디컴파일러를 이용해 소스 코드가 없는 JNI 프로그램 정적 분석하기

Static Analysis of JNI Programs via Binary Decompilation

Published on TSE Vol. 49, No. 5, 2023.

박지희, 이성호, 홍재민, 류석영

2024-07-08

PLRG
Programming Language Research Group

자바 네이티브 인터페이스(JNI)

```
1  package pack;
2  public class Example {
3    static { System.load("lib.so"); }
4    int bar(int x) { /* ... */ }
5    native int foo();
6  }
```

bar(3) 호출

```
7  jint Java_pack_Example_foo
8          (JNIEnv *env, jobject thiz) {
9    jclass cls = (*env)->GetObjectClass(env, thiz);
10   jmethodID jmid =
11     (*env)->GetMethodID(env, cls, "bar", "(I)I");
12   jint result =
13     (*env)->CallIntMethod(env, thiz, jmid, 3);
14   return result;
15 }
```

Java 코드

C / C++ 코드

자바 네이티브 인터페이스 (JNI)

- 메소드 이름 규칙
- 타입 변환 규칙
- 인터페이스 함수
- 예외 처리 규칙

PLRG Programming Language Research Group

소스 코드가 없는 JNI 프로그램의 정적 분석

대부분 구분 분석 (syntactic analysis) 기반

- 패턴 매치, 문자열 검색 등

JN-SAF: 기호실행을 이용한 바이너리 JNI 프로그램의 데이터 흐름 분석

- 경로 폭발 문제 때문에 효과적으로 공간을 탐색하지 못함

동기: 디컴파일러를 이용한 프로그램 분석

디컴파일러: 바이너리 코드를 C 코드로 바꿔주는 툴

기존의 빠르고 효율적인 소스 기반 분석기를 사용 가능

그러나, 분석에 사용할 만한 코드 품질이 나오지 않음

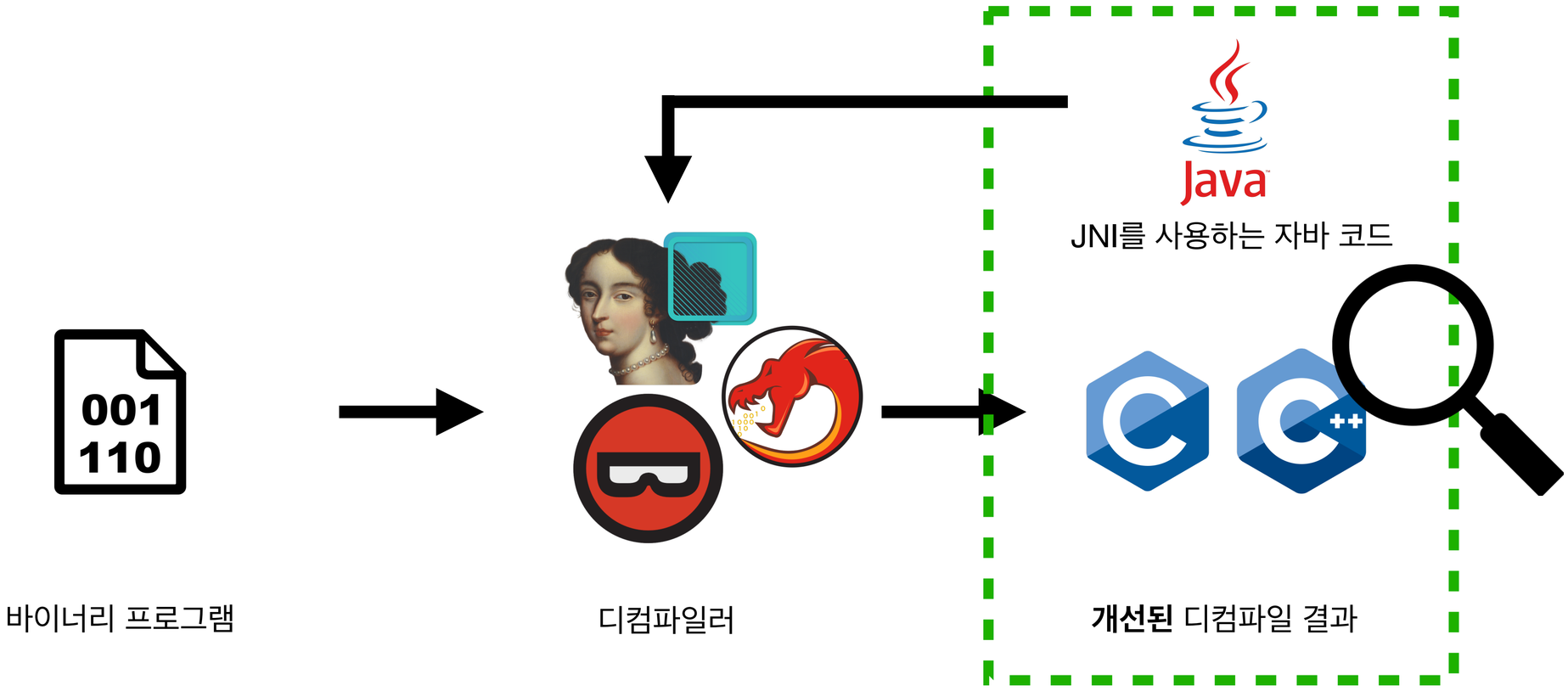
```
jstring Java_get_1column_1name(
  JNIEnv* env, jclass cls, jlong tidx, jint cidx)
{
   jstring result;
   char* name = access_column_name(tidx, cidx);
    if (name)
      result =(*env)->NewStringUTF(env, name);
    else
      result = NULL;
    return result;
}
```

원본 코드

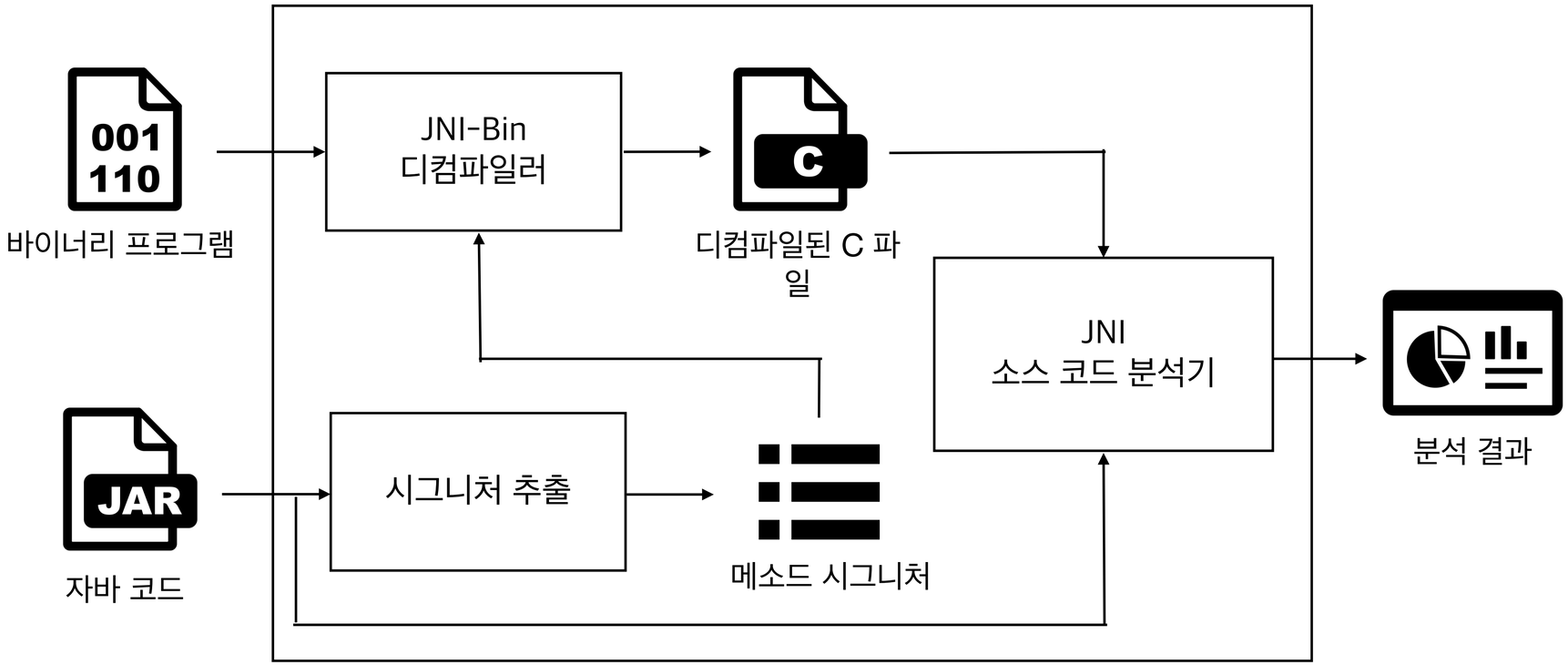
```
int Java_get_1column_1name(
  int a1, int a2, int a3, int a4, int a5)
{
   int result;
    if (sub_F1AC(a3, a5))
      result = (*(int (**)(int, int))
                (*(_DWORD *)a1 + 668))(a1);
    else
      result = 0;
    return result;
}
```

디컴파일된 코드

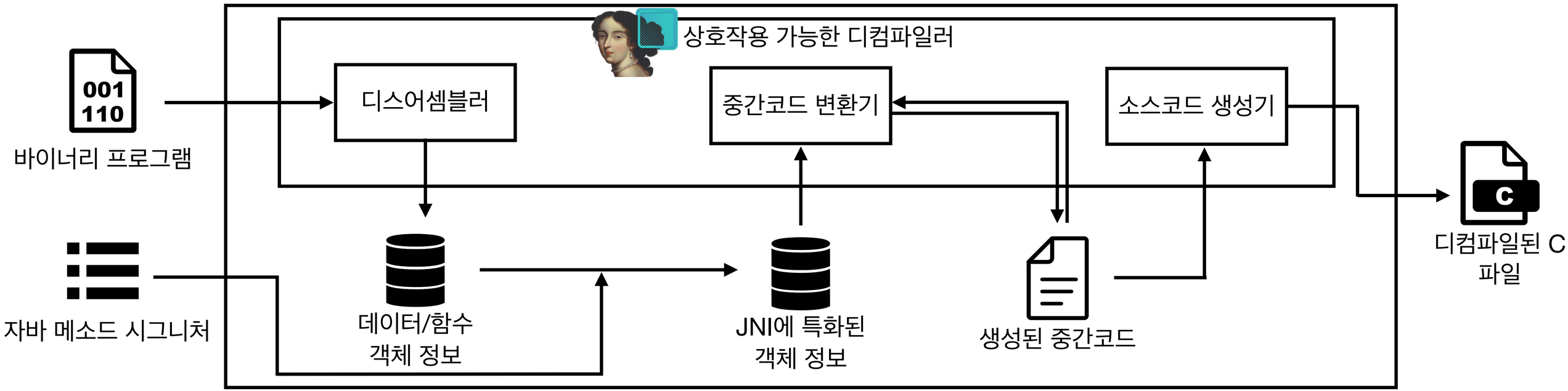
아이디어: 자바 코드의 정보를 이용해 디컴파일 품질 개선



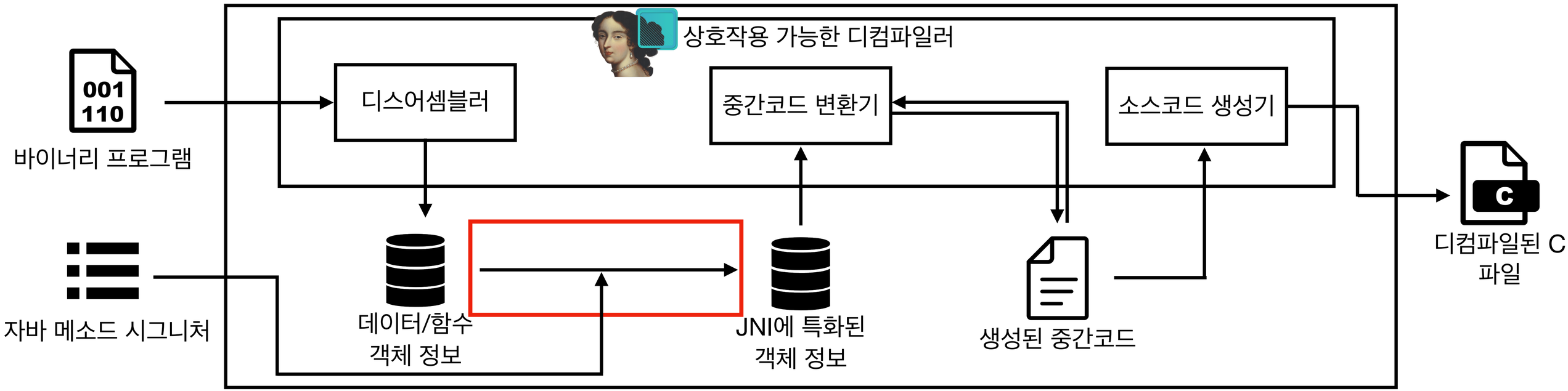
디컴파일러 기반 JNI 분석기의 전체 구조



JNI-Bin 디컴파일러



1. Java 코드에서 JNI 함수 시그니처 알아내기



1. Java 코드에서 JNI 함수 시그니처 알아내기

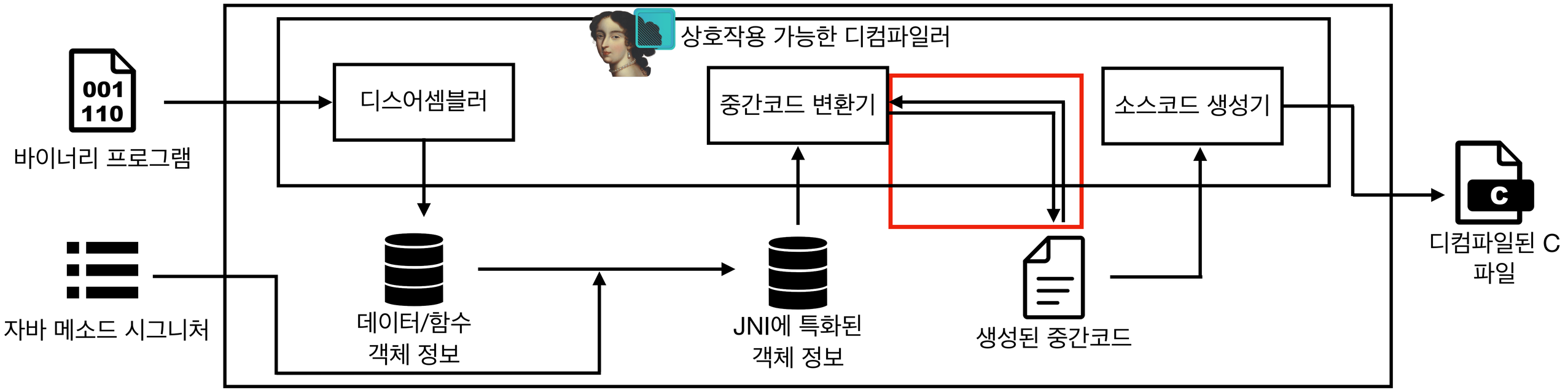
Java 코드에서 메소드의 시그니처 추출 후 대응되는 JNI 함수 시그니처로 재작성

```
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. JNI와 관련된 타입 전파하기



2. JNI와 관련된 타입 전파하기

디컴파일러가 JNI 관련 타입을 JNI 비관련 타입으로 형변환하는 경우 모두 JNI 관련 타입으로 수정 후 전파

```
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"));
}
```

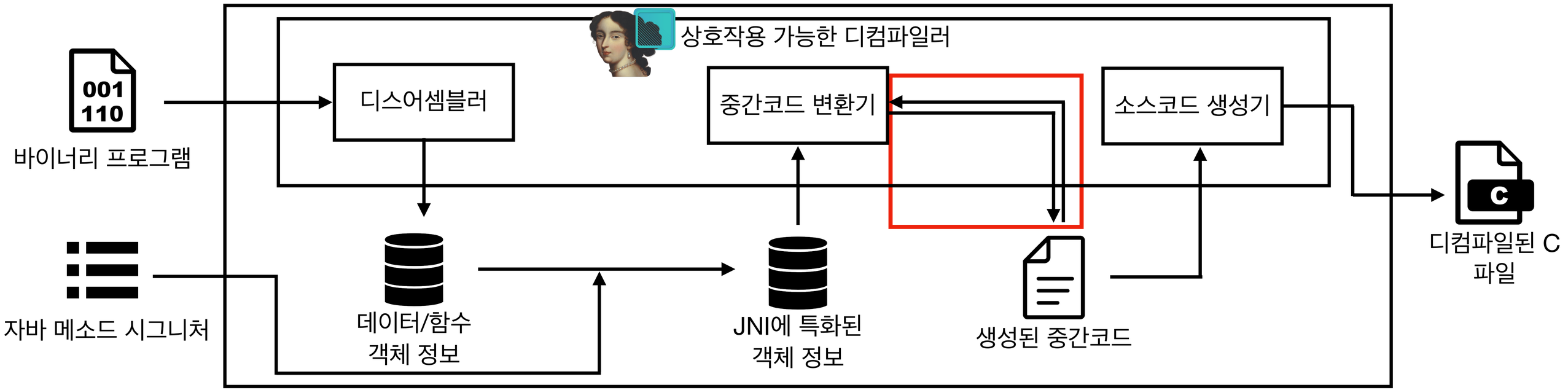
```
int Java_createFrame(JNIEnv* a1, ...) {
  ...
  jbitmap = (jobject)create_graphics(
    a1, v69[0], v69[1]);
  ...
}

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

JNI 타입이 있는 함수 집합

{ Java_createFrame }

{ Java_createFrame, create_graphics }

3. JNI 인터페이스 함수 시그니처 알아내기



3-1. 고정 시그니처의 JNI 인터페이스 함수

디컴파일러가 제공하는 API를 이용해 호출 시그니처 수정

```
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. 가변 시그니처의 JNI 인터페이스 함수

실제 Java 메소드를 가리킬 때 쓰이는 jmethodID 타입 변수에 대한 값 분석

분석 결과를 기반으로 디컴파일러 API를 이용해 시그니처 수정

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

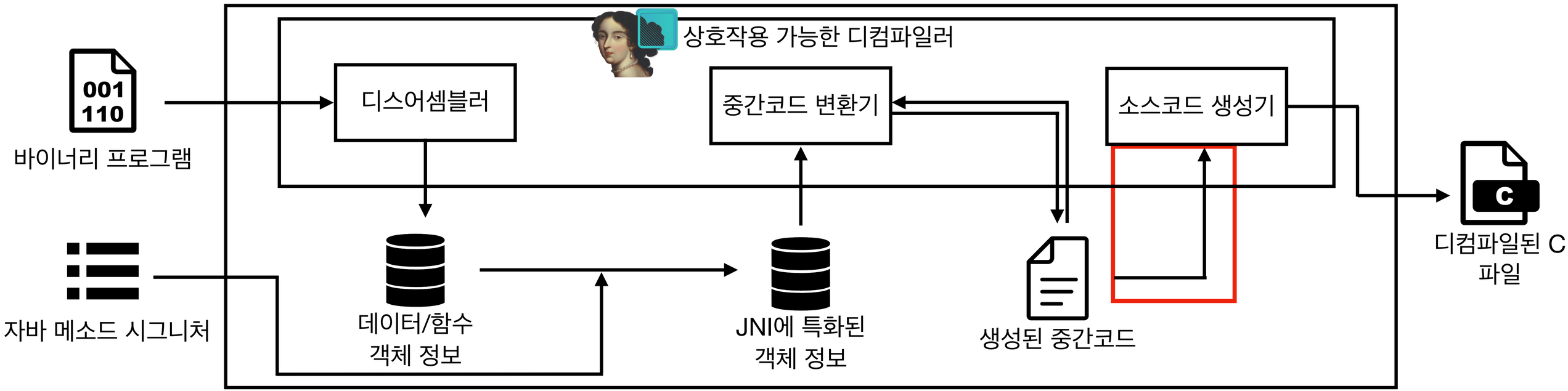
```
jstring Java_prepare(JNIEnv* a1, jobject a2,
  jobject a3, jobject a4, jint a5) {
 ...
  v14 = (*a1)->GetObjectClass(a1, a3);
  if ( v14 ) {
    v11 = (*a1)->GetMethodID(a1,
      v14, "set_param", "(Ljava/lang/String;)I");
    ...
    v10 = (*a1)->CallIntMethod(a1, a3, v11, v17);
   ...
  }
}
```

```
int CallIntMethod(JNIEnv*, object, jmethodID, jstring)
```

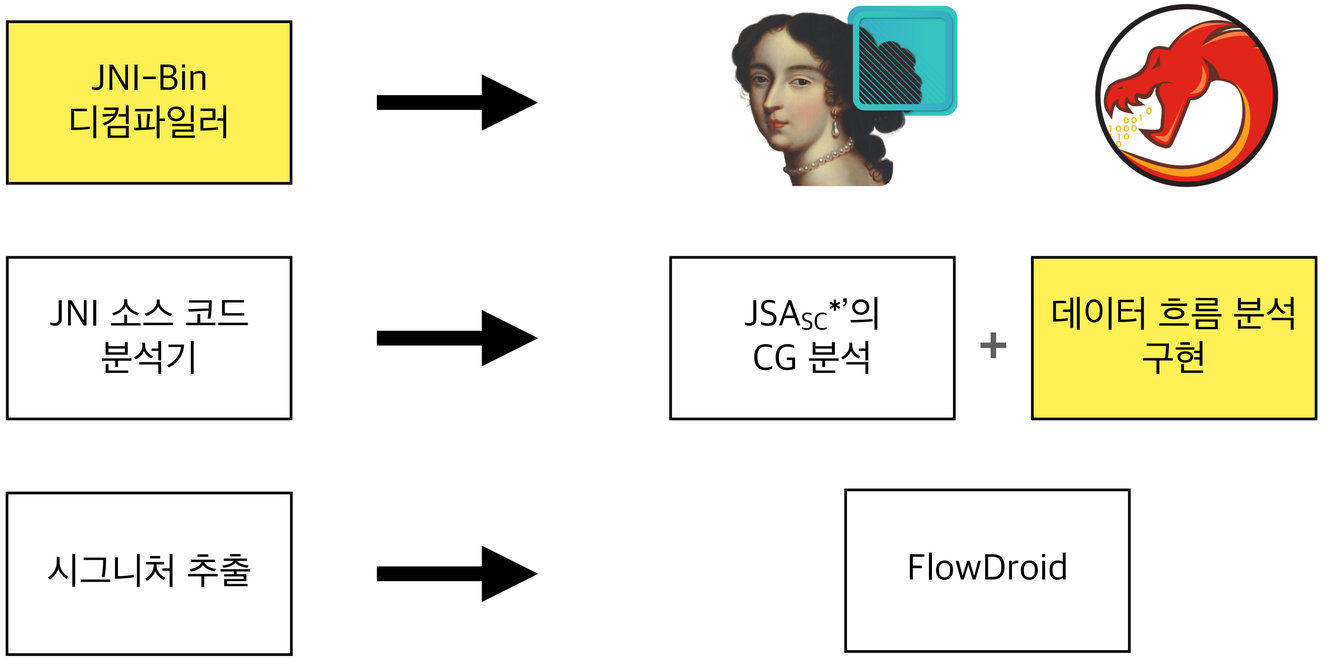
4. 컴파일 가능한 소스 코드 만들기

문법 오류/매크로 오류, 선언되지 않은 변수 등의 문제

각 경우마다 휴리스틱을 이용, 휴리스틱으로 커버되지 않으면 삭제



구현 및 실험



* 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

실험 타겟

NativeFlowBench – 특별히 어려운 JNI 상호작용을 가진 16개의 벤치마크 앱

작은 코드 LoC (8-42)

F-Droid – JNI를 사용하는 10개의 오픈 소스 안드로이드 앱

상대적으로 큰 코드 LoC(42-13226)

CG 분석 결과 – NativeFlowBench

모든 Java -> C 호출하는 함수 이름

모든 C -> Java 호출하는 함수 이름 / 접근하는 field 이름 정확히 분석

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

CG 분석 결과 – F-Droid

90%(Ghidra)-100%(IDA-Pro) 정확하게 분석

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

데이터 흐름 분석 결과 비교

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)

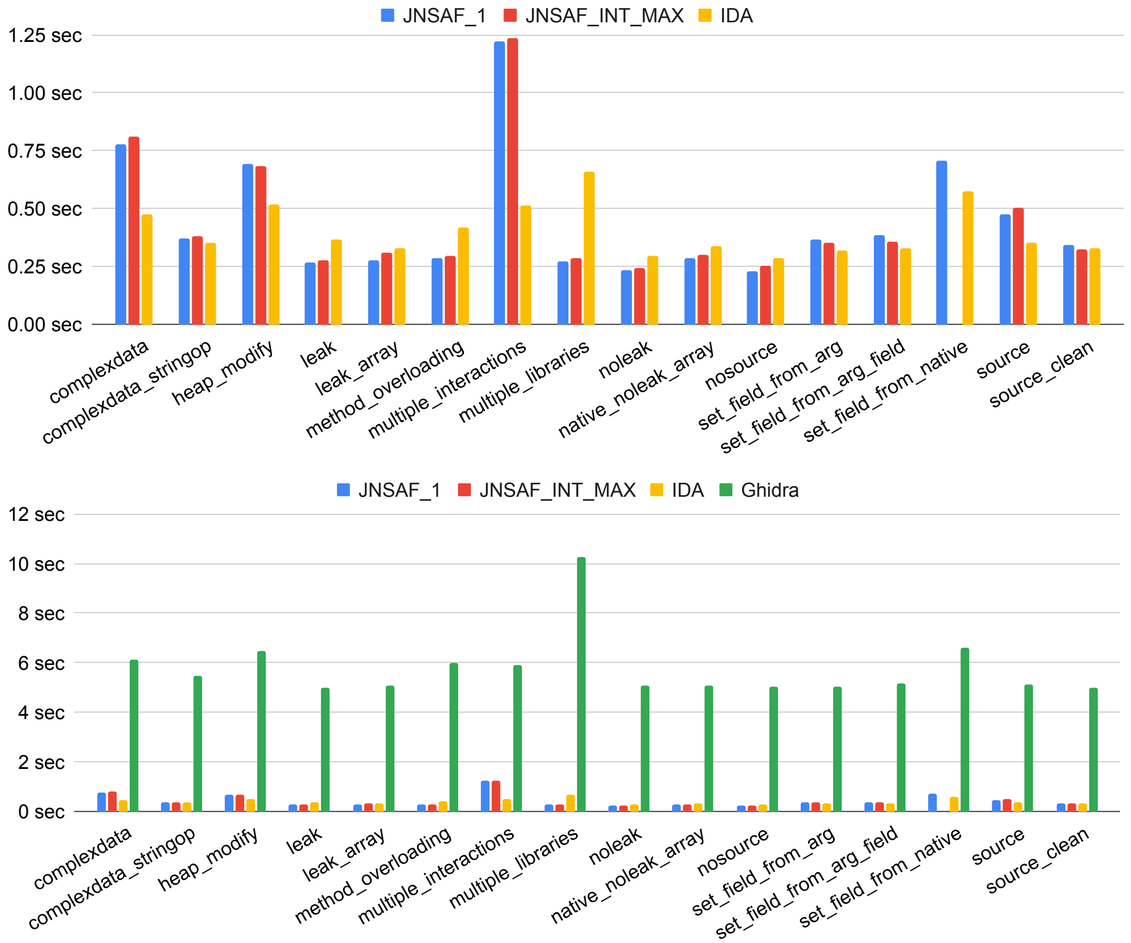
Recall

100% (Ours) vs 64%(JN-SAF)

JN-SAF의 경우 1개의 앱에 대해 분석 시간 초과

PLRG Programming Language Research Group

데이터 흐름 분석 실행 시간 비교



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

	$JN\text{-}SAF_{1}$	$JN\text{-}SAF_{INT\_MAX}$	$JSA_{DEC\text{-}IDA}$	$JSA_{DEC\text{-}Ghidra}$
Time (s)	0.432	0.441	0.392	5.718

PLRG Programming Language Research Group

한계점 & 향후 연구

한계점

- 난독화된 바이너리에 대해서는 적용 불가능
- 디컴파일러 결과의 정확성에 대해서 어떤 보장도 할 수 없음

향후 연구

- 바이너리에서 고급 언어로 변환할 때의 정확성을 보장하기 위한 이론 및 변환기 개발