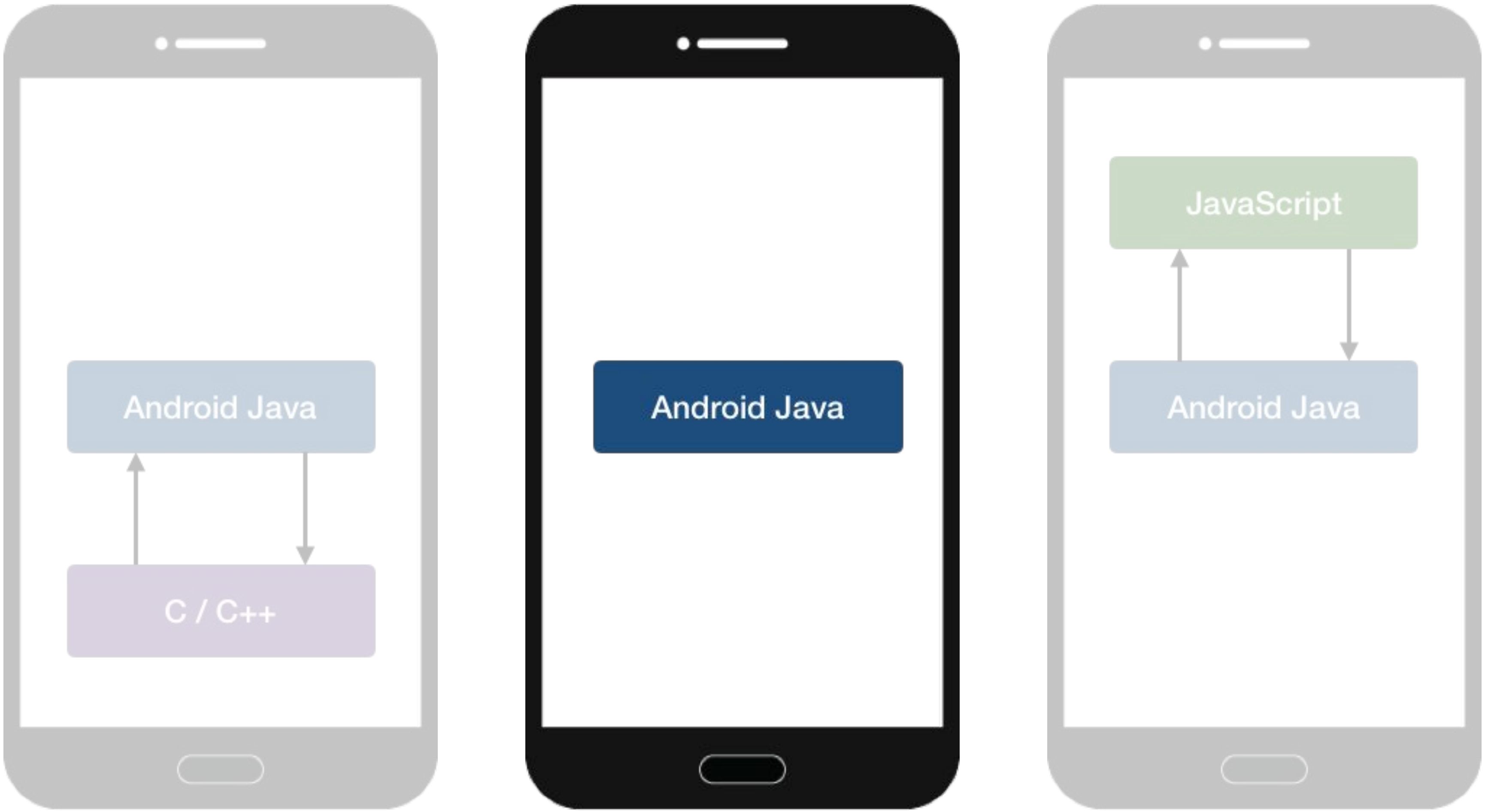
Static Analysis for Multilingual Android Apps

2021. 02. 01. Sungho Lee @ KCSE'21

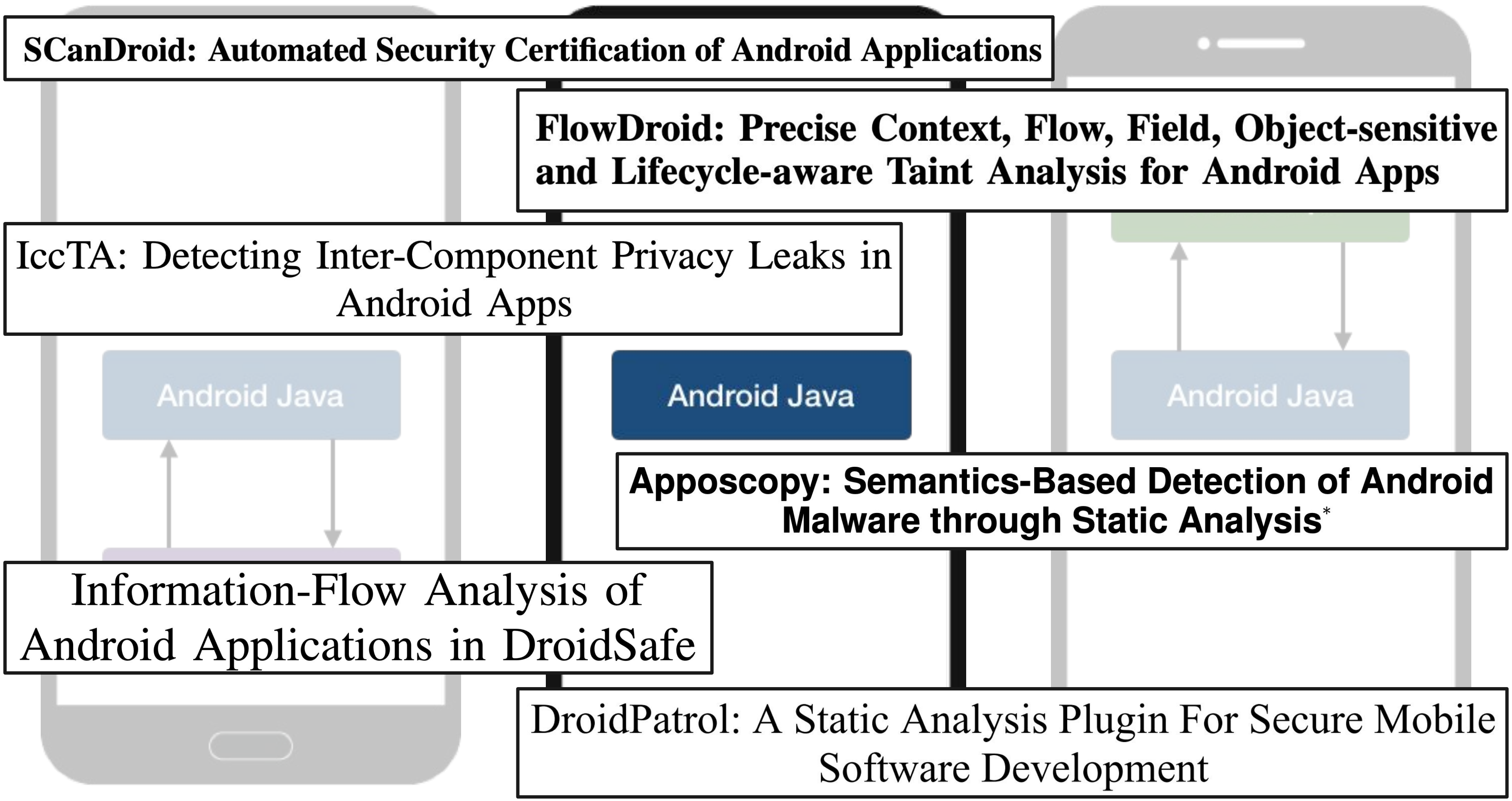
Profile

- Education
  - B.S. @ Ajou Univ.
  - M.S. and Ph.D. @ KAIST
    - Majoring in Programming Language (especially, static analysis)
- Working experience
  - Visiting faculty researcher @ Google
    - 1st Visiting Faculty Researcher in APAC
    - Deep-learning compiler validation
    - Hypervisor verification for Android system
  - Assistant professor @ CNU (*present*)
- Software Analysis and Testing Laboratory (SW@)
  - https://sites.google.com/view/sat-lab/home

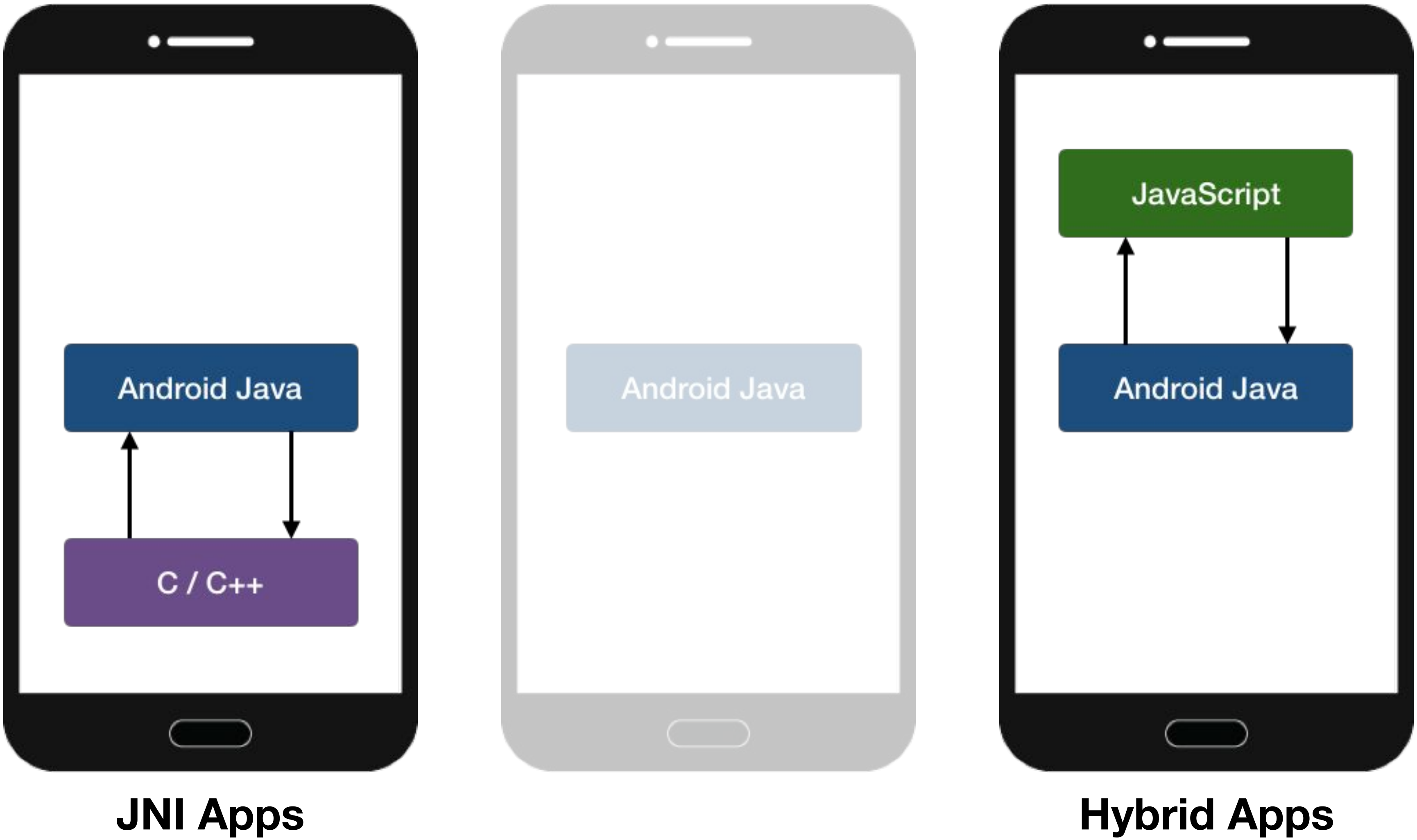
Three Types of Android Apps



Three Types of Android Apps



Three Types of Android Apps



Three Types of Android Apps

"By 2016, more than 50% of mobile apps deployed will be hybrid"

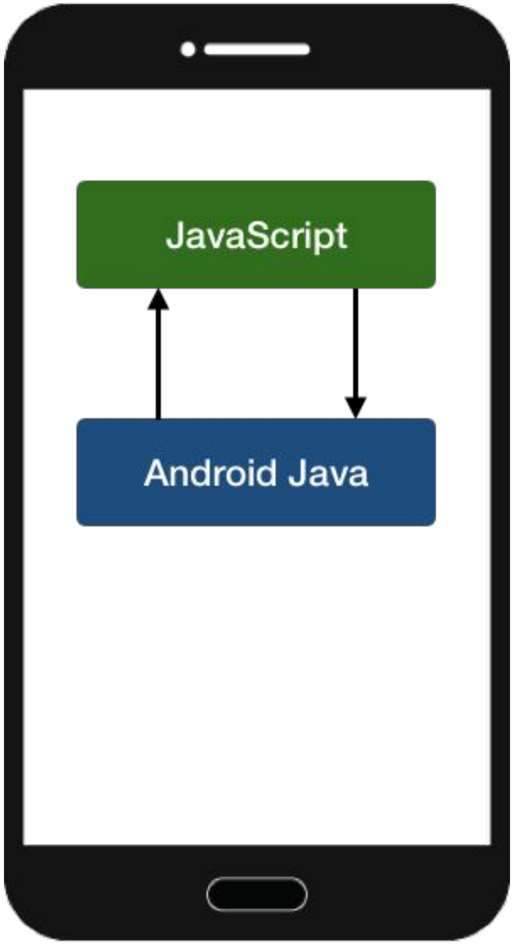
Gartner

source: http://www.gartner.com/newsroom/i d/2324917

"32.7% of developers surveyed expect to completely abandon native development in favor of hybrid."

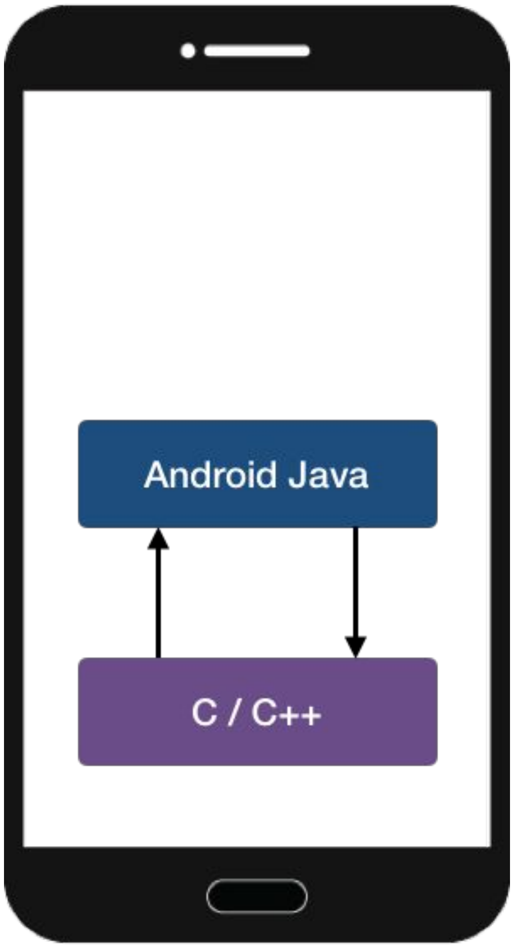
Ionic Developer Survey 2017

source: https://ionicframework.com/survey/2017#trends



Hybrid Apps

Three Types of Android Apps



JNI Apps

“there is substantial usage (39.7%) of native code”

JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code (CCS’18)

“446,562 apps (37.0%) used at least one of the previously mentioned ways of executing native code”

Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy (NDSS’16)

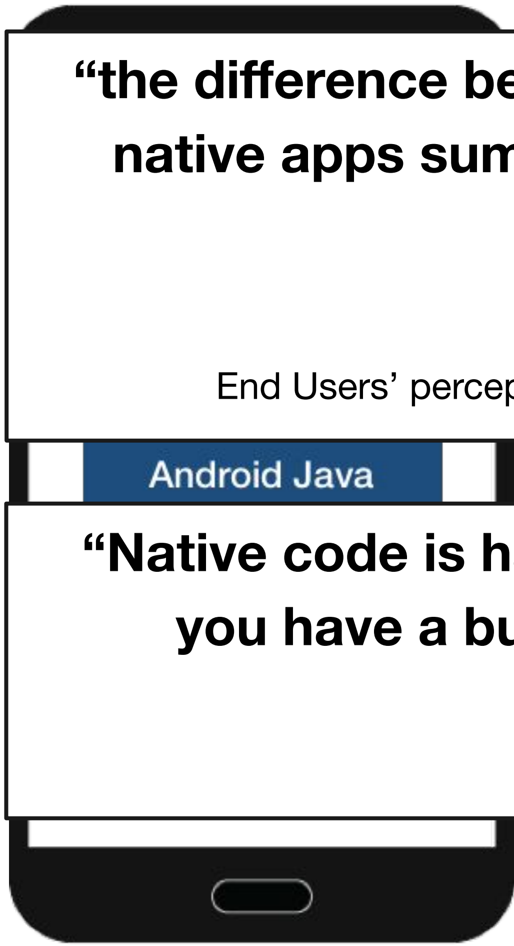
Three Types of Android Apps

“the difference between the perceived bugginess of hybrid and native apps sums up to ~18.42 points with a higher value for hybrid apps”

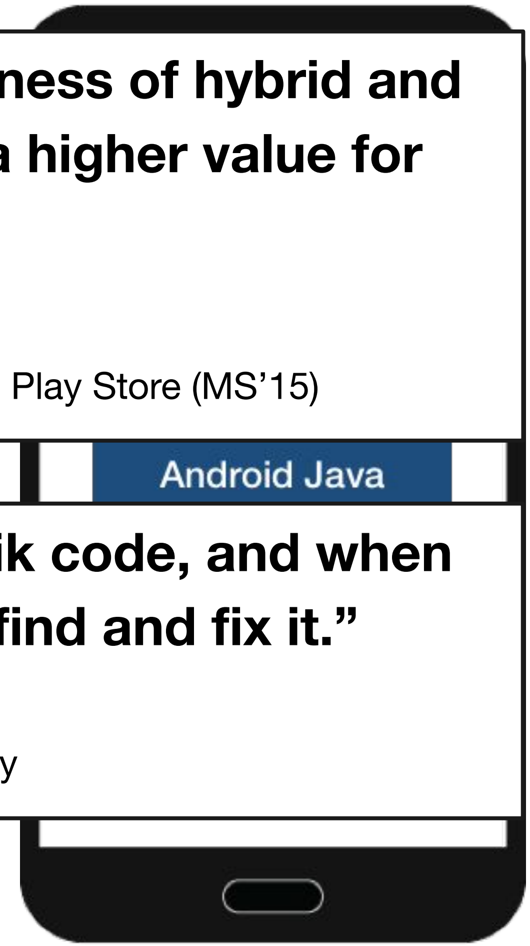
End Users’ perception of Hybrid Mobile Apps in the Google Play Store (MS’15)

“Native code is harder to get right than Dalvik code, and when you have a bug, it’s often a lot harder to find and fix it.”

Android Developer Official Blog - Tim Bray



JNI Apps



Hybrid Apps

Bug and Security Vulnerability Detection

in **Multilingual Android apps**

Composing Static Analyzers for

Bug and Security Vulnerability Detection

in **Multilingual Android apps**

Android hybrid app analysis

1) **HybriDroid: Static Analysis Framework for Android Hybrid Applications (ASE'16)**
2) **Towards understanding and reasoning about Android interoperations (ICSE'19)**
3) Adlib: Analyzer for Mobile Ad Platform Libraries (ISSTA'19)

Composing Static Analyzers for **Bug and Security Vulnerability** Detection in **Multilingual Android apps**

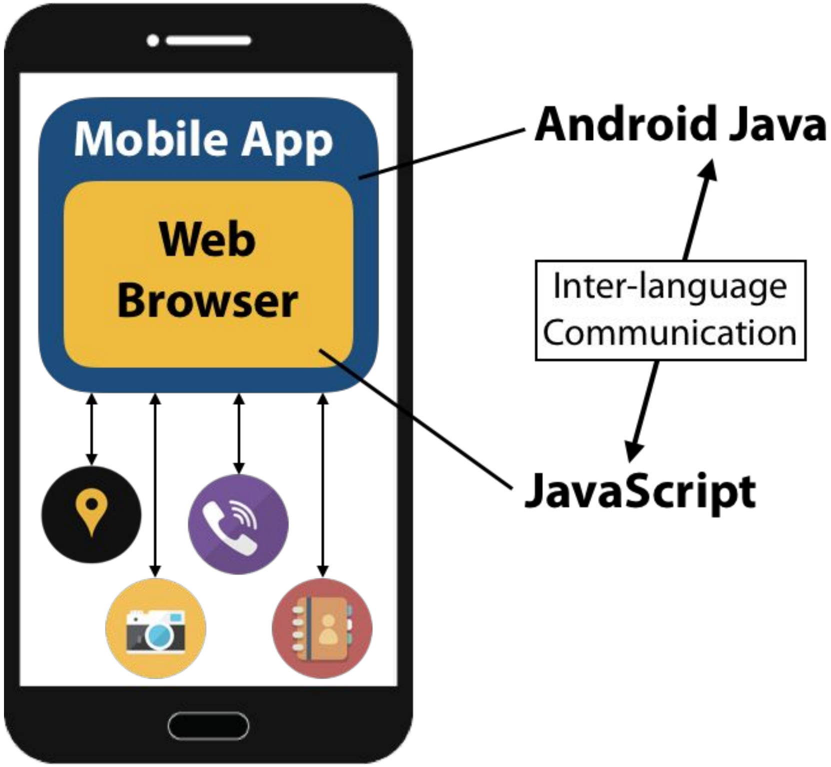
Android JNI app analysis

4) **JNI program analysis with automatically extracted C semantic summary (ISSTA'19 DS)**
5) **Broadening Horizons of Multilingual Program Analysis: Semantic Summary Extraction for JNI Program Analysis (ASE'20)**
6) JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs (ICSE'21)

Static Analysis for Android Hybrid Applications

ASE'16 & ICSE'19

Android Hybrid Apps



Interoperation: Java - JavaScript

Android Java

```
class JSApp{
  @JavascriptInterface
  public int alert(String m){
    ...
  }
}

...

addJavascriptInterface(
         new JSApp(), “app”);
```

Java Bridge

JavaScript

```
app.alert(“Hello Hybrid”);
```

JavaScript Bridge

Interoperation: Java - JavaScript

Android Java

```
class JSApp{
  @JavascriptInterface
  public int alert(String m){
    ...
  }
}

...

addJavascriptInterface(
         new JSApp(), “app”);
```

Java Bridge

JavaScript

```
app.alert(“Hello Hybrid”);
```

JavaScript Bridge

Differences between Java and JavaScript

Android Java

Chapter 4. Types, Values, and Variables

The Java programming language is a *statically typed* language, which means that every variable and every expression has a type that is known at compile time.

The Java programming language is also a *strongly typed* language, because types limit the values that a variable (§4.12) can hold or that an expression can produce,

The types of the Java programming language are divided into two categories: primitive types and reference types. The primitive types (§4.2) are the `boolean` type an special null type. An object (§4.3.1) is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to o

≠

JavaScript

6 ECMAScript Data Types and Values

Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this clause. Types are further subclassified into ECM

Within this specification, the notation "Type(*x*)" is used as shorthand for "the *type* of *x*" where "type" refers to the ECMAScript language and specification types defined in this clause. When the term equivalent to saying "no value of any type".

Differences between Java and JavaScript

Android Java

8.4.9. Overloading

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name

This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the

When a method is invoked (§15.12), the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are

JavaScript

Buggy Interoperation (1)

Android Java

```
class JSApp{
  @JavascriptInterface
  public int divide(int x, int y){
    return x/y;
  }
}

...

addJavascriptInterface(
         new JSApp(), “app”);
```

Divide by zero?

JavaScript

```
var list = [0, 1, 2, 3, 4];
var a = list[3];
var b = list[?];

if( b !== 0 )
  app.divide(a, b);
```

Buggy Interoperation (1)

Android Java

```
class JSApp{
  @JavascriptInterface
  public int divide(int x, int y){
    return x/y;
  }
}

...

addJavascriptInterface(
         new JSApp(), “app”);
```

y = 0

Divide by zero!

JavaScript

```
var list = [0, 1, 2, 3, 4];
var a = list[3];
var b = list[5];

if( b !== 0 )
  app.divide(a, b);
```

b = undefined

Buggy Interoperation (2)

Android Java

```
class JSBridge{
  @JavascriptInterface
  public void sendName(String a){
    ...
  }

  @JavascriptInterface
  public void sendName(int a){
    ...
  }
}

addJavascriptInterface(
         new JSBridge(), “app”);
```

JavaScript

```
app.sendName(“Sungho”);
```

Buggy Interoperation (2)

Android Java

```
class JSBridge{
  @JavascriptInterface
  public void sendName(String a){
    ...
  }

  @JavascriptInterface
  public void sendName(int a){
    ...
  }
}

addJavascriptInterface(
         new JSBridge(), “app”);
```

JavaScript

```
app.sendName(“Sungho”);
```

Buggy Interoperation (3)

Android Java

```
class JSBridge1{
  @JavascriptInterface
  public void getName(){
    return “Sungho”;
  }

class JSBridge2{
  @JavascriptInterface
  public void getName(){
    return “Sora”;
  }
}

addJavascriptInterface(
         new JSBridge1(), “app1”);
addJavascriptInterface(
         new JSBridge2(), “app2”);
```

JavaScript

```
app2.f = app1.getName;
app2.f();
```

Buggy Interoperation (3)

Android Java

```
class JSBridge1{
  @JavascriptInterface
  public void getName(){
    return “Sungho”;
  }

class JSBridge2{
  @JavascriptInterface
  public void getName(){
    return “Sora”;
  }
}

addJavascriptInterface(
         new JSBridge1(), “app1”);
addJavascriptInterface(
         new JSBridge2(), “app2”);
```

JavaScript

```
app2.f = app1.getName;
app2.f();
```

Interoperation Semantics for Hybrid Apps

Operational Semantics for Multi-Language Programs

JACOB MATTHEWS and ROBERT BRUCE FINDLER
University of Chicago

Interoperability is big business, a fact to which .NET, the JVM, and COM can attest. Language designers are well aware of this, and they are designing programming languages that reflect it—for instance, SML.NET, F#, Mondrian, and Scala all treat interoperability as a central design feature. Still, current multi-language research tends not to focus on the semantics of these features, but only on how to implement them efficiently. In this article, we attempt to rectify that by giving a technique for specifying the operational semantics of a multi-language system as a composition of the models of its constituent languages. Our technique abstracts away the low-level details of interoperability like garbage collection and representation coherence, and lets us focus on semantic properties like type-safety, equivalence, and termination behavior. In doing so it allows us to adapt standard theoretical techniques such as subject-reduction, logical relations, and operational equivalence for use on multi-language systems. Generally speaking, our proofs of properties in a multi-language context are mutually referential versions of their single language counterparts.

We demonstrate our technique with a series of strategies for embedding a Scheme-like language into an ML-like language. We start by connecting very simple languages with a very simple strategy, and work our way up to languages that interact in sophisticated ways and have sophisticated features such as polymorphism and effects. Along the way, we prove relevant results such as type-soundness and termination for each system we present using adaptations of standard techniques.

Beyond giving simple expressive models, our studies have uncovered several interesting facts about interoperability. For example, higher-order function contracts naturally emerge as the glue to ensure that interoperating languages respect each other's type systems. Our models also predict that the embedding strategy where foreign values are opaque is as expressive as the embedding strategy where foreign values are translated to corresponding values in the other language, and we were able to experimentally verify this behavior using PLT Scheme's foreign function interface.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.2.12 [**Software Engineering**]: Interoperability

General Terms: Languages

Additional Key Words and Phrases: Operational semantics, interoperability

This work was funded in part by the National Science Foundation.
J. Matthews is currently at Google Irvine and R. B. Findler is currently at Northwestern University.
Authors' address: J. Matthews; email: jcobm@cs.chicago.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 0164-0925/2009/04-ART12 $5.00
DOI 10.1145/1498926.1498930 http://doi.acm.org/10.1145/1498926.1498930

ACM Transactions on Programming Languages and Systems, Vol. 31, No. 3, Article 12, Pub. date: April 2009.

Operational Semantics for Multi-Language Programs (TOPLAS'09)

- Formalization for interoperations with explicit language boundaries between ML-like and Scheme-like languages

$$\mathbf{e} = \cdots | ({}^{\tau}MS\ e)$$

$$e = \cdots | (SM^{\tau}\ \mathbf{e})$$

Interoperation Semantics for Hybrid Apps

Android Java

```
class JSApp1{
    @JavascriptInterface
    public int send(String msg){...}
}

class JSApp2{
    @JavascriptInterface
    public int send(String msg){...}
}
...
if(?)
    addJavascriptInterface(
                new JSApp1(), “app”);
else
    addJavascriptInterface(
                new JSApp2(), “app”);
```

JavaScript

Dynamic determination

```
var msg = handler.getMsg();

app.send(msg);
```

Interoperation Semantics for Hybrid Apps

Android Java

```
class JSApp1{
    @JavascriptInterface
    public int send(String msg){...}
}

class JSApp2{
    @JavascriptInterface
    public int send(String msg){...}
}
...
if(?)
    addJavascriptInterface(
                new JSApp1(), “app”);
else
    addJavascriptInterface(
                new JSApp2(), “app”);
```

JavaScript

Indistinguishable JS bridge

```
var msg = handler.getMsg();
app.send(msg);
```

?

Interoperation Semantics for Hybrid Apps

$$\mathcal{B}\,\mathcal{O}\,\mathcal{E}[\mathsf{e}] \rightarrow \ldots \rightarrow \mathcal{B}'\,\mathcal{O}'\,\mathcal{E}[\underline{\mathtt{SV}^{\tau^{\mathtt{v}}}(\mathsf{e}')}] \rightarrow \ldots \rightarrow \mathcal{B}''\,\mathcal{O}''\,\mathcal{E}[\mathtt{v}]$$

Explicit Language Boundary

$$\mathcal{B} = O \mapsto \mathbf{O}$$

$$\mathcal{O} = O \times F \mapsto (V \ \cup \ \mathbf{M})$$

O = JavaScript Object

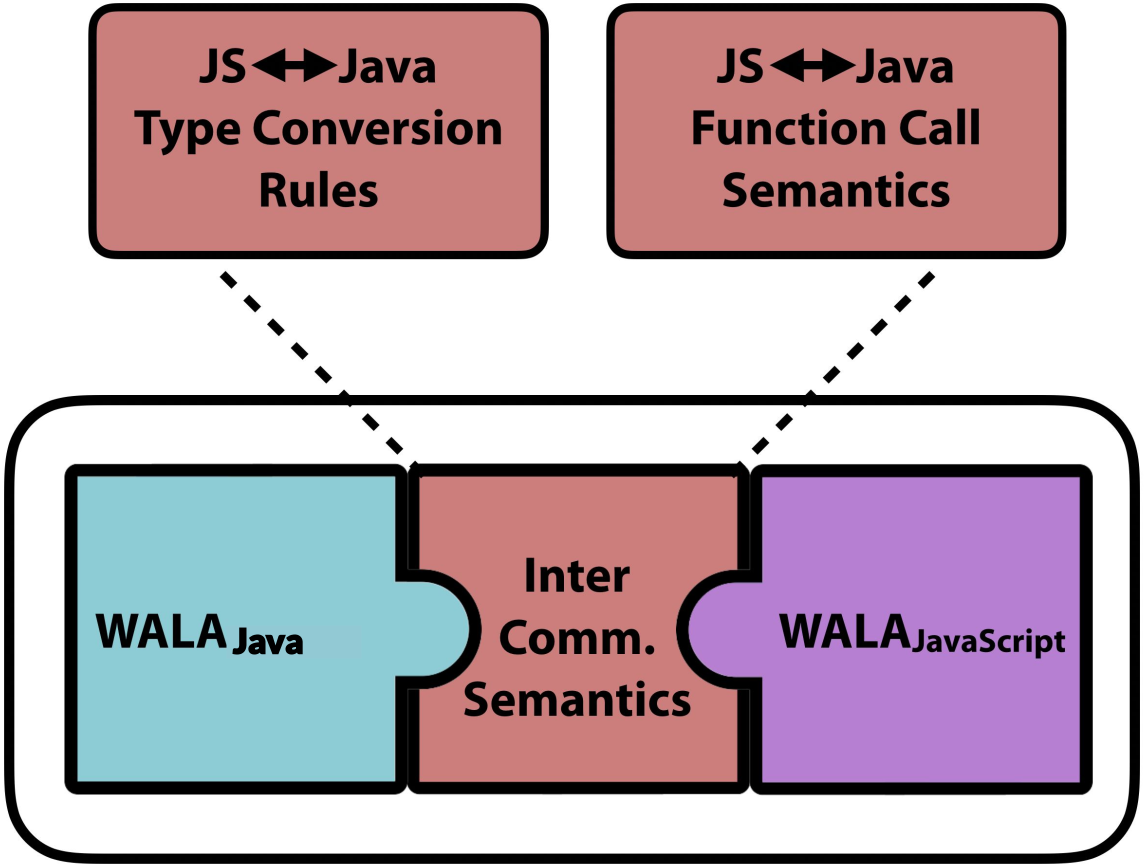
$\mathbf{O}$ = Java Object

F = JavaScript Field

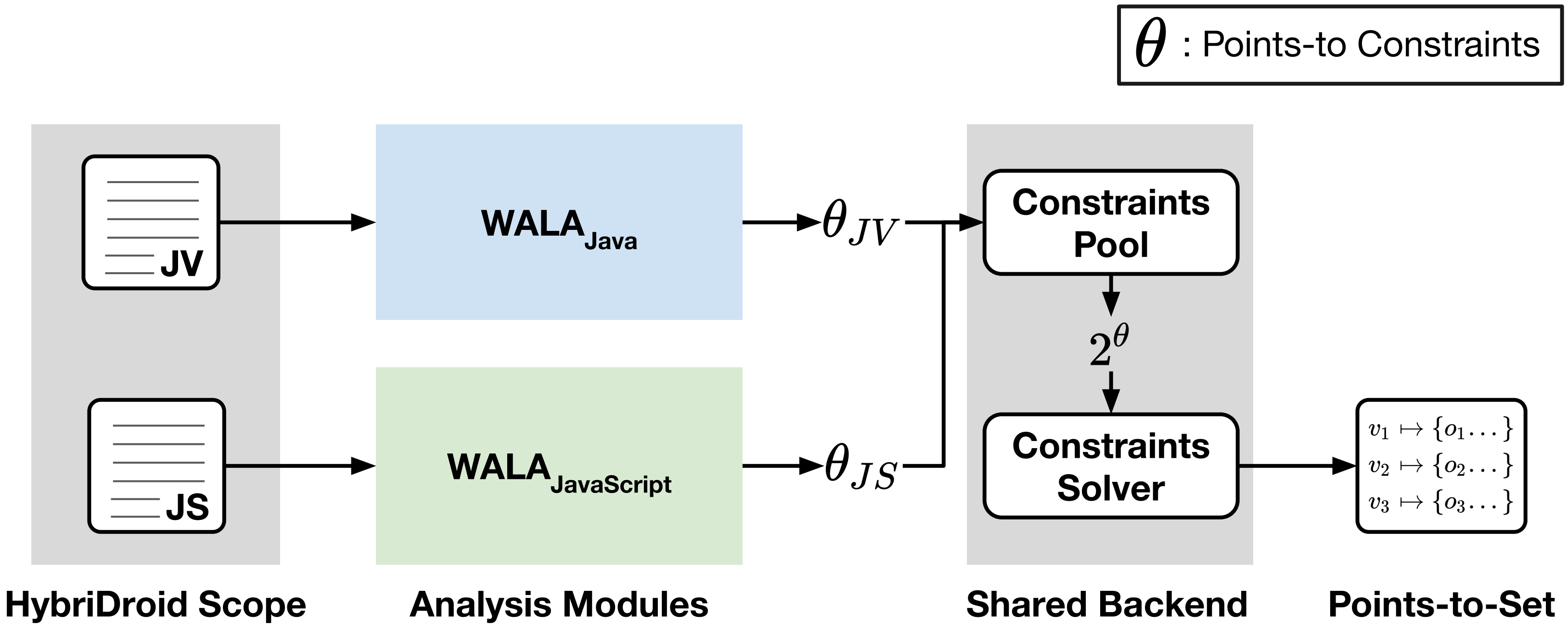
V = JavaScript Value

$\mathbf{M}$ = Java Method

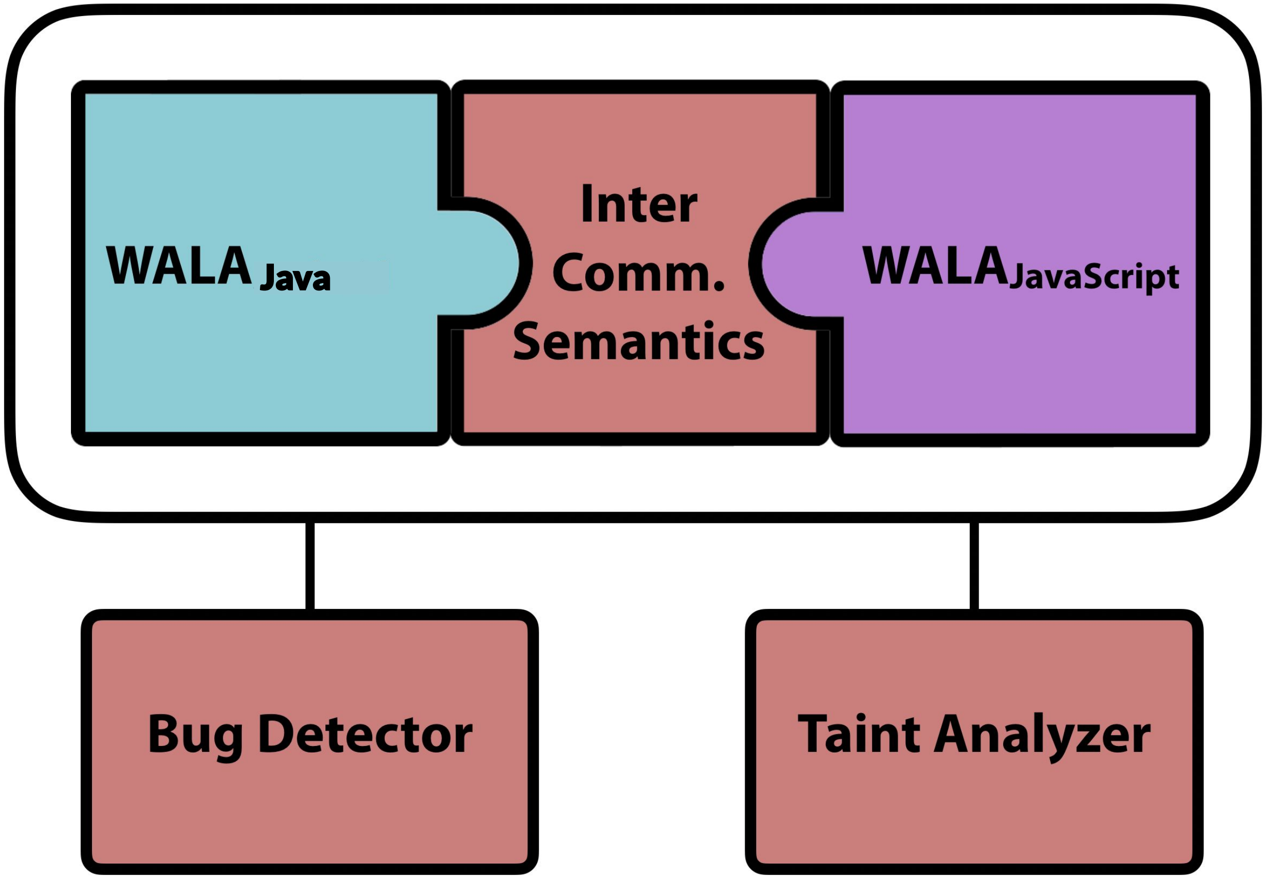
HybriDroid: Overview



HybriDroid: Analysis Model



HybriDroid: Client Analyses



Bug Detection: MethodNotFound

Android Java

```
class JSApp{
  @JavascriptInterface
  public int alert(String m){
    ...
  }
}

...

addJavascriptInterface(
         new JSApp(), “app”);
```

Java Bridge

JavaScript

```
app.alert(“Hello Hybrid”, 3);
```

JavaScript Bridge

Bug Detection: MethodNotFound

Android Java

JavaScript

MethodNotFound Exception

```
class JSApp{
  @JavascriptInterface
  public int alert(String m){
    ...
  }
}

...

addJavascriptInterface(
         new JSApp(), "app");
```

Java Bridge

```
app.alert("Hello Hybrid", 3);
```

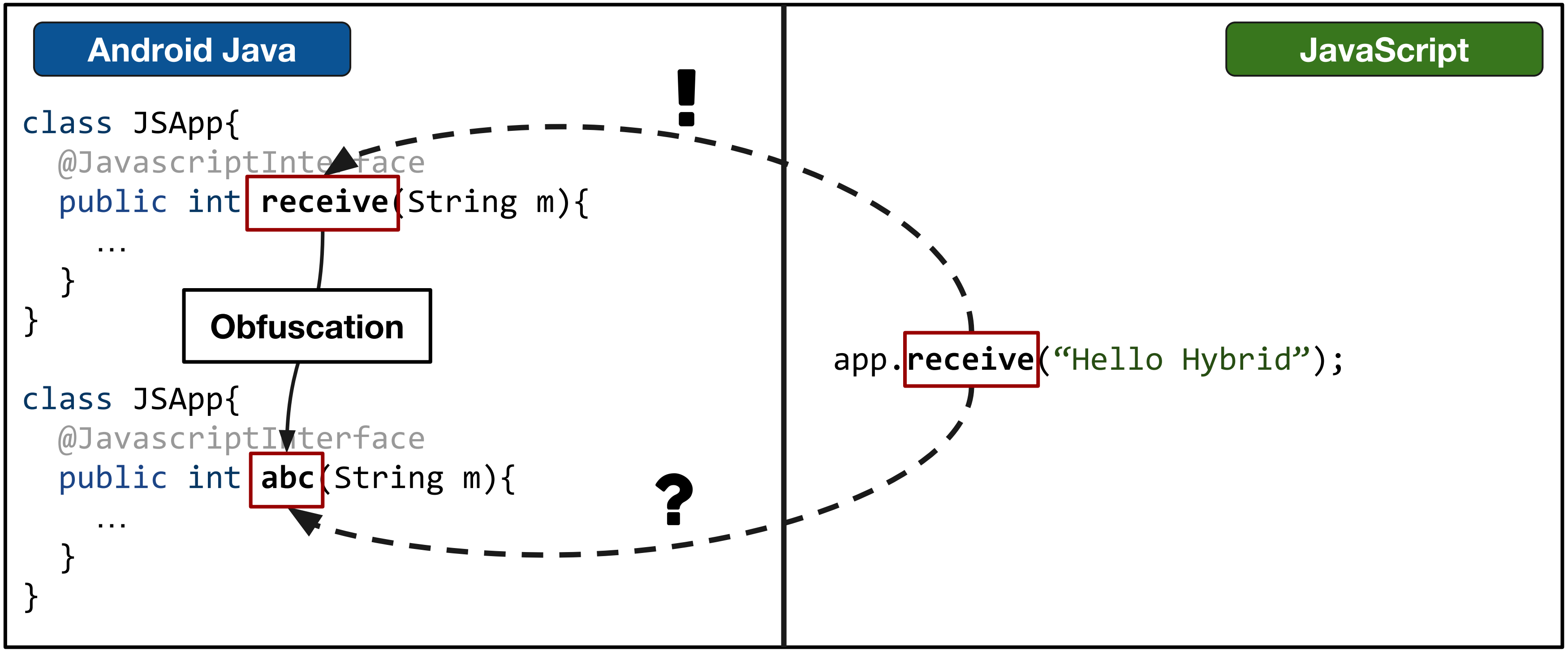
JavaScript
Bridge

Bug Detection: Results

Hybrid App	Bug Type (#)	#FP	#TP
com.gameloft.android.ANMP.GloftDMHM	MethodNotFound (1)	0	1
com.creativemobile.DragRacing	MethodNotFound (1)	1	0
com.gau.go.launcherex	MethodNotFound (2)	2	0
com.tripadvisor.tripadvisor	MethodNotFound (1)	0	1
com.dianxinos.dxbs	MethodNotFound (1)	0	1
com.magmamobile.game.LostWords	MethodNotFound (1)	1	0
com.daishin	MethodNotFound (1)	0	1
com.carezone.caredroid.careapp	MethodNotFound (5)	0	5
com.pateam.kanomthai	MethodNotFound (2)	0	2
com.acc5.16	MethodNotFound (6)	0	6
jp.cleanup.android	MethodNotFound (1)	1	0
ligamexicana.futbol	MethodNotFound (2)	2	0
com.sysapk.weighter	MethodNotFound (1)	0	1
com.youmustescape3guide.free	MethodNotFound (6)	0	6
Total	MethodNotFound (31)	7	24

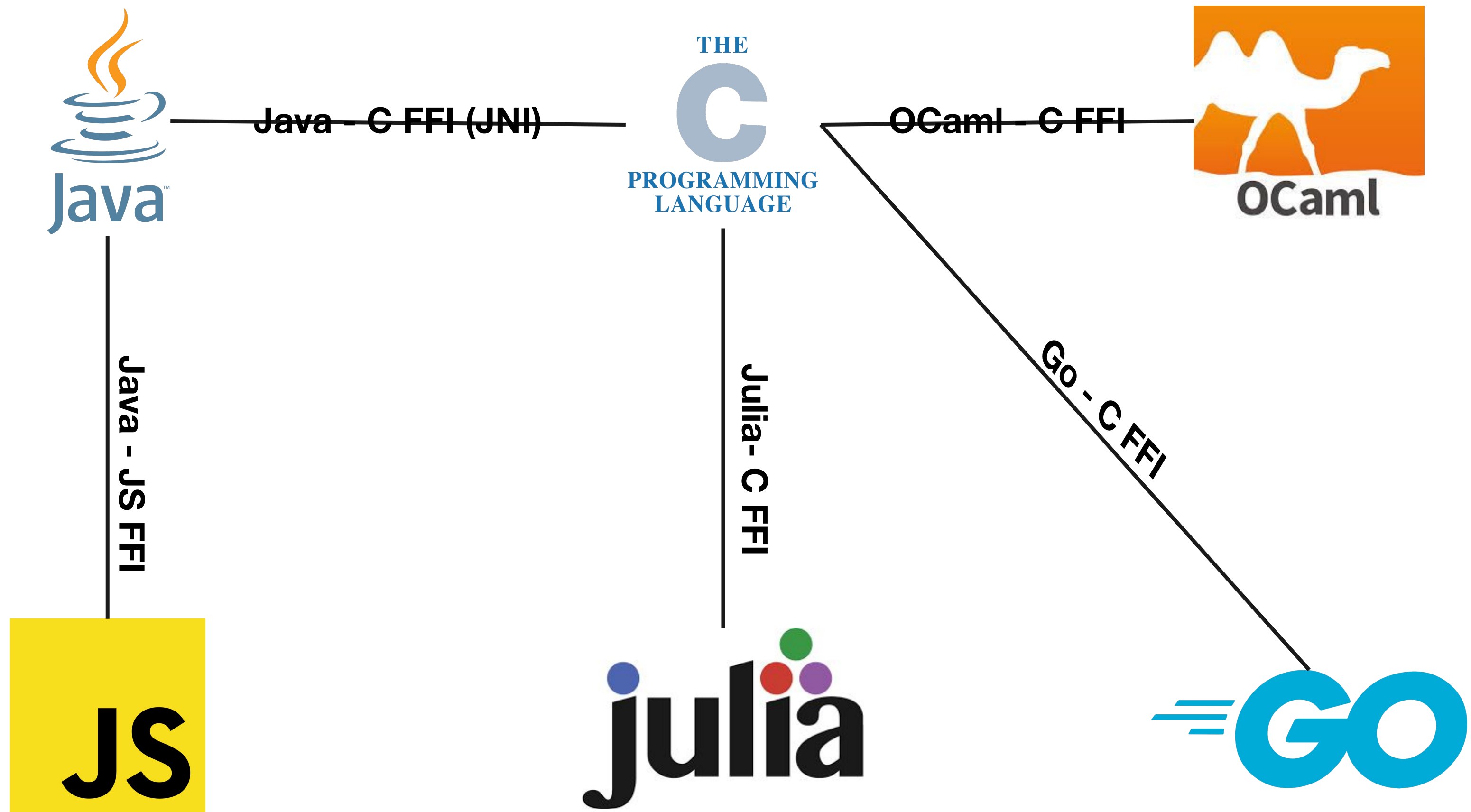
Bug Detection: Results

Hybrid App	Bug Type (#)	#FP	#TP	Bug Cause (#)
com.gameloft.android.ANMP.GloftDMHM	MethodNotFound (1)	0	1	Obfuscation (1)
com.creativemobile.DragRacing	MethodNotFound (1)	1	0	
com.gau.go.launcherex	MethodNotFound (2)	2	0	
com.tripadvisor.tripadvisor	MethodNotFound (1)	0	1	Obfuscation (1)
com.dianxinos.dxbs	MethodNotFound (1)	0	1	Obfuscation (1)

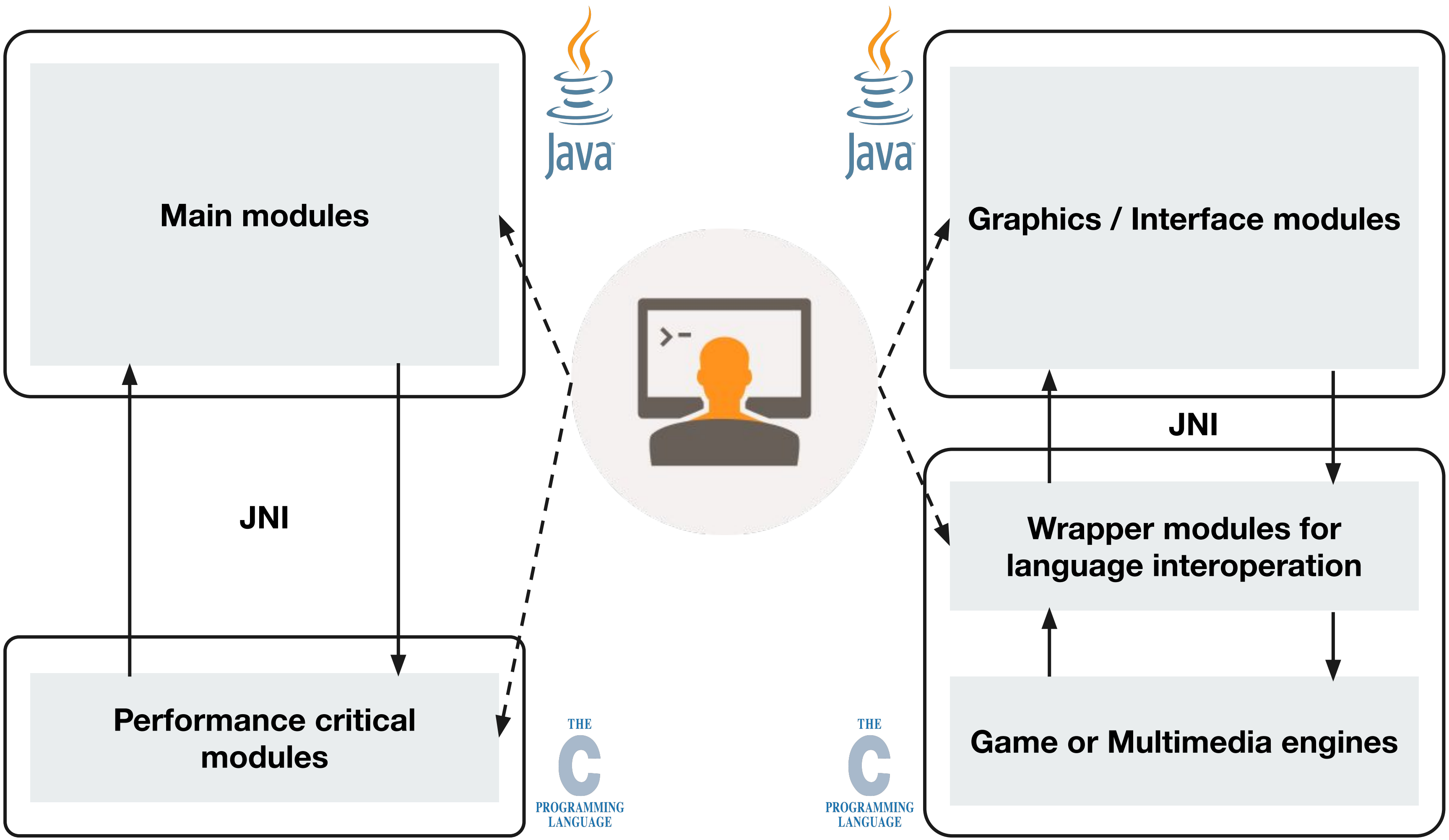


Static Analysis for JNI Programs
ISSTA'19 DS & ASE'20

Multilingual programs



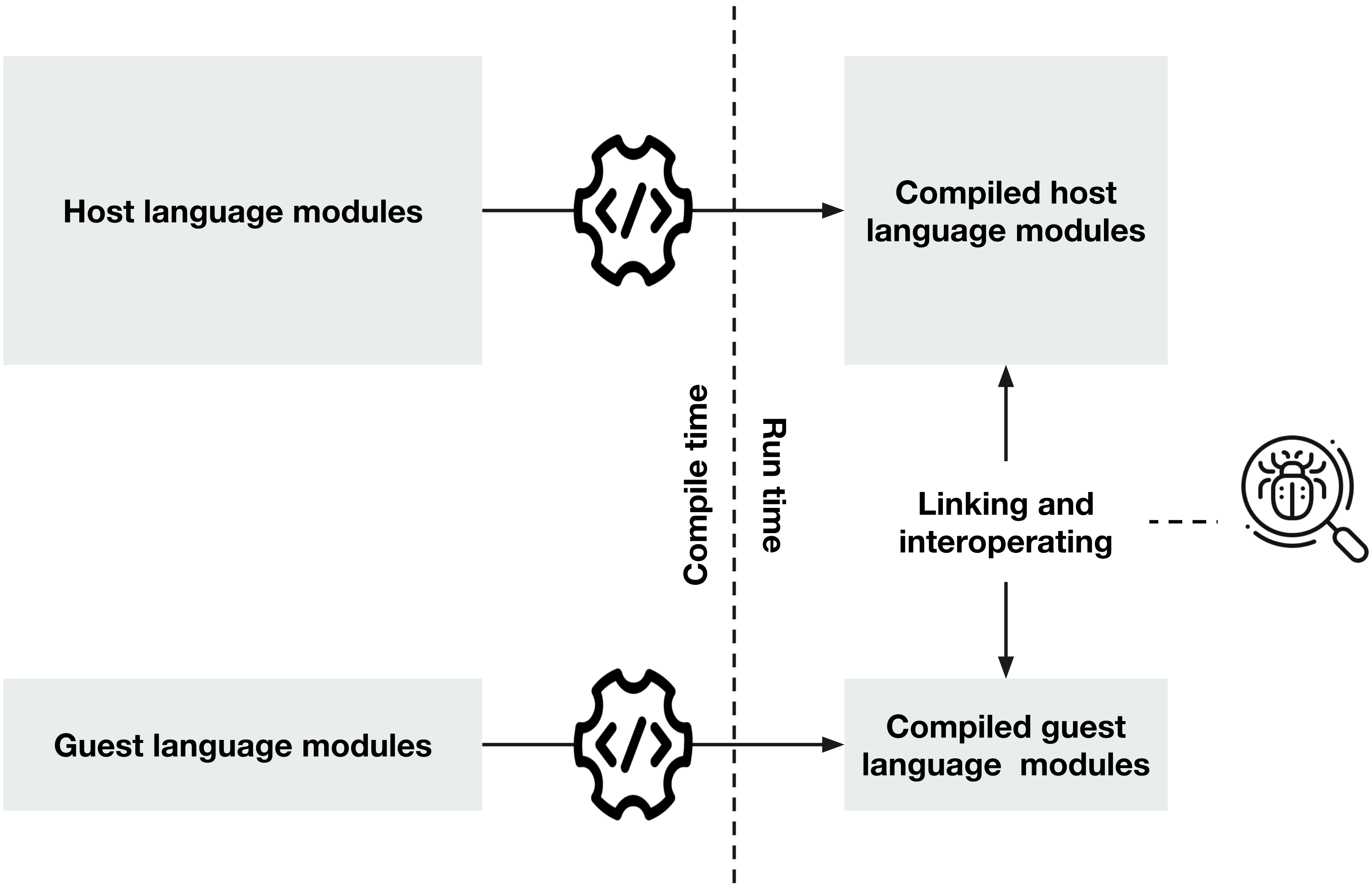
Advantages: performance and reusability



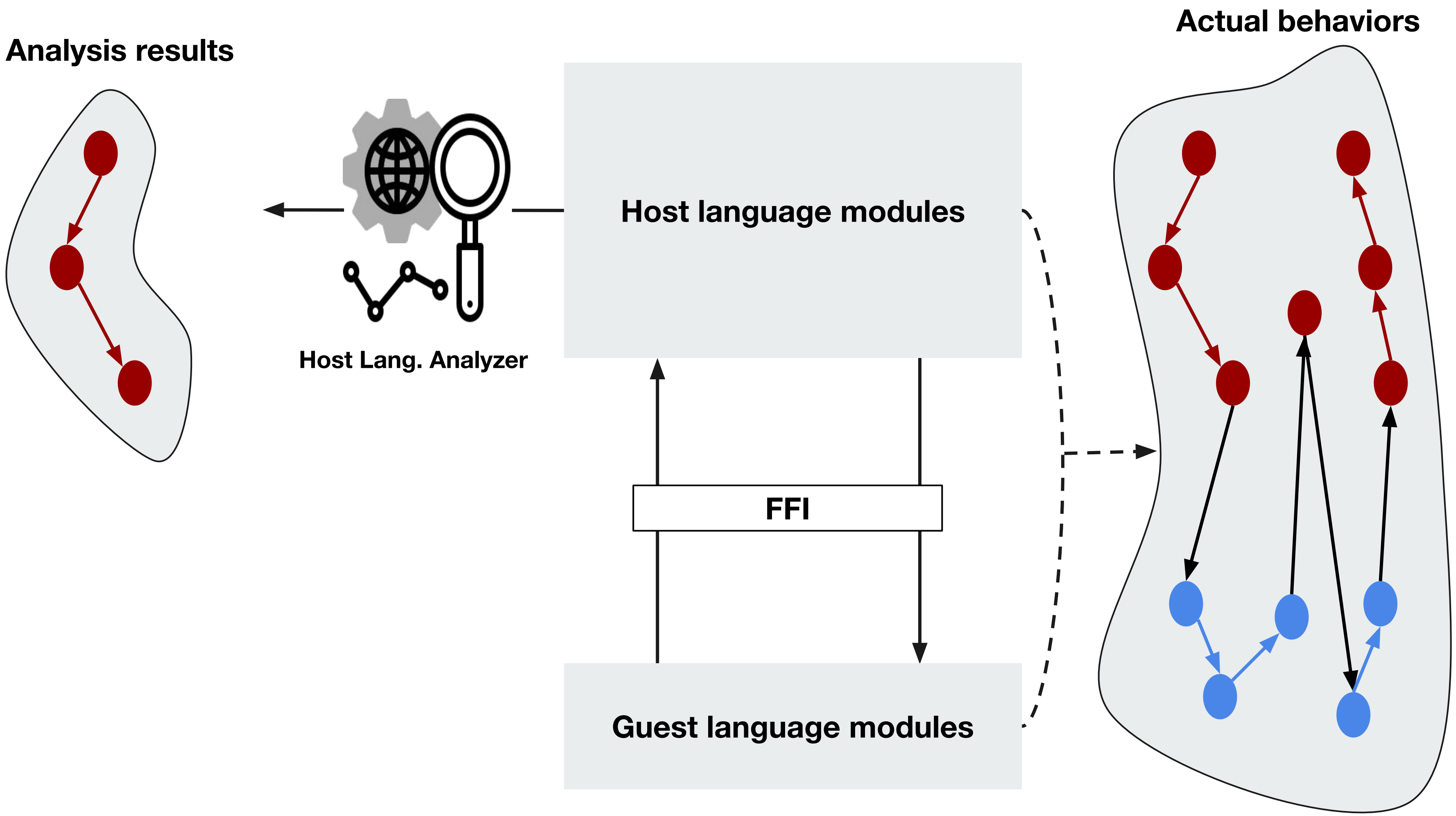
Improving performance

Reusing existing libraries

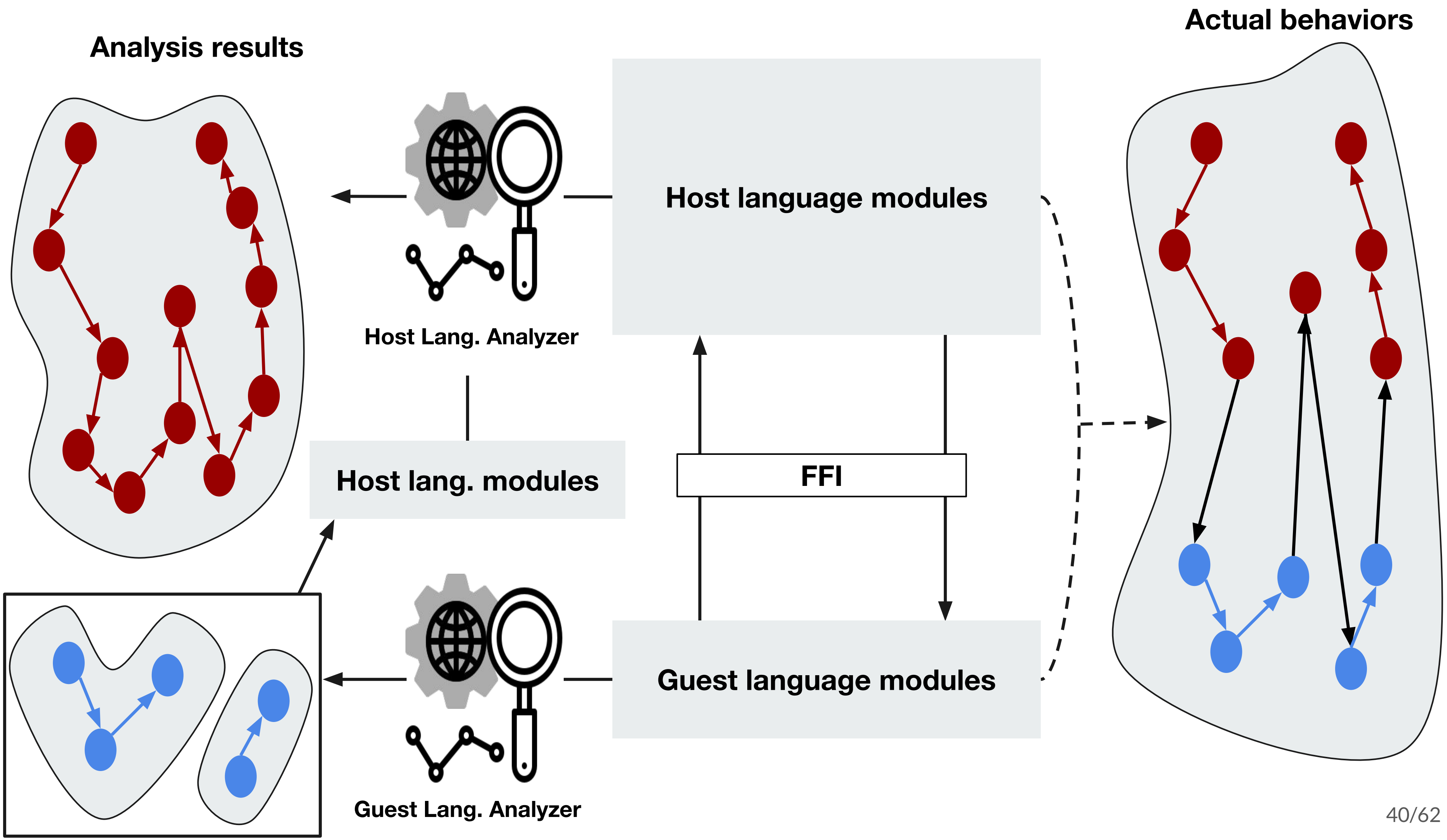
Disadvantage: absence of static checking



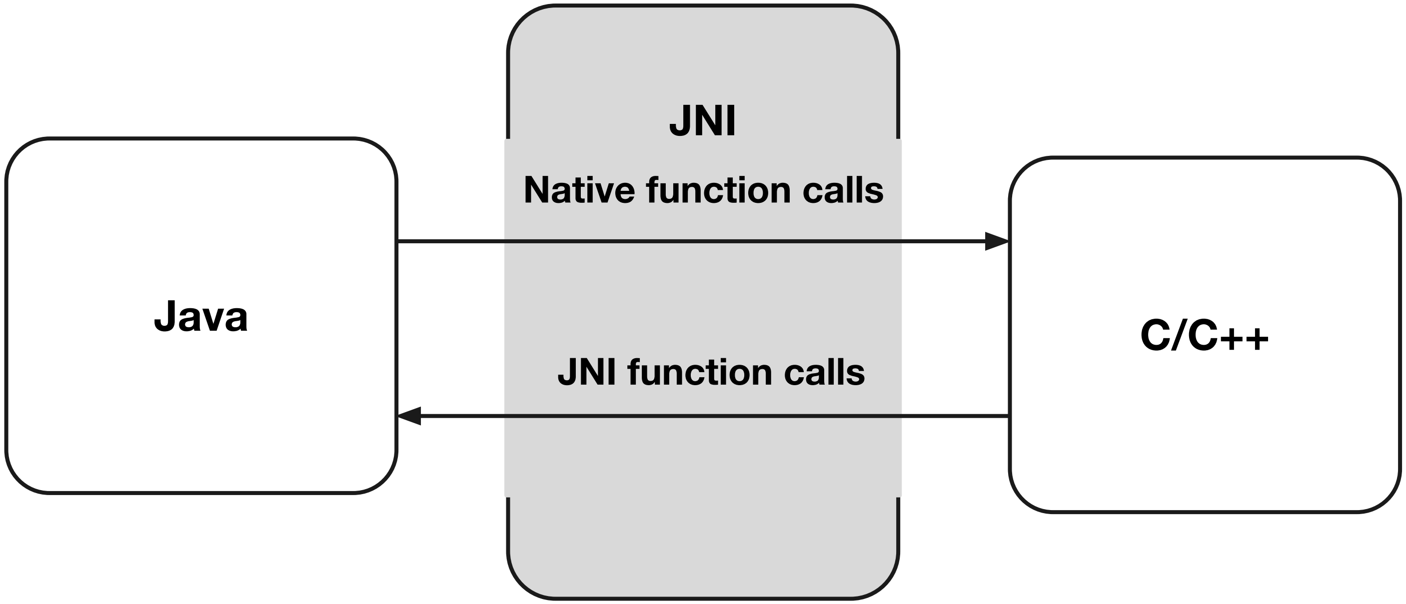
Limitation of static analyzers

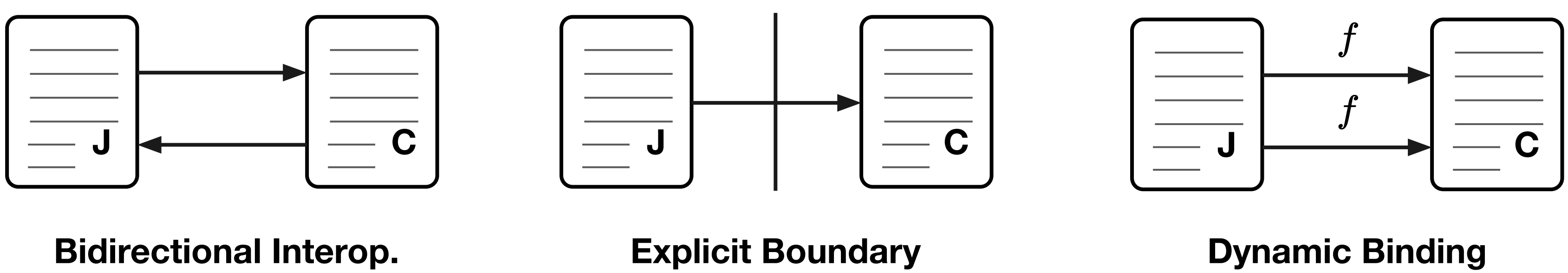


Our approach

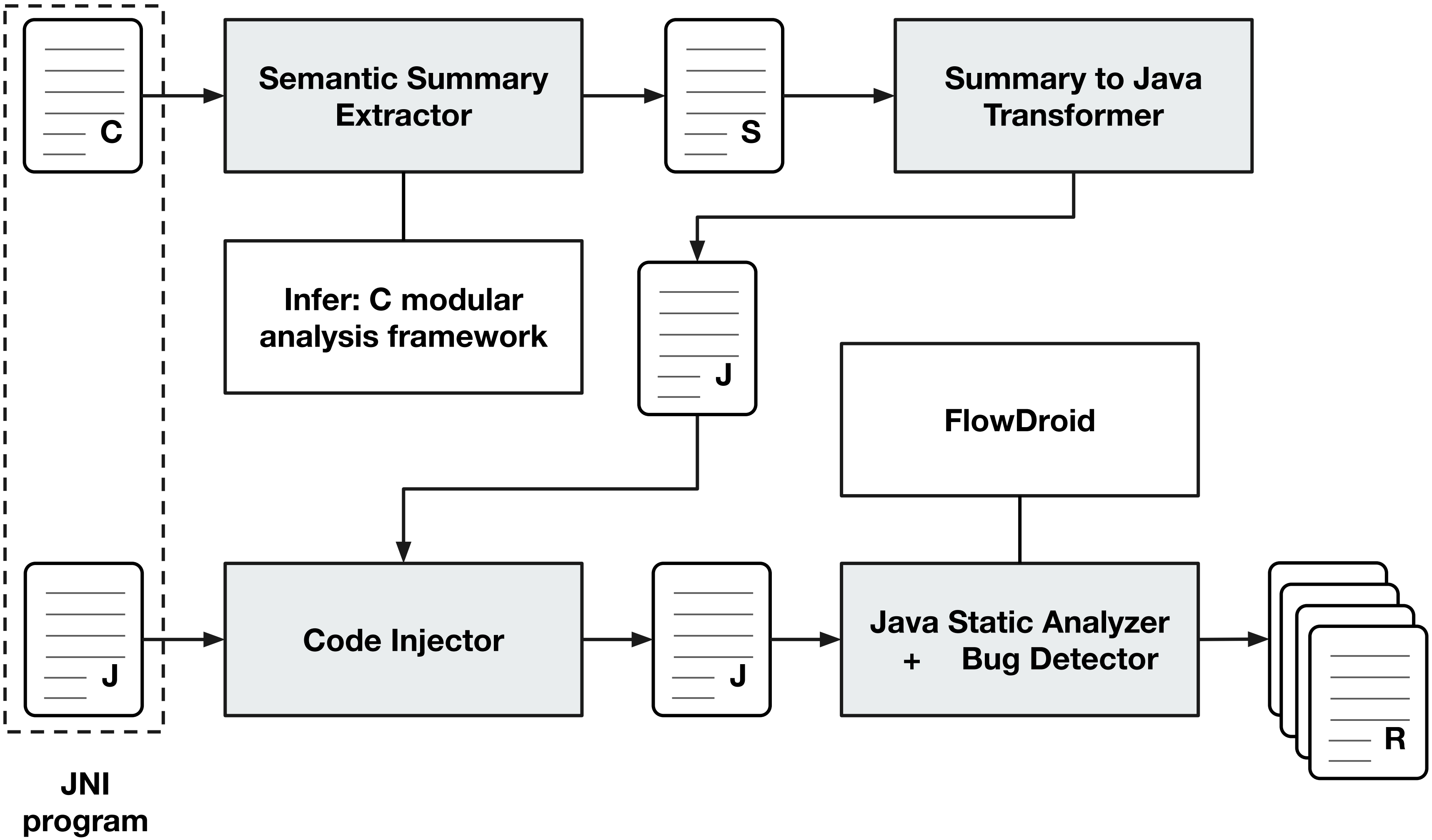


JNI Program: Java Native Interoperation





Overall structure of JNI program analysis



Example: analysis results of existing analysis

Java

```
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

native function call

native method

C

```
void Java_com_exmaple_App_callJava(JNIEnv* env, jobject /* this */, jobject app) {
  jclass klass = (*env)->GetObjectClass(env, app);
  jmethodID mid = (*env)->GetMethodID(env, klass, "foo", "()V");
  (*env)->CallVoidMethod(env, app, mid);
}
```

Example: analysis results of existing analysis

Java

```java
package com.example;

class CApp{
  static { System.loadLibrary(“lib”); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

native function call

native method

C

```c
void Java_com_exmaple_App_callJava(JNIEnv* env, jobject /* this */, jobject app) {
  jclass klass = (*env)->GetObjectClass(env, app);
  jmethodID mid = (*env)->GetMethodID(env, klass, “foo”, “()V”);
  (*env)->CallVoidMethod(env, app, mid);
}
```

JNI function calls

Example: analysis results of existing analysis

Java

```
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

native function call

native method

C

```
void Java_com_exmaple_App_callJava(JNIEnv* env, jobject /* this */, jobject app) {
  jclass klass = (*env)->GetObjectClass(env, app);
  jmethodID mid = (*env)->GetMethodID(env, klass, "foo", "()V");
  (*env)->CallVoidMethod(env, app, mid);
}
```

JNI function calls

Example: analysis results of existing analysis

Java

```
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}

void Java_com_exmaple_App_callJava(JNIEnv* env, jobject /* this */, jobject app) {
  jclass klass = (*env)->GetObjectClass(env, app);
  jmethodID mid = (*env)->GetMethodID(env, klass, "foo", "()V");
  (*env)->CallVoidMethod(env, app, mid);
}
```

native function call

Existing Java analysis

native method

exec

C

JNI function calls

Example: analysis results of existing analysis

Java

```java
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

C

```c
void Java_com_exmaple_App_callJava(JNIEnv* env, jobject /* this */, jobject app) {
  jclass klass = (*env)->GetObjectClass(env, app);
  jmethodID mid = (*env)->GetMethodID(env, klass, "foo", "()V");
  (*env)->CallVoidMethod(env, app, mid);
}
```

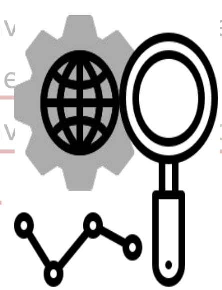
JNI function calls

Example: analysis results of existing analysis

Java

```java
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

```c
void Java_com_exmaple_App_callJava(JNIEnv                ect /* t
  jclass klass = (*env)->GetObjectClass(e
  jmethodID mid = (*env)->GetMethodID(env              oo", "()V");
  (*env)->CallVoidMethod(env, app, mid);
}
```



Guest Lang. Analyzer

JNI function calls

@1 GetObjectClass(v_1, v_2)	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	arg_3
@2 GetMethodID(v_1, v_2, v_3, v_4)	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	$ret_{@1}$
	v_3	$\mapsto$	"foo"
	v_4	$\mapsto$	"()V"
@3 CallVoidMethod(v_1, v_2, v_3)	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	arg_3
	v_3	$\mapsto$	$ret_{@2}$

Example: analysis results of existing analysis

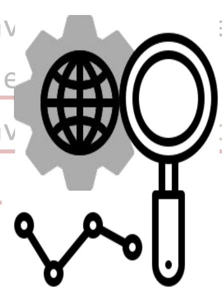
```
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

Java

```
void callJava(CApp app) {
  Class c = GetObjectClass(app);
  Method m = GetMethodID(app, "foo()V");
  CallVoidMethod(c, m);
}
```

@1 GetObjectClass(v_1, v_2)	$v_1 \mapsto arg_1$ $v_2 \mapsto arg_3$
@2 GetMethodID(v_1, v_2, v_3, v_4)	$v_1 \mapsto arg_1$ $v_2 \mapsto ret_{@1}$ $v_3 \mapsto$ "foo" $v_4 \mapsto$ "()V"
@3 CallVoidMethod(v_1, v_2, v_3)	$v_1 \mapsto arg_1$ $v_2 \mapsto arg_3$ $v_3 \mapsto ret_{@2}$

```
void Java_com_exmaple_App_callJava(JNIEnv          ect /* t
  jclass klass = (*env)->GetObjectClass(e
  jmethodID mid = (*env)->GetMethodID(env          oo", "()V");
  (*env)->CallVoidMethod(env, app, mid);
}
```



Guest Lang. Analyzer

JNI function calls

Example: analysis results of existing analysis

```java
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

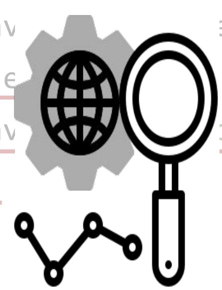
Java

```
void callJava(CApp app) {
  Class c = GetObjectClass(app);
  Method m = GetMethodID(app, "foo()V");
  CallVoidMethod(c, m);
}
```

@1 GetObjectClass(v_1, v_2)	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	arg_3
@2 GetMethodID(v_1, v_2, v_3, v_4)	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	$ret_{@1}$
	v_3	$\mapsto$	"foo"
	v_4	$\mapsto$	"()V"
@3 CallVoidMethod(v_1, v_2, v_3)	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	arg_3
	v_3	$\mapsto$	$ret_{@2}$

```
void Java_com_exmaple_App_callJava(JNIEnv ... ect /* t
  jclass klass = (*env)->GetObjectClass(e
  jmethodID mid = (*env)->GetMethodID(env ... oo", "()V");
  (*env)->CallVoidMethod(env, app, mid);
}
```

JNI function calls



Guest Lang. Analyzer

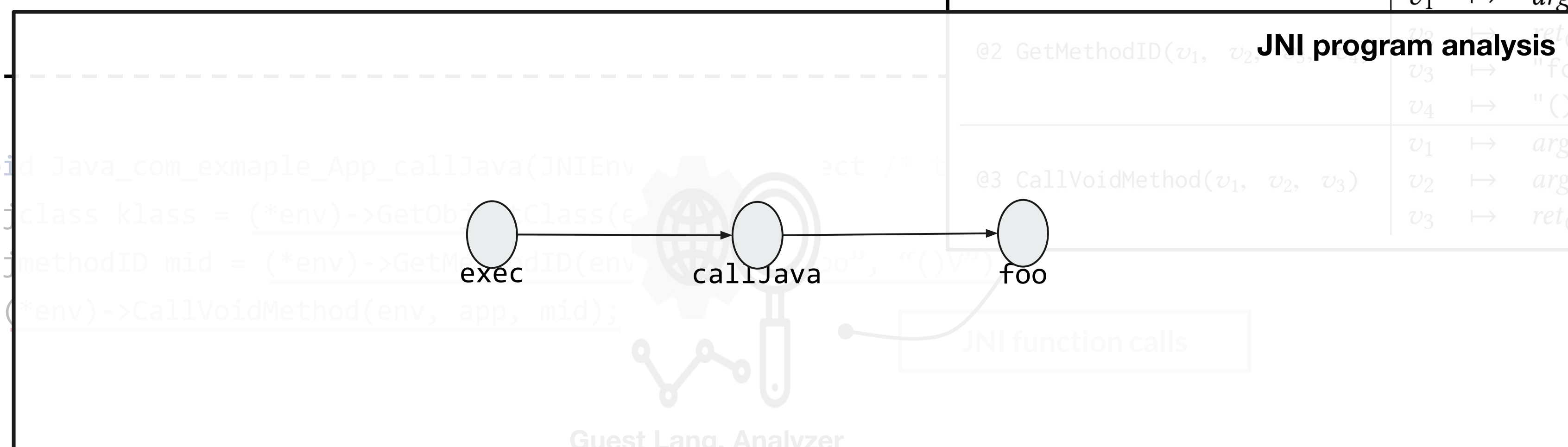
Example: analysis results of existing analysis

```
package com.example;

class CApp{
  static { System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo() { /* do something */ }
  void bar() { /* do something */ }
  native void callJava(CApp app);
}
```

Java

```
void callJava(CApp app) {
  Class c = GetObjectClass(app);
  Method m = GetMethodID(app, "foo()V");
  CallVoidMethod(c, m);
}
```

@1 GetObjectClass(v_1, v_2)	$v_1 \mapsto arg_1$ $v_2 \mapsto arg_3$
@2 GetMethodID(v_1, v_2, ...)	$v_1 \mapsto arg_1$ ...
@3 CallVoidMethod(v_1, v_2, v_3)	$v_1 \mapsto$ ...



Evaluation: call graph construction

Name	#LoC$_C$	Call$_{C\to J}$ #Precise	Call$_{C\to J}$ #Resolved	Call$_{C\to J}$ Total	GetField$_{C\to J}$ #Precise	GetField$_{C\to J}$ #Resolved	GetField$_{C\to J}$ Total	SetField$_{C\to J}$ #Precise	SetField$_{C\to J}$ #Resolved	SetField$_{C\to J}$ Total	Time (sec.) C	Time (sec.) Java
Graph 89	449027	1	1	1	0	0	0	0	0	0	2149.27	5.26
APV PDF Viewer	312429	3	3	7	4	4	4	4	4	4	1620.19	3.85
Lumicall	277763	27	27	27	15	15	31	4	4	4	121.19	28.22
Timidity AE	214052	3	3	3	0	0	0	0	0	0	119.21	5.08
Plumble	150190	1	1	2	20	20	52	2	2	6	84.45	19.78
CommonsLab	122508	10	10	10	0	0	0	0	0	0	58.17	9.59
CrossWords	72786	81	108	131	15	95	119	19	81	106	1553.19	15.14
Sipdroid	70288	0	0	0	49	49	69	4	4	4	66.08	16.21
Xmp Mod Player	69157	0	0	0	0	0	0	2	2	2	51.92	3.79
DroidZebra	38084	126	126	184	0	0	0	0	0	0	514.16	6.90
Fwknop2	16458	0	0	0	13	13	13	0	0	0	50.46	6.41
Taps of Fire	11357	0	0	0	0	0	9	0	0	4	92.72	4.23
agram	1550	0	0	0	0	0	0	3	3	3	3.76	3.39
VotAR	869	7	7	7	2	2	2	1	1	1	2.21	3.49
Total		259	286	372	118	198	299	39	101	134		

For **real-world 50 JNI apps** on F-Droid,

- Resolved **585 / 805 (73%)** foreign function calls from C to Java
    - CallMethod: 286 / 372 (77%), GetField: 198 / 299 (66%), SetField: 101 / 134 (75%)
    - 417 out of 585 (71%) resolved foreign function calls are precise
- Analyzed over **400,000 lines of C code** in about 35 minutes

Evaluation: interoperation bug detection

Name	Wrong FF Call(#)	Exception Mishandling(#)
Graph 89	*WrongDesc* (1) *TypeMismatching* (3)	-
APV PDF Viewer	*MissingFun* (2) *TypeMismatching* (2)	-
Lumicall	*MissingFun* (1)	*UnsafeSubsequentCall* (23)
Sipdroid	*MissingFun* (1)	*UnsafeSubsequentCall* (25)
VotAR	*WrongDesc* (1)	-
Taps of Fire	*WrongDesc* (1)	-
Xmp Mod Player	*WrongDesc* (3)	-
CrossWords	*MissingFun* (3)	-
DroidZebra	-	*MissingHandling* (4)
NetGuard	-	*InappositeHandling* (4)

For **real-world 50 JNI apps** on F-Droid,

- Found **74 interoperation bugs in 10 apps**
  - 18 wrong foreing function call bugs in 8 apps
  - 56 exception mishandling bugs in 4 apps

Case: wrong foreing function call (1)

```
// Java
native Channel inheritedChannelImpl

// C
/*
jobject Java_org_sipdroid_net_impl_OSNetworkSystem_inheritedChannelImpl
*/
```

Missing C function

Case: wrong foreing function call (2)

```
// Java
synchronized private native int parseFile

// C
void Java_cx_hell_android_lib_pdf_PDF_parseFile
```

Declared Type Mismatching

Case: wrong foreing function call (3)

```
// C
jmethodID method = (*DbusJNIenv)->GetStaticMethodID(DbugJNIenv, class,
                        “ReceiveFile”, “(Ljava/lang/String;Ljava/lang/String;)V”);
...
(*DbusJNIenv)->CallStaticIntMethod(DbusJNIenv, class, method, jSrc, jDst);
```

Wrong Descriptor

Exception mishandling?

There are two ways to handle an exception in native code:

- The native method can choose to return immediately, causing the exception to be thrown in the Java code that initiated the native method call.
- The native code can clear the exception by calling `ExceptionClear()`, and then execute its own exception-handling code.

After an exception has been raised, the native code must first clear the exception before making other JNI calls.

source: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html#wp9502

Case: missing exception handling (1)

```
// C
int jniThrowException (...) {
  jclass ec = env->FindClass(className); // Unsafe JNI function call
  …
}

void oneTimeInitializationImpl(...) {
  jmethodID m = env->GetStaticMethodID(...);
  if (m == NULL) jniThrowException(...); // Exception occured
  …
}
```

Unsafe subsequent JNI function call

Case: missing exception handling (2)

```
// Java
public void run() { droidzebra_json_get_int(0, null); }

// C
jint droidzebra_json_get_int(jobject json) {
  jclass cls = env->GetObjectClass(json);
  …
  value = env->CallIntMethod(json, mid, ...);
  if ( env->ExceptionCheck() ) return -1; // Exception is checked, but not cleared
  return value ;
}
```

Missing exception handling

Case: missing exception handling (3)

```
// C
jobject jniNewObject(...) {
  jobject obj = env->NewObject(...);
  if ( object == NULL ) log_android(...);
  else jniCheckException(env); // Check exceptions only when no exception occurs
  return object ;
}
```

Inapposite exception handling

Composing Static Analyzers for

Bug and Security Vulnerability Detection

in **Multilingual Android apps**

WALA$_{Java}$ & WALA$_{JavaScript}$ and **FlowDroid & Infer**

Composing Static Analyzers for

Bug and Security Vulnerability Detection

in **Multilingual Android apps**

Hybrid apps and **JNI apps**