivated researchers to come up with bespoke combinations of existing program analyses that capture the semantics of such inter-language communication, and hence can detect the vulnerabilities automatically and on a large scale. Even though it is somewhat piecemeal, the bottom-up approach will remain fruitful in the future, as it caters to concrete, specific challenges—which are therefore more likely to be manageable, and have a clear practical relevance.

On the other hand, there is also room for more research in multilingual program analysis that is *top-down*: starting from the state of the art in modeling and reasoning about the semantics of *one* standalone programming language, researchers can explore extensions and combinations that cover the additional ground involving inter-language communication. An interesting direction for future work following the top-down approach could be leveraging frameworks that have been used to rigorously model the semantics of programming languages—one at a time—in particular "lightweight" techniques such as the K framework [32]{w} and PLT Redex [15]{x}. A top-down approach could push the boundaries of multilingual program analysis in a way that goes beyond the immediate, practical issues that are important for practitioners.

References

1. Mouna Abidi and Foutse Khomh. Towards the definition of patterns and code smells for multi-language systems. In *EuroPLoP '20: European Conference on Pattern Languages of Programs 2020, Virtual Event, Germany, 1-4 July, 2020*, pages 37:1–37:13. ACM, 2020.
2. Mouna Abidi, Md. Saidur Rahman, Moses Openja, and Foutse Khomh. Are multi-language design smells fault-prone? An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 30(3):29:1–29:56, 2021.
3. Mouna Abidi, Md. Saidur Rahman, Moses Openja, and Foutse Khomh. Multi-language design smells: a backstage perspective. *Empir. Softw. Eng.*, 27(5):116, 2022.
4. Vitor Monte Afonso, Paulo Lício de Geus, Antonio Bianchi, Yanick Fratantonio, Christopher Krügel, Giovanni Vigna, Adam Doupé, and Mario Polino. Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy. In *Network and Distributed System Security Symposium*, 2016.
5. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2nd edition, 2011.
6. Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. DroidNative: Automating and optimizing detection of Android native code malware variants. *Computers & Security*, 65:230–246, 2017.
7. Roberto Amadini. A survey on string constraint solving. *ACM Comput. Surv.*, 55(2):16:1–16:38, 2023.
8. Andrew W. Appel. *Modern compiler implementation*. Cambridge University Press, 2nd edition, 2002.
9. Sora Bae, Sungho Lee, and Sukyoung Ryu. Towards understanding and reasoning about Android interoperations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 223–233, 2019.
10. Achim D. Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 72–88, Cham, 2016. Springer International Publishing.
11. Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *Model Checking Software, 12th International*

SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005.

12. Erika Chin and David Wagner. Bifocals: Analyzing WebView vulnerabilities in Android applications. In *Revised Selected Papers of the 14th International Workshop on Information Security Applications - Volume 8267*, WISA 2013, pages 138–159, Berlin, Heidelberg, 2013. Springer-Verlag.
13. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.
14. Frank S. de Boer and Stijn de Gouw. Being and change: Reasoning about invariance. In *Correct System Design*, volume 9360 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2015.
15. Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
16. George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. Identifying Java calls in native code via binary scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 388–400, New York, NY, USA, 2020. Association for Computing Machinery.
17. Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in Android applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 377–396, May 2016.
18. Vijay Ganesh, Adam Kiezun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A string solver for testing, analysis and vulnerability detection. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2011.
19. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.
20. Philipp Haller and Ludvig Axelsson. Quantifying and explaining immutability in Scala. In Vasco T. Vasconcelos and Philipp Haller, editors, *Proceedings Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*, volume 246 of *EPTCS*, pages 21–27, 2017.
21. S Fouzul Hidhaya, Angelina Geetha, B Nandha Kumar, Loganathan Venkat Sravanth, and A Habeeb. Supplementary event-listener injection attack in smart phones. *KSII Transactions on Internet and Information Systems (TIIS)*, 9(10):4191–4203, 2015.
22. Jiajun Hu, Lili Wei, Yepang Liu, and Shing-Chi Cheung. wTest: WebView-oriented testing for Android applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 992–1004, New York, NY, USA, 2023. Association for Computing Machinery.
23. Sungjae Hwang, Sungho Lee, and Sukyoung Ryu. An empirical study of JVMs' behaviors on erroneous JNI interoperations. *IEEE Trans. Software Eng.*, 50(4):979–994, 2024.
24. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

25. Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. Towards efficient, multi-language dynamic taint analysis. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, pages 85–94, New York, NY, USA, 2019. Association for Computing Machinery.
26. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.
27. Julien Lancia. Detecting fault injection vulnerabilities in binaries with symbolic execution. In *14th International Conference on Electronics, Computers and Artificial Intelligence, ECAI 2022, Ploiesti, Romania, June 30 - July 1, 2022*, pages 1–8. IEEE, 2022.
28. Sungho Lee, Julian Dolby, and Sukyoung Ryu. HybriDroid: Static analysis framework for Android hybrid applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 250–261, New York, NY, USA, 2016. Association for Computing Machinery.
29. Sungho Lee, Hyogun Lee, and Sukyoung 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*, ASE '20, pages 127–137, New York, NY, USA, 2020. Association for Computing Machinery.
30. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, 2004.
31. Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. PolyCruise: A Cross-Language dynamic information flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2513–2530, Boston, MA, August 2022. USENIX Association.
32. Dorel Lucanu, Traian-Florin Serbanuta, and Grigore Rosu. K framework distilled. In *Rewriting Logic and Its Applications - 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24-25, 2012, Revised Selected Papers*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012.
33. Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 343–352, New York, NY, USA, 2011. Association for Computing Machinery.
34. Amit Kr Mandal, Agostino Cortesi, Pietro Ferrara, Federica Panarotto, and Fausto Spoto. Vulnerability analysis of Android auto infotainment apps. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 183–190. ACM, 2018.
35. Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of Python programs with native C extensions. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 323–345, Cham, 2021. Springer International Publishing.
36. Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*, volume 50, 2015.
37. James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.
38. Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen. Reasoning about function objects. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2010.

39. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 5th edition, 2021.
40. Victor Pankratius, Felix Schmidt, and Gilda Garretón. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 123–133. IEEE Computer Society, 2012.
41. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 247–258. ACM, 2005.
42. Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In *Proceedings of the 19th International Symposium on Formal Methods (FM)*, volume 8442 of *Lecture Notes in Computer Science*, pages 514–530. Springer, May 2014.
43. Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324, 2015.
44. Michael Pradel and Koushik Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
45. Jyoti Prakash, Abhishek Tiwari, and Christian Hammer. Unifying pointer analyses for polyglot inter-operations through summary specialization, 2023.
46. Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T.S. Chan. On tracking information flows through JNI in Android applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 180–191, 2014.
47. Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. BabelView: Evaluating the impact of code injection attacks in mobile webviews. 11050:25–46, 2018.
48. Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013*, pages 4:1–4:7. ACM, 2013.
49. Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 258–282. Springer, 2012.
50. Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.
51. Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. Jucify: A step towards Android code unification for enhanced static analysis, 2021.
52. Mohamed Shehab and Abeer AlJarrah. Reducing attack surface on cordova-based hybrid mobile apps. In *Proceedings of the 2Nd International Workshop on Mobile Development Lifecycle*, New York, NY, USA, 2014. ACM.
53. A. J. Summers, S. Drossopoulou, and P. Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.
54. Kwangwon Sun and Sukyoung Ryu. Analysis of JavaScript programs: Challenges and research trends. *ACM Comput. Surv.*, 50(4), aug 2017.
55. Mengtao Sun and Gang Tan. NativeGuard: protecting Android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in*

Wireless & Mobile Networks, WiSec '14, pages 165–176, New York, NY, USA, 2014. Association for Computing Machinery.

56. Mingshen Sun, Tao Wei, and John C.S. Lui. Taintart: A practical multi-level information-flow tracking system for Android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 331–342, New York, NY, USA, 2016. Association for Computing Machinery.
57. Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 365–378. USENIX Association, 2008.
58. Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.
59. Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel C. Briand. Search-driven string constraint solving for vulnerability detection. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 198–208. IEEE / ACM, 2017.
60. Abhishek Tiwari, Jyoti Prakash, Sascha Groß, and Christian Hammer. LUDroid: A large scale analysis of Android-web hybridization. In *2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 256–267, Los Alamitos, CA, USA, oct 2019. IEEE Computer Society.
61. Abhishek Tiwari, Jyoti Prakash, Sascha Groß, and Christian Hammer. A large scale analysis of Android-web hybridization. *Journal of Systems and Software*, 170:110775, 2020.
62. Abhishek Tiwari, Jyoti Prakash, and Christian Hammer. Demand-driven information flow analysis of WebView in Android hybrid apps. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*, pages 415–426. IEEE, 2023.
63. Abhishek Tiwari, Jyoti Prakash, Alimerdan Rahimov, and Christian Hammer. Understanding the impact of fingerprinting in Android hybrid apps. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 28–39, 2023.
64. Marco Trudel, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Really automatic scalable object-oriented reengineering. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lecture Notes in Computer Science*, pages 477–501. Springer, July 2013.
65. Alexi Turcotte, Ellen Arteca, and Gregor Richards. Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:32, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
66. Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong 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*, CCS '18, pages 1137–1150, New York, NY, USA, 2018. Association for Computing Machinery.
67. Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021.
68. Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 742–755. IEEE, 2018.

69. Dongjun Youn, Sungho Lee, and Sukyoung Ryu. Declarative static analysis for multilingual programs using CodeQL. *Software: Practice and Experience*, 53(7):1472–1495, March 2023.
70. Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, XiaoFeng Wang, Long Lu, and Haixin Duan. An empirical study of web resource manipulation in real-world mobile applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1183–1198, Baltimore, MD, August 2018. USENIX Association.

URL References

a. An example of URL reference: `https://bugcounting.net`
b. `https://developer.android.com/reference/android/webkit/WebView`
c. `https://developer.android.com/reference/android/webkit/JavascriptInterface`
d. Mobile Developer Survey `https://ionicframework.com/survey/2017#trends`
e. `https://developer.android.com/develop/ui/views/layout/webapps/webview#BindingJavaScript`
f. `https://github.com/gustavogenovese/androidSamples/blob/036c1c6f9bc35392d2c18d6876073b13c31a47b8/webview/src/main/java/com/gustavogenovese/webview/SynchronousJavascriptInterface.java#L41`
g. `https://apkcombo.com/ko/mas-que-panes-y-peces/com.a2stacks.apps.app57191abb7ab09/`
h. `https://apkcombo.com/zh/sally-s-makeup-salon/com.nuttyapps.sally.makeup.salon/`
i. `https://www.9apps.com/ar/android-apps/Social-network-circus/`
j. `https://docs.scala-lang.org/overviews/collections-2.13/conversions-between-java-and-scala-collections.html`
k. `https://github.com/scala/scala/blob/06a7509e3a9083793038bf4449281491a8614eb0/src/library/scala/collection/convert/JavaCollectionWrappers.scala#L138`
l. `https://www.scala-lang.org/api/2.12.6/scala/collection/mutable/AbstractBuffer.html`
m. `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html`
n. `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html#of(E...)`
o. `https://blog.jetbrains.com/idea/2018/01/fumigating-the-idea-ultimate-code-using-dataflow-analysis/`
p. `https://www.jetbrains.com/help/idea/analyzing-data-flow.html#analyze-stack-traces`
q. `https://blog.jetbrains.com/scala/2021/10/28/dataflow-analysis-for-scala/`
r. `https://blog.jetbrains.com/scala/2021/10/28/dataflow-analysis-for-scala/#interprocedural-analysis`
s. `https://github.com/mkowsiak/jnicookbook/`
t. `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html`
u. `https://github.com/arguslab/NativeFlowBench/tree/master/icc_javatonative`
v. `https://github.com/arguslab/NativeFlowBench/tree/master/icc_nativetojava`
w. The K Semantic Framework `https://kframework.org/`
x. PLT Redex `https://redex.racket-lang.org/`