Static Analysis of JNI Programs via Binary Decompilation

소프트웨어재난연구센터 2023 여름정기워크숍 주요연구발표

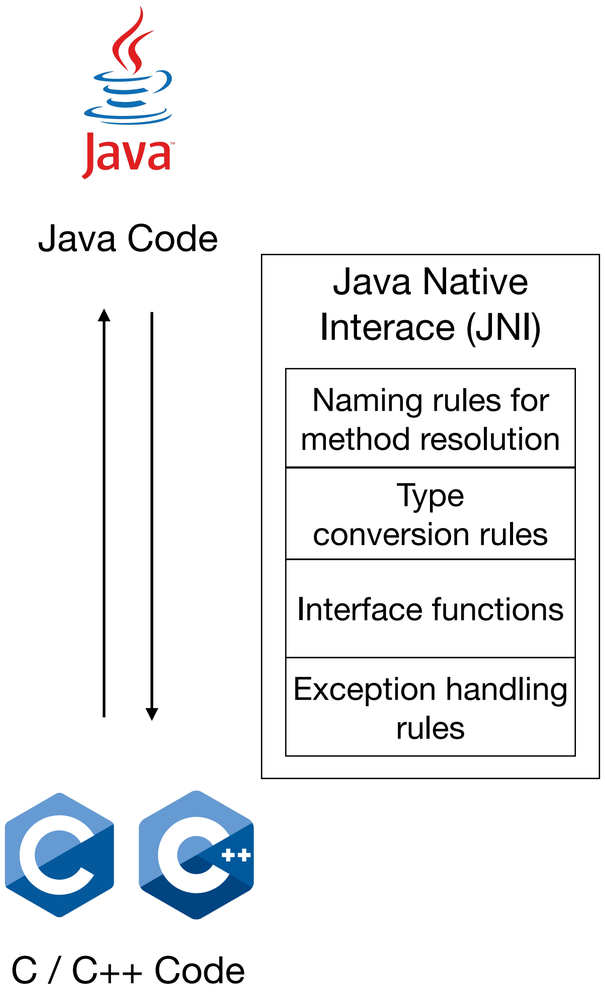
박지희†*, 이성호‡*, 홍재민†, 류석영†

†: KAIST, ‡: 충남대학교, *: 공동 1저자

2023-07-06

PLRG
Programming Language Research Group

Java Native Interface




```java
package pack;
public class Example {
  static { System.load("lib.so"); }
  int bar(int x) { /* ... */ }
  native int foo();
}
```

(a) Java code

```c
jint Java_pack_Example_foo
        (JNIEnv *env, jobject thiz) {
  jclass cls = (*env)->GetObjectClass(env, thiz);
  jmethodID jmid =
    (*env)->GetMethodID(env, cls, "bar", "(I)I");
  jint result =
    (*env)->CallIntMethod(env, thiz, jmid, 3);
  return result;
}
```

(b) C source code of `lib.so`

Bug in Native JNI Application

- JNI exception handling error
- Data leakage using JNI data-flow
- Using outdated(vulnerable) libraries

"Google Play의 100,000개 앱 중
39.7%의 앱이 native code를 사용"

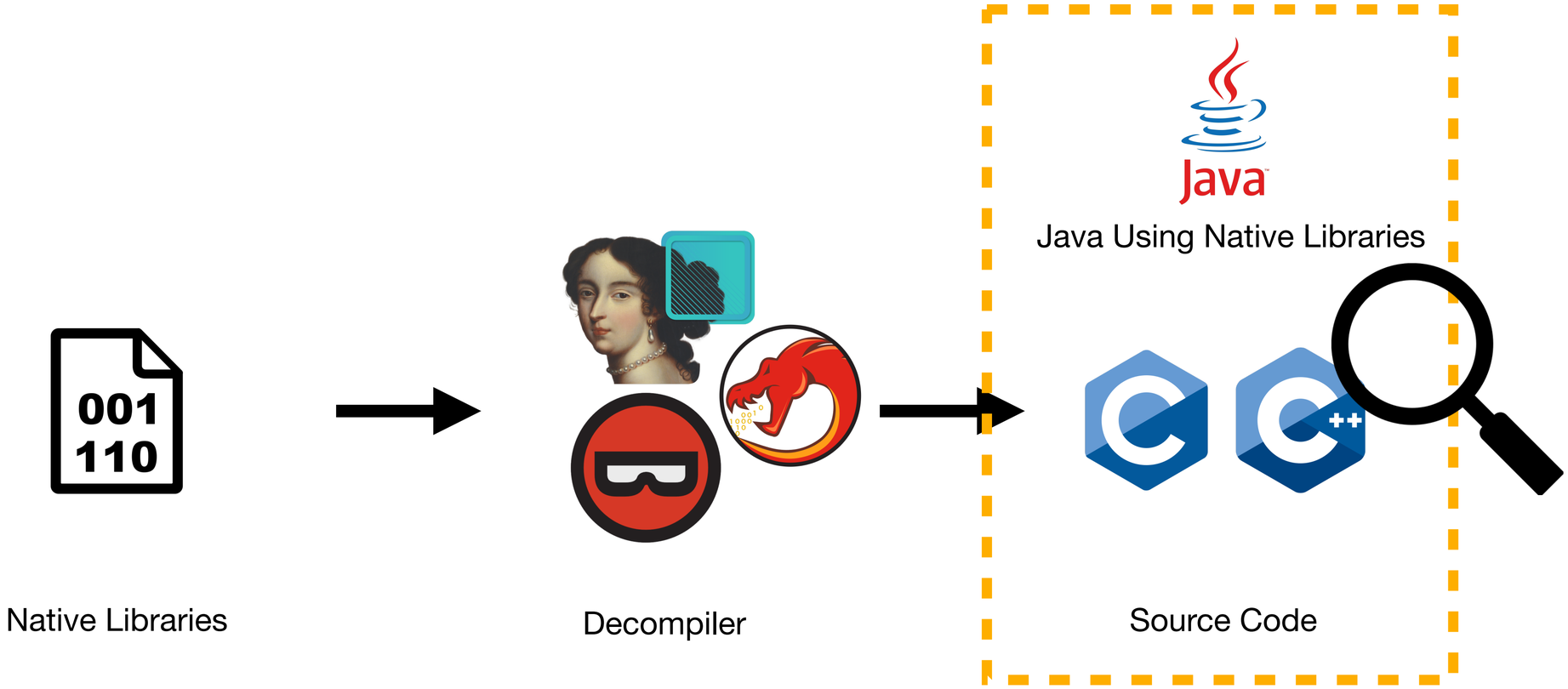
JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code (CCS'18)

Static Analysis of Native JNI Program

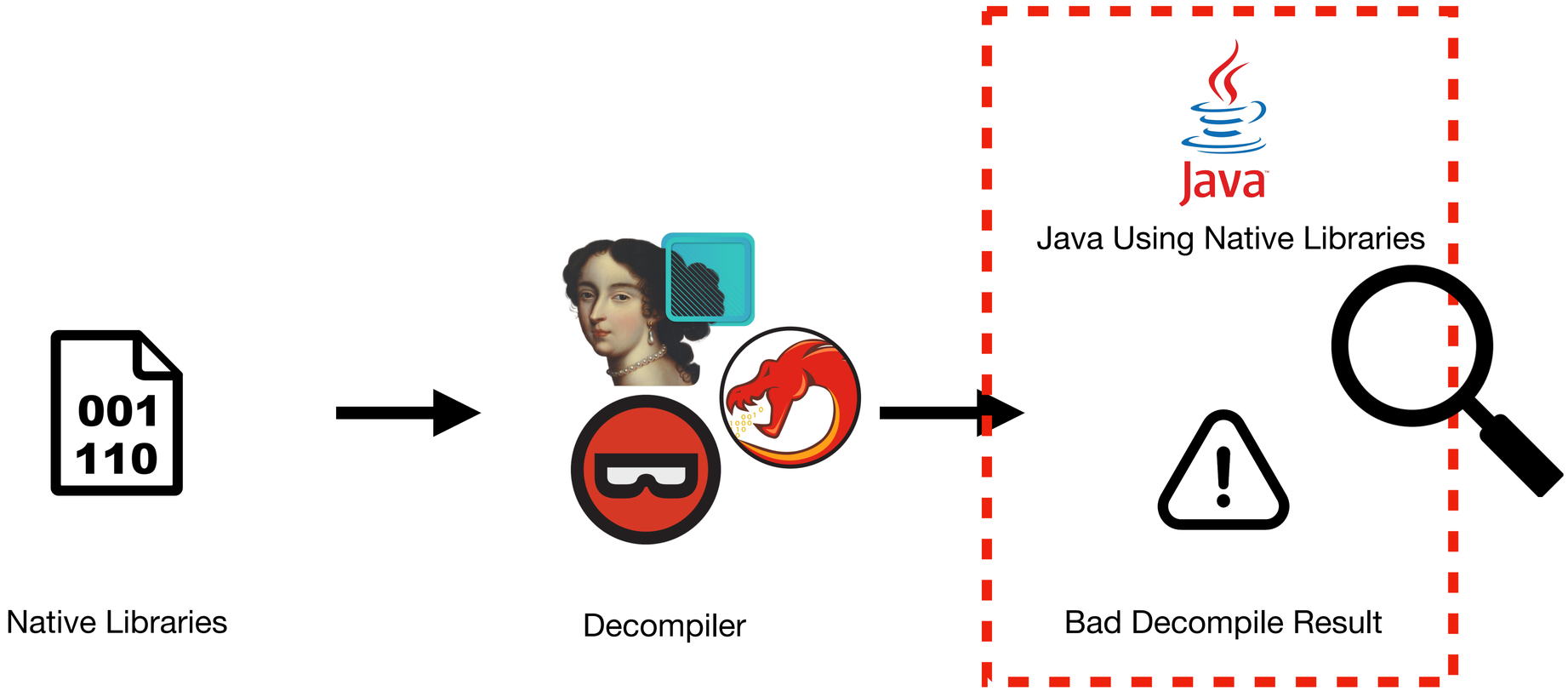
- Most works are syntactic analysis
  - Pattern matching, search strings

- JN-SAF: Information flow analysis using symbolic execution
  - Path explosion issue

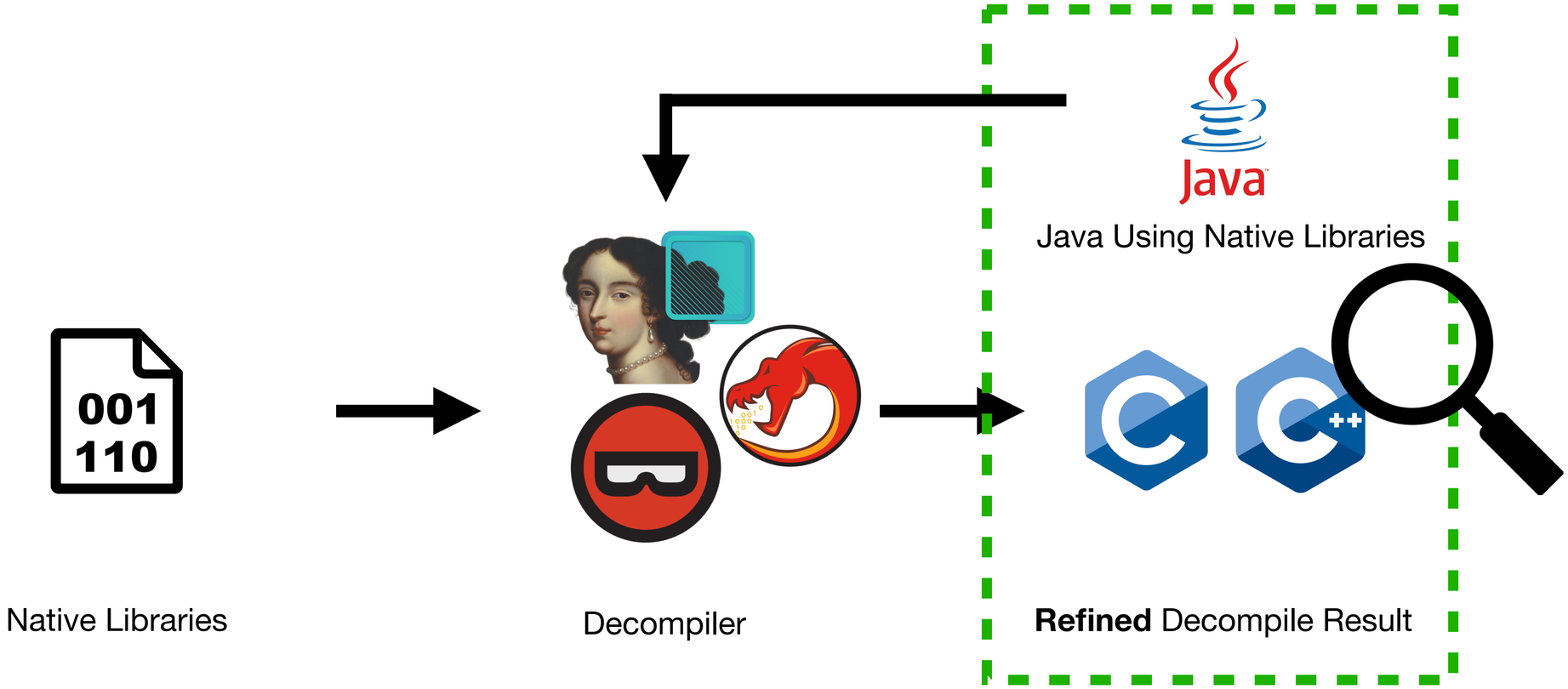
Idea: Use Decompiler to Analyze Binary?



Idea: Use Decompiler to Analyze Binary?



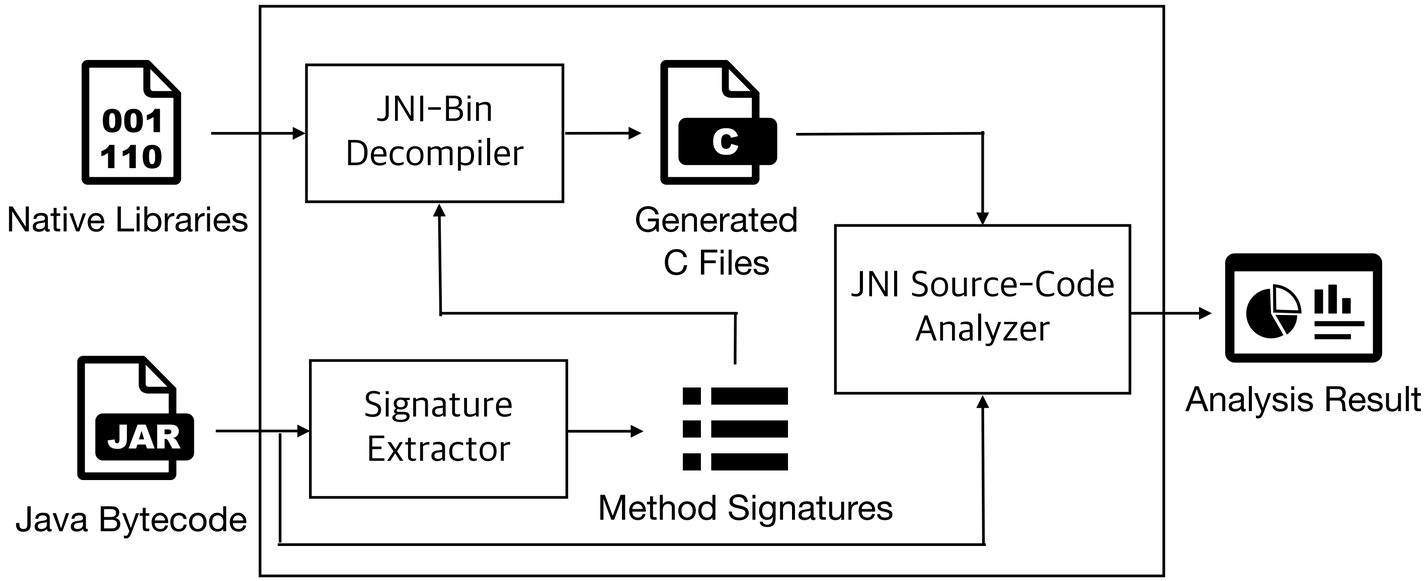
New Idea: Use Java Information to Refine Decompilation



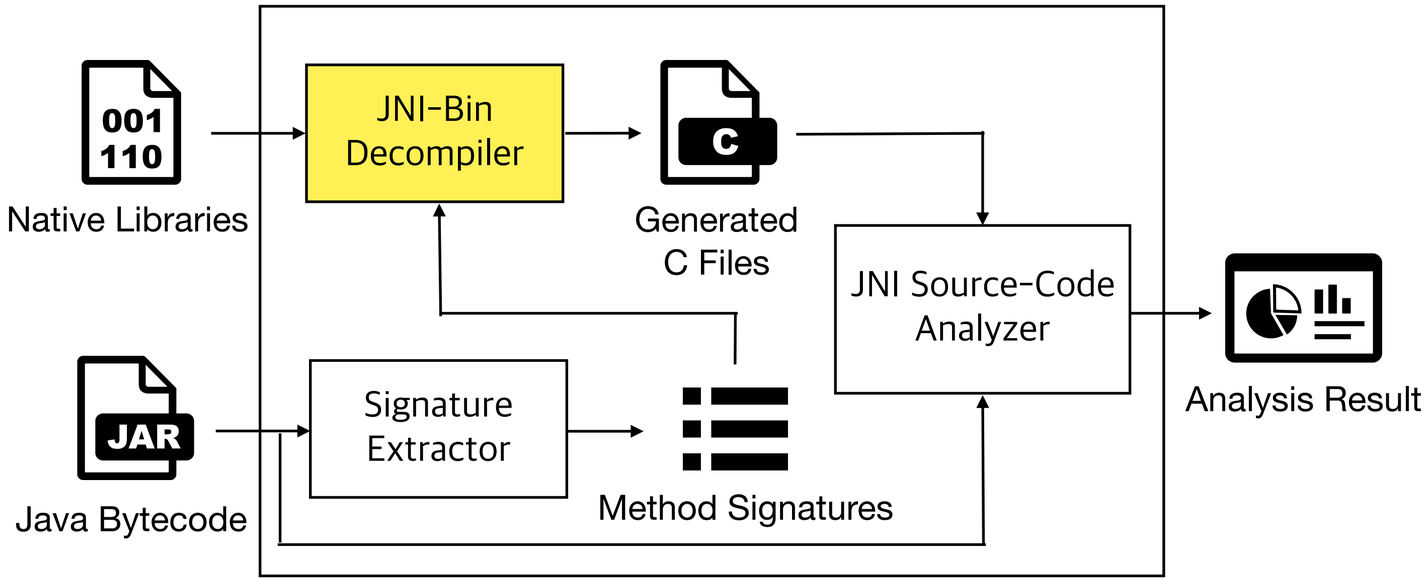
Contributions

- Propose to analyze native JNI programs using binary decompilation
- Identify challenges of decompiling binaries into JNI C code and suggest **specialized solutions using Java analysis result**

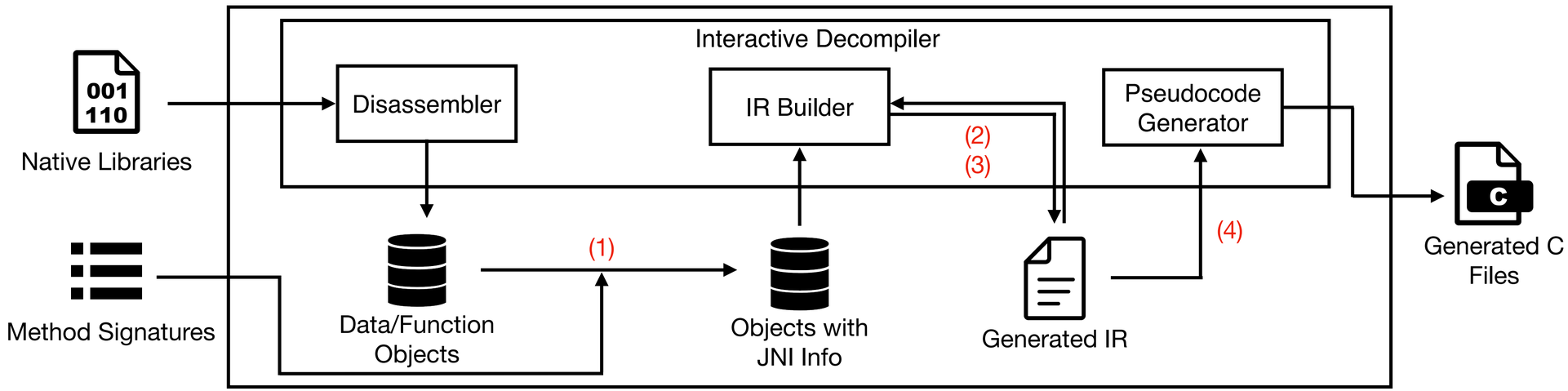
Overview of Approach



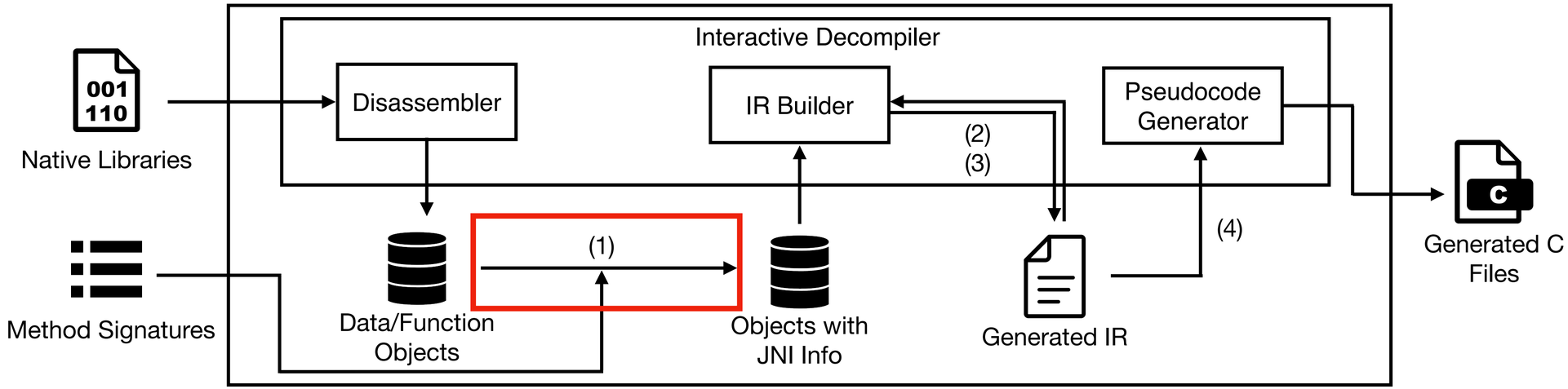
Overview of Approach



JNI-Bin Decompiler



1. Resolve Function Signature from Java



1. Resolve Function Signature from Java

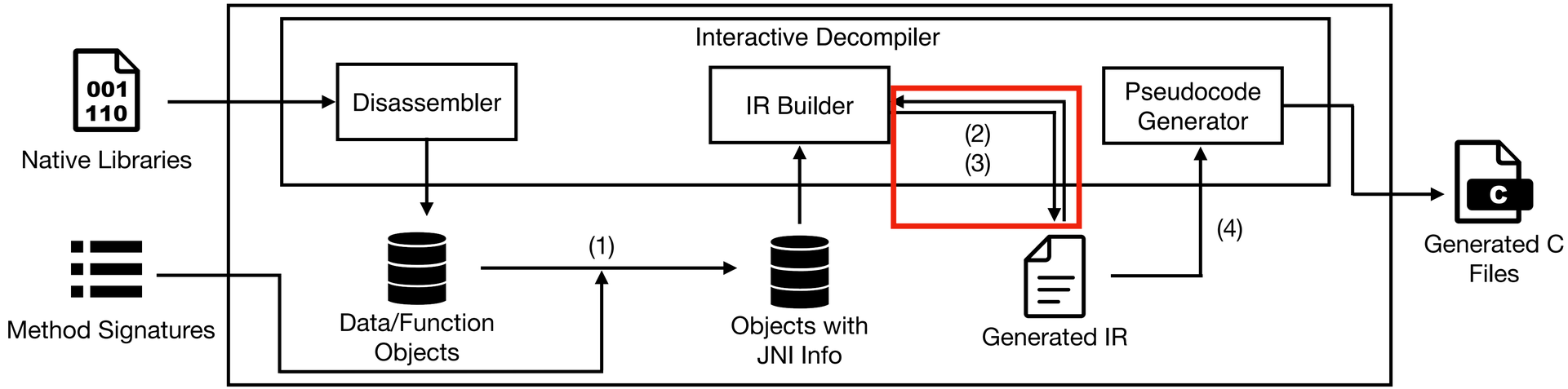
- Extract signature from Java and rewrite native function signature

```
static native String get_column_name(long a3, int a4)
```

```
int Java_get_1column_1name(
  int a1, int a2, int a3, int a4, int a5) {
   int result;
    if (sub_F1AC(a3, a5))
      result = (*(int (**)(int, int))
                (*(_DWORD *)a1 + 668))(a1);
    else
      result = 0;
    return result;
}
```

```
jstring Java_get_1column_1name(
  JNIEnv* a1, jclass a2, jlong a3, jint a4) {
   jstring result;
    if (sub_F1AC(a3, a4))
      result = (jstring)
        (*(int (*)(JNIEnv *))
        (*a1)->NewStringUTF)(a1);
    else
      result = 0;
    return result;
}
```

2. Type Propagation for JNI-related Type



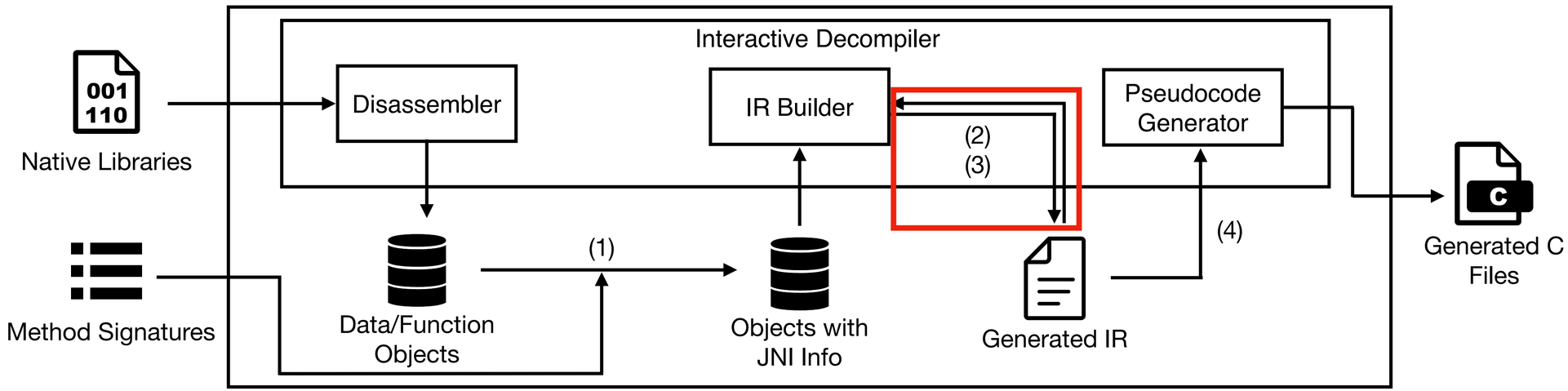
2. Type Propagation for JNI-related Type

- Force to propagate JNI-related type when decompiling

```
int Java_createFrame(JNIEnv* a1, ...) {
  ...
  jbitmap = (jobject)create_graphics(
    (int)a1, v69[0], v69[1]);
  ...
 }

int create_graphics(int a1, int a2, int a3) {
  ...
  int v9;
  v9 = (*(int (**)(int, const char*))
  (*(int *)a1 + 24)(
    a1, "android/graphics/Bitmap"));
}
```

```
int Java_createFrame(JNIEnv* a1, ...) {
  ...
  jbitmap = (jobject)create_graphics(
    a1, v69[0], v69[1]);
  ...
 }

int create_graphics(JNIEnv* a1, int a2, int a3) {
  ...
  jclass v9;
  v9 = (*a1)->FindClass(
        a1, "android/graphics/Bitmap"));
}
```

3. Resolve JNI Function calls



3-1. JNI Function with Fixed Arity Signature

- Rewrite fixed-arity JNI function call expression

```
jstring Java_get_1column_1name(
  JNIEnv* a1, jclass a2, jlong a3, jint a4) {
   jstring result;
    if (sub_F1AC(a3, a4))
      result = (jstring)
        (*(int (*)(JNIEnv *))(*a1)->NewStringUTF)(a1);
    else
      result = 0;
    return result;
}
```

```
jstring Java_get_1column_1name(
  JNIEnv* a1, jclass a2, jlong a3, jint a4) {
   char* v5;
   jstring result;
   v5 = (char*)sub_F1AC(a3, a4);
    if (v5)
      result =(*a1)->NewStringUTF(a1, v5);
    else
      result = 0;
    return result;
}
```

3-2. JNI function with Variadic Signature

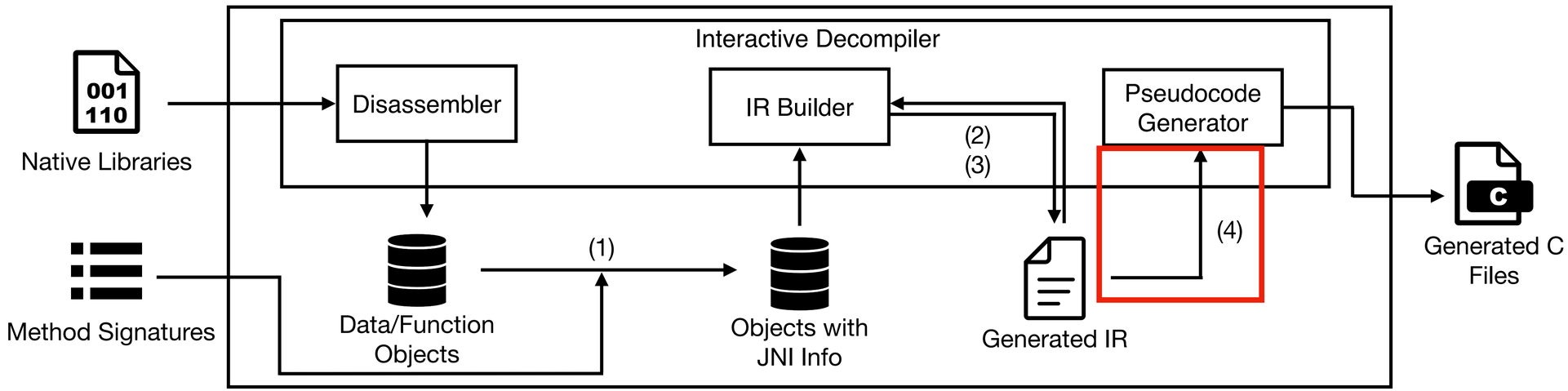
- Use Intra-procedural string analysis to find methodID
- From the analysis result and related Java code, rewrite variadic JNI function calls

```
jstring Java_prepare(JNIEnv* a1, jobject a2,
  jobject a3, jobject a4, jint a5) {

  jint v10;
  jmethodID v11;
  jclass v14;
  jstring v17;
  ...
  v14 = (*a1)->GetObjectClass(a1, a3);
  if ( v14 ) {
    v11 = (*a1)->GetMethodID(a1,
      v14, "set_param", "(Ljava/lang/String;)I");
    ...
    v10 = (*a1)->CallIntMethod(a1, a3, v11, v17, a4);
    ...
  }
}
```

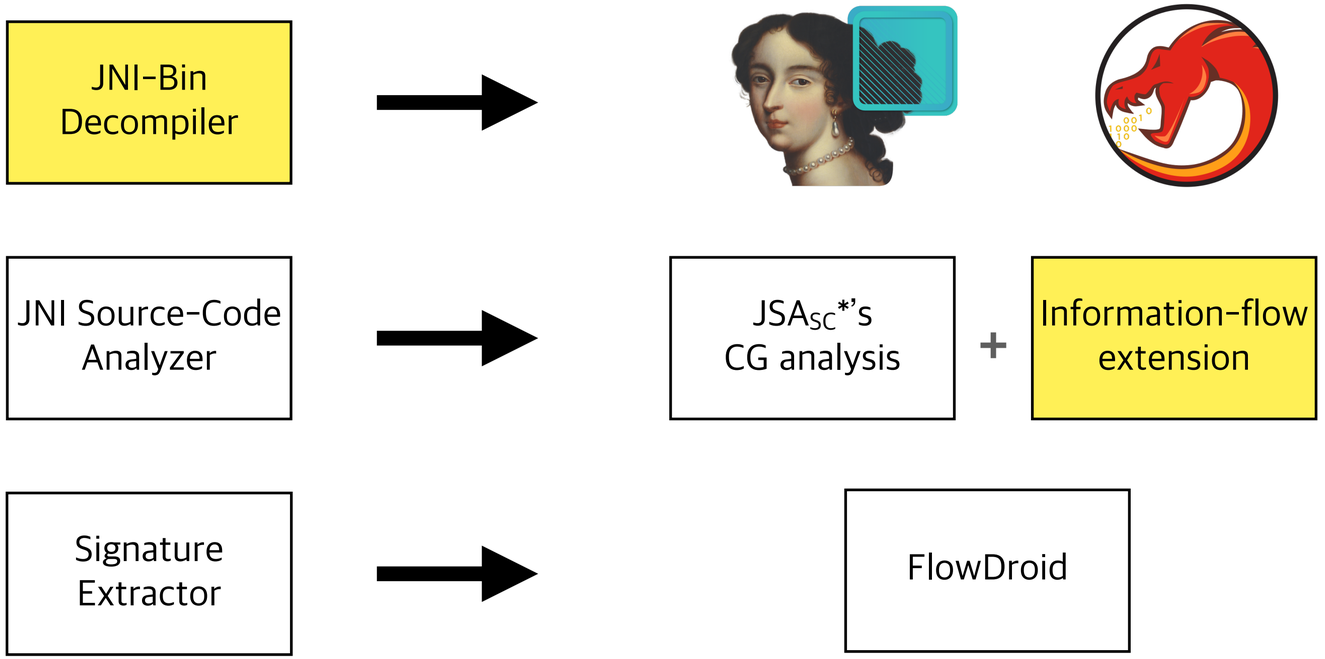
```
jstring Java_prepare(JNIEnv* a1, jobject a2,
  jobject a3, jobject a4, jint a5) {

  jint v10;
  jmethodID v11;
  jclass v14;
  jstring v17;
  ...
  v14 = (*a1)->GetObjectClass(a1, a3);
  if ( v14 ) {
    v11 = (*a1)->GetMethodID(a1,
      v14, "set_param", "(Ljava/lang/String;)I");
    ...
    v10 = (*a1)->CallIntMethod(a1, a3, v11, v17);
   ...
  }
}
```

PLRG Programming Language Research Group

4. Generating Compilable Code from Pseudocode



Implementation



* S. Lee, H. Lee and S. Ryu, "Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis," ASE 2020

Evaluation Target

- **NativeFlowBench** - 23 challenging Android-JNI apps (use 16 out of 23)
  - Small CLoC(8-42), targeting complex JNI interoperation

- **F-Droid** - 10 real-world open-source Android-JNI apps
  - Relatively large CLoC(42-13226)

Call Graph Analysis Result

- 100% Resolved for Benchmark

RESULTS OF ANALYZING BENCHMARKS

Benchmark	JLoC	CLoC	$Call_{J \to C}$			$Call_{C \to J}$			$Field_{C \to J}$		
			$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$	JSA_{SC}	$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$	JSA_{SC}	$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$	JSA_{SC}
native_complexdata	90	35	2	2	2	2	2	2	0	0	0
native_complexdata_stringop	88	29	1	1	1	0	0	0	0	0	0
native_heap_modify	63	26	1	1	1	2	2	2	2	2	2
native_leak	61	17	1	1	1	0	0	0	0	0	0
native_leak_array	63	21	1	1	1	0	0	0	0	0	0
native_method_overloading	63	32	1	1	1	0	0	0	0	0	0
native_multiple_interactions	73	37	2	2	2	1	1	1	1	1	1
native_multiple_libraries	63	35	1	1	1	0	0	0	0	0	0
native_noleak	62	13	1	1	1	0	0	0	0	0	0
native_noleak_array	63	21	1	1	1	0	0	0	0	0	0
native_nosource	41	8	1	1	1	0	0	0	0	0	0
native_set_field_from_arg	109	22	1	1	1	0	0	0	2	2	2
native_set_field_from_arg_field	113	23	1	1	1	0	0	0	3	3	3
native_set_field_from_native	100	42	1	1	1	3	3	3	5	5	5
native_source	58	19	1	1	1	2	2	2	1	1	1
native_source_clean	89	19	1	1	1	0	0	0	1	1	1
Total			18	18	18	10	10	10	15	15	15

PLRG Programming Language Research Group

Call Graph Analysis Result

- 90%(Ghidra)-100%(IDA-Pro) Resolved for F-Droid

RESULTS OF ANALYZING REAL-WORLD OPEN-SOURCE JNI APPS

Application	JLoC	CLoC	Summary (#)			$Call_{J \to C}$			$Field_{C \to J}$		
			$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$	JSA_{SC}	$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$	JSA_{SC}	$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$	JSA_{SC}
AsciiCam	2272	120	3	3	3	0	0	0	0	0	0
PracticeHub	1058	348	1	0	1	1	0	1	0	0	0
simpleRT	97	493	3	3	3	3	3	3	0	0	0
SpiritF	6479	13226	2	2	2	1	1	1	0	0	0
AndroSS	1681	334	2	2	2	4	4	4	0	0	0
Overchan	52051	1721	18	15	18	0	0	0	0	0	0
Fwknop2	2220	8418	1	1	1	1	1	1	13	13	13
Compass	1683	42	3	3	3	0	0	0	0	0	0
AndIodine	1178	7972	9	9	9	5	5	5	0	0	0
Obsqr	1070	8673	1	1	1	0	0	0	0	0	0
Total			43	39	43	15	14	15	13	13	13

Information Flow Analysis Result

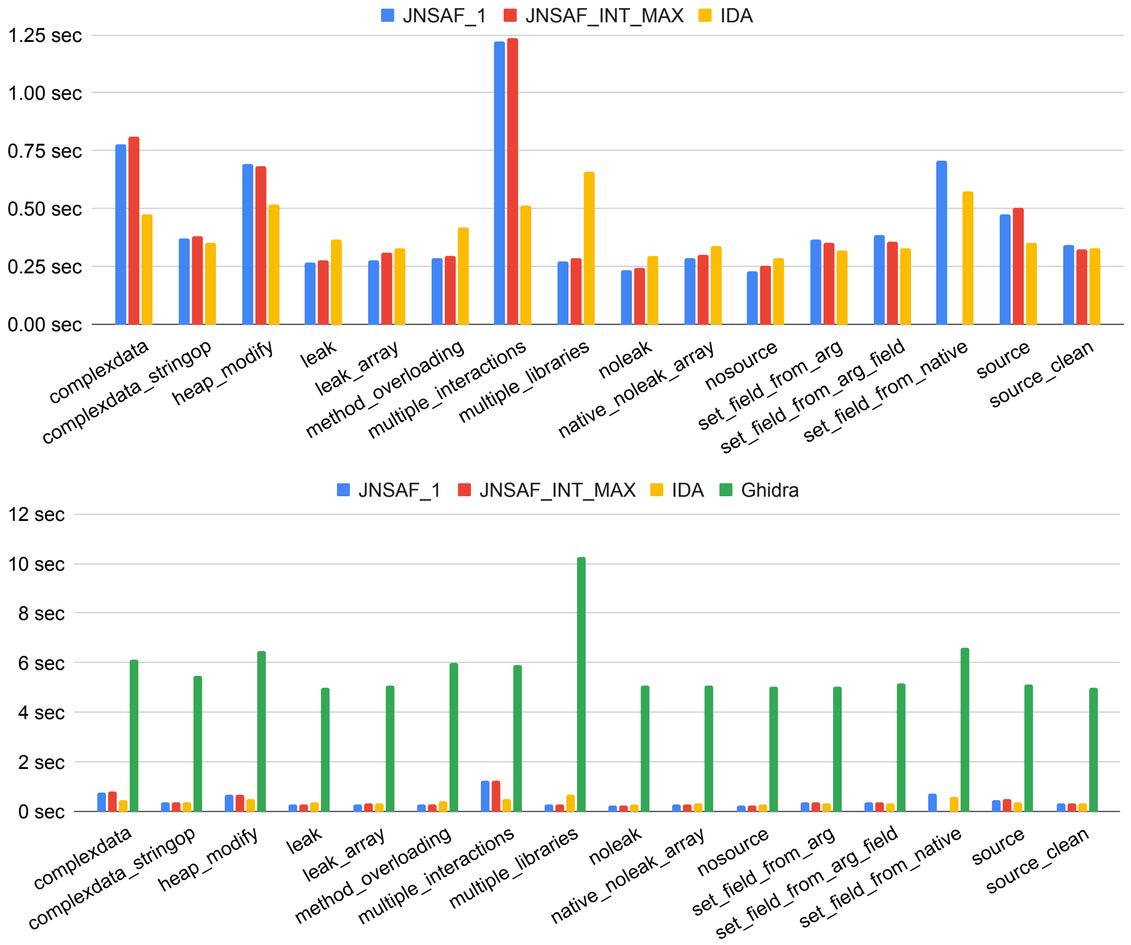
RESULTS OF DETECTING DATA LEAKEAGES IN BENCHMARKS

Benchmark	Data leakage		
	$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$	JN-SAF
native_complexdata	◯	◯	◯
native_complexdata_stringop			
native_heap_modify	◯	◯	×
native_leak	◯	◯	◯
native_leak_array	◯	◯	◯
native_method_overloading	◯	◯	◯
native_multiple_interactions	◯	◯	◯
native_multiple_libraries	◯	◯	◯
native_noleak			
native_noleak_array	⊗	⊗	⊗
native_nosource			
native_set_field_from_arg	◯◯	◯◯	◯×
native_set_field_from_arg_field	◯◯	◯◯	◯×
native_set_field_from_native	◯◯	◯◯	TIMEOUT
native_source	◯	◯	×
native_source_clean	⊗	⊗	
Total	16	16	9

◯ = True positive ⊗ = False positive ×= False negative

- Precision: 87.5% (Ours) vs 88% (JN-SAF)
- Recall: 100% (Ours) vs 64%(JN-SAF)
- 1 Timeout for JN-SAF

Information Flow Analysis Execution Time



ARITHMETIC MEAN OF JNI INTEROPERATION BEHAVIOR EXTRACTION TIME (EXCLUDING NATIVE_SET_FIELD_FROM_NATIVE)

	JN-SAF$_{1}$	JN-SAF$_{\texttt{INT_MAX}}$	JSA$_{\text{DEC-IDA}}$	JSA$_{\text{DEC-Ghidra}}$
Time (s)	0.432	0.441	0.392	5.718

PLRG Programming Language Research Group

Challenging Code Patterns for JN-SAF

```java
// MultiplePaths.java
class MainActivity {
 native void send(int secret);
}
```

```c
// C
jint filter(jint secret) {
 jint ret = 0;
 if (test1)
  ret = 1;
 ... // two more branches
 if (test4)
  ret = secret;
 return ret;
}
void send (
 JNIEnv* env, jobject thiz,
    jint secret) {
 sink(filter(secret)); // sink
}
```

(a) Multiple execution paths

```java
// GlobalJNI.java
class MainActivity {
 native void send(int secret);
 void sink(int secret) {
  ... // leak the data
 }
}
```

```c
// C
jmethodID mid;
void init(
 JNIEnv* env, jobject thiz) {
 jclass cls =
  (*env)->GetObjectClass(env, thiz);
 mid = (*env)->GetMethodID(
    env, cls, "sink", "(I)V");
}
void send(
 JNIEnv* env, jobject thiz, jint secret) {
 (*env)->CallVoidMethod(env, thiz, mid,
    secret);
}
```

(b) Caching data in global variables

```java
// DynamicDispatch.java
class MainActivity {
 native void send(Sender sender, int secret);
}
class Sender {
 void maySink(int secret) {
  ... // do not leak the data
 }
}
class ChildSender extends Sender {
 @Override
 void maySink(int secret) {
  ... // leak the data
 }
}
```

```c
// C
void send(
 JNIEnv* env, jobject thiz, jobject sender,
    jint secret) {
 jclass cls =
  (*env)->GetObjectClass(env, sender);
 jmethodID mid = (*env)->GetMethodID(
    env, cls, "maySink", "(I)V");
 (*env)->CallVoidMethod(env, sender, mid,
    secret);
}
```

(c) Dynamic dispatch