DryJIN: Detecting Information Leaks in Android Applications

Minseong Choi[1], Yubin Im[2], Steve Ko[3], Yonghwi Kwon[4], Yuseok Jeon[1], Haehyun Cho[2]

[1] UNIST, [2] Soongsil University, [3] Simon Fraser University, [4] University of Maryland

Minseong Choi
E-mail : liberty@unist.ac.kr

Yubin Im
E-mail : th8548@soongsil.ac.kr

Steve Ko
E-mail : steveyko@sfu.ca

Yonghwi Kwon
E-mail : yongkwon@umd.edu

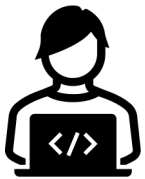
Yuseok Jeon
E-mail : ysjeon@unist.ac.kr

Haehyun Cho
E-mail : haehyun@ssu.ac.kr

Threat of Android Application

Threat of Android Application

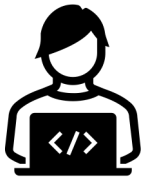
Attacker’s Intention



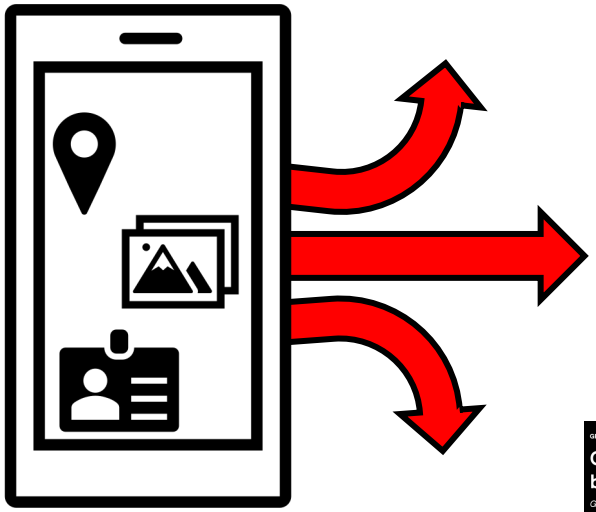
Developer’s Fault

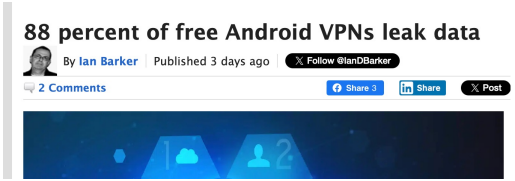
Threat of Android Application

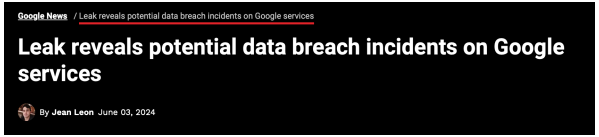
Attacker's Intention

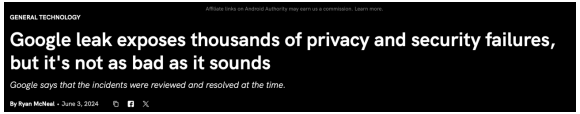


Developer's Fault



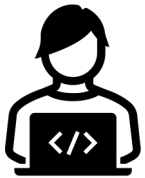




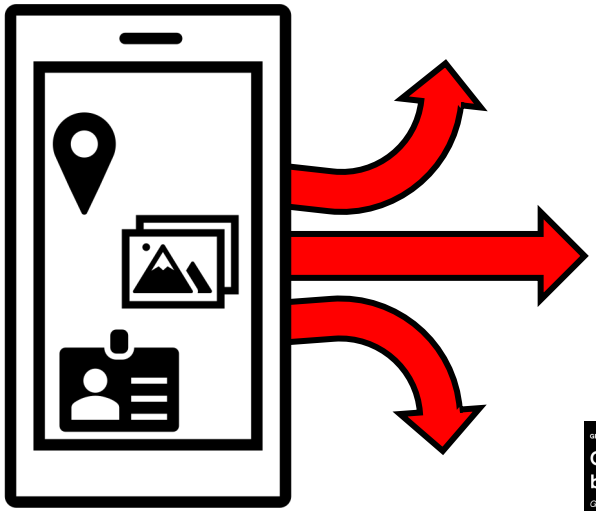


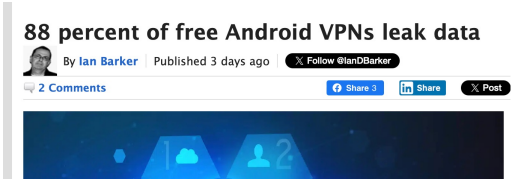
Threat of Android Application

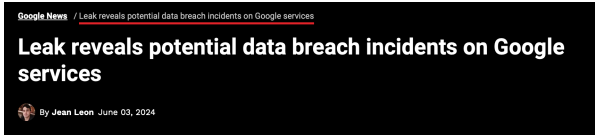
Attacker's Intention

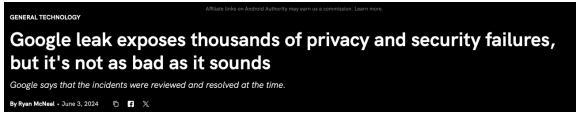


Developer's Fault





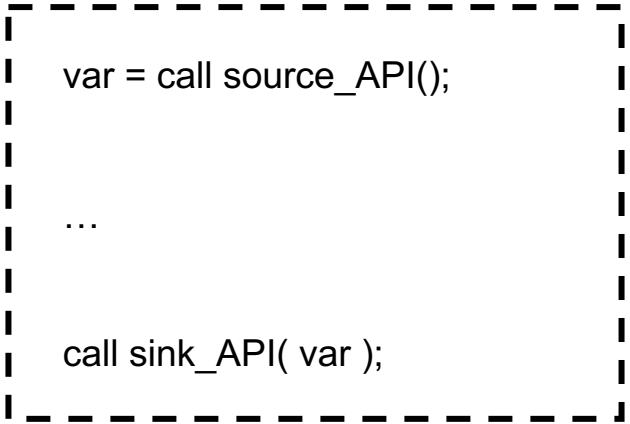




Information leaks in Android are a common issue.

Information Leak Detection in Android

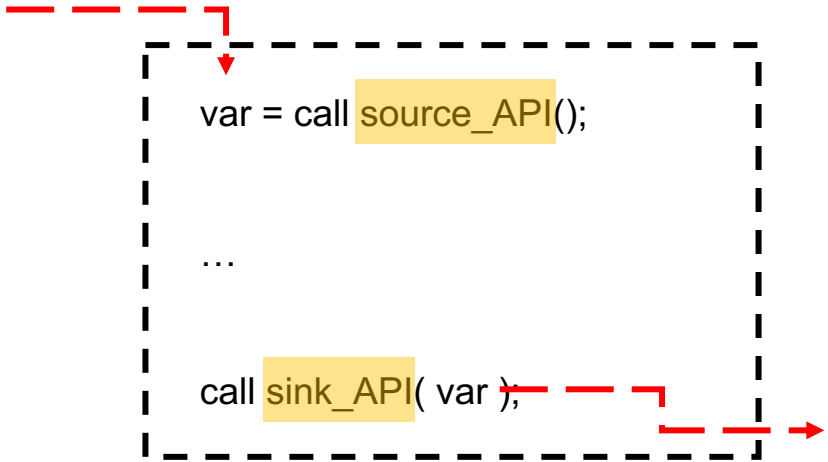
- **Path reachability problem** of a specific data.



Application Process

Information Leak Detection in Android

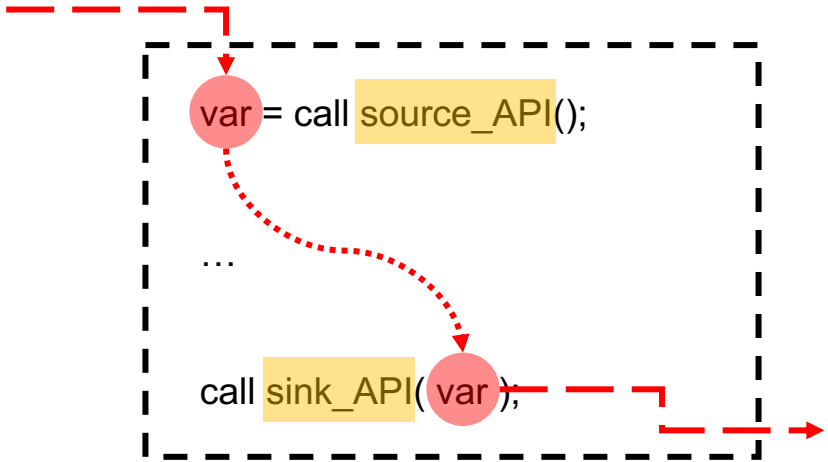
- **Path reachability problem** of a specific data.
- Identifying APIs to read sensitive information (i.e., source) and write out of an app (i.e., sink).



Application Process

Information Leak Detection in Android

- **Path reachability problem** of a specific data.
- Identifying APIs to read sensitive information (i.e., source) and write out of an app (i.e., sink).
- Taint analysis **traces data flows** between them.

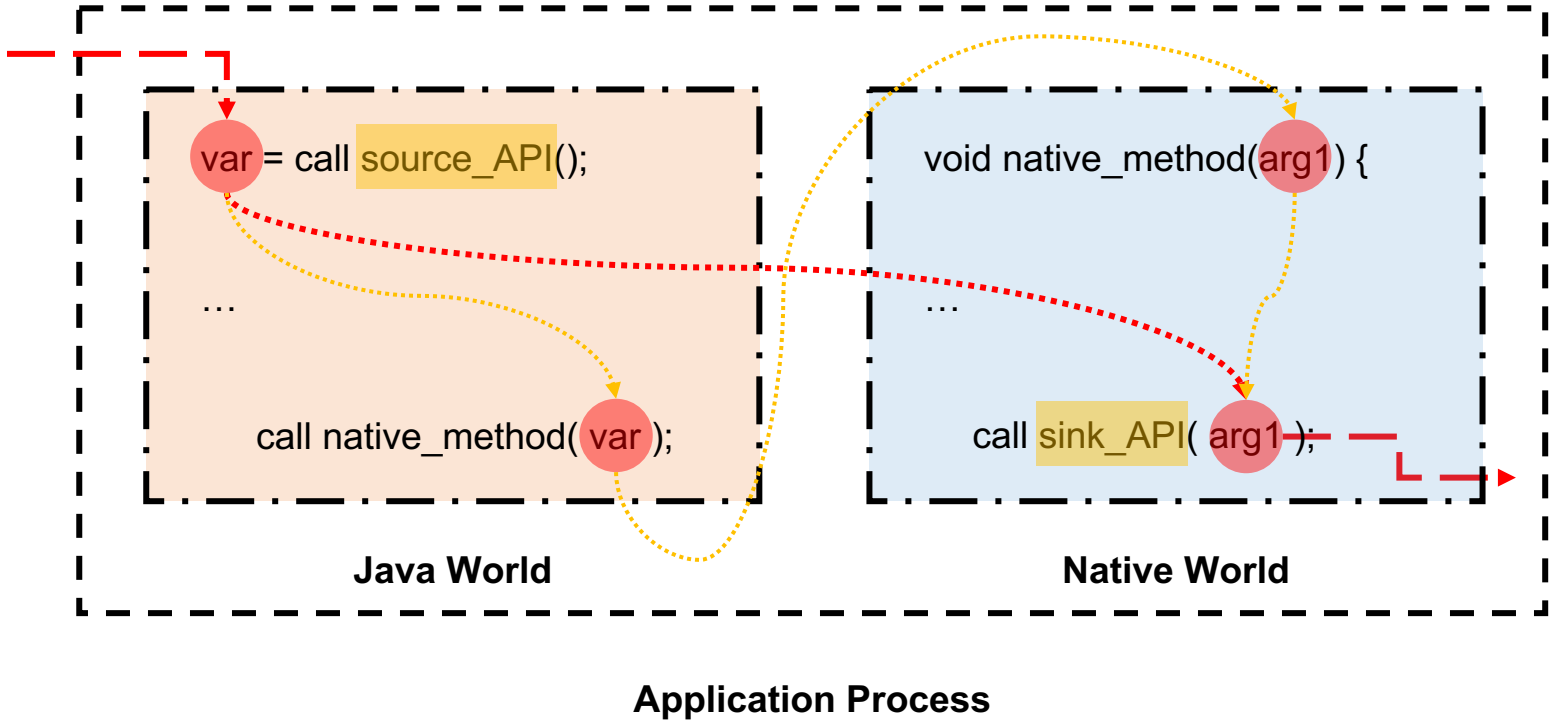


Android's Nature: Cross-language Module

- **Native library** is compiled codes by using **C/C++**.

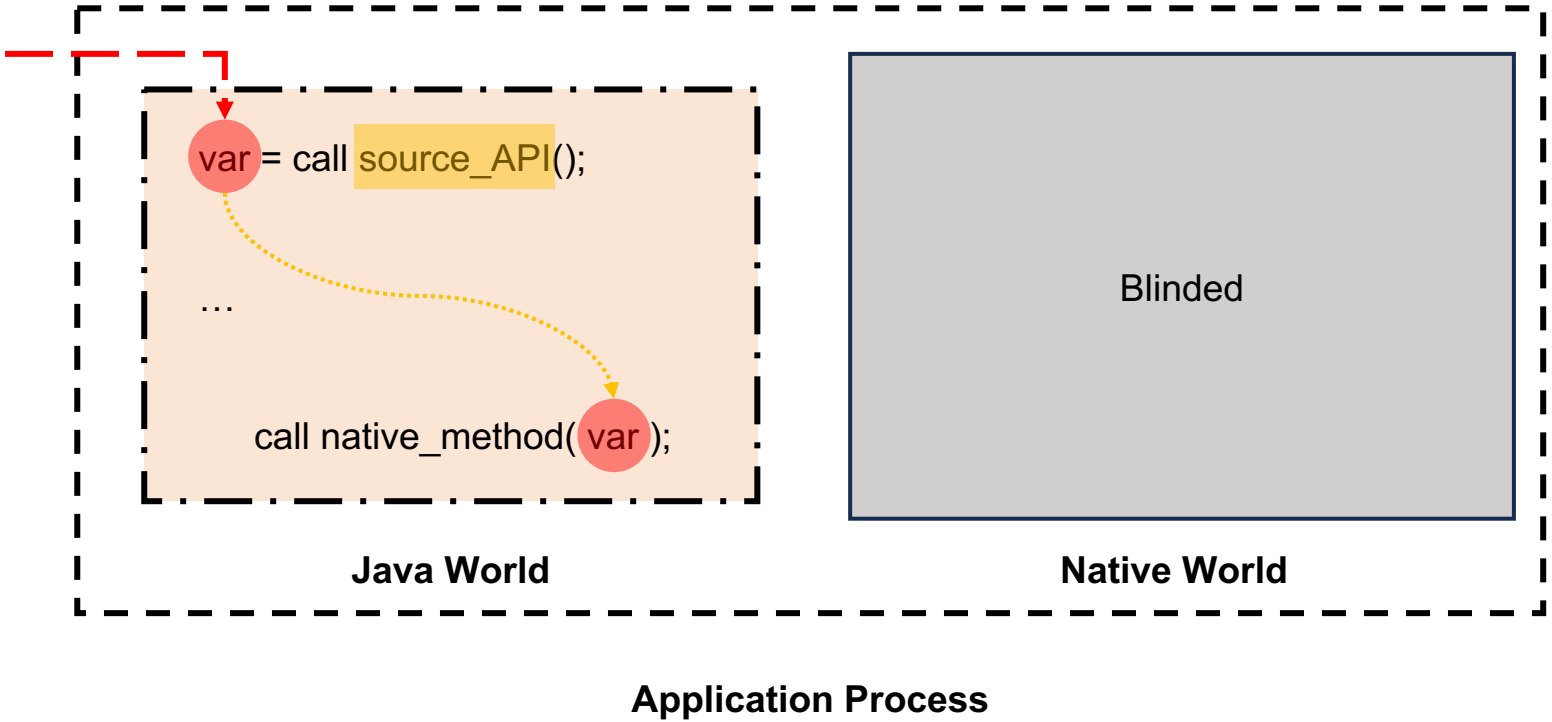
Android's Nature: Cross-language Module

❖ **Native library** is compiled codes by using **C/C++**.



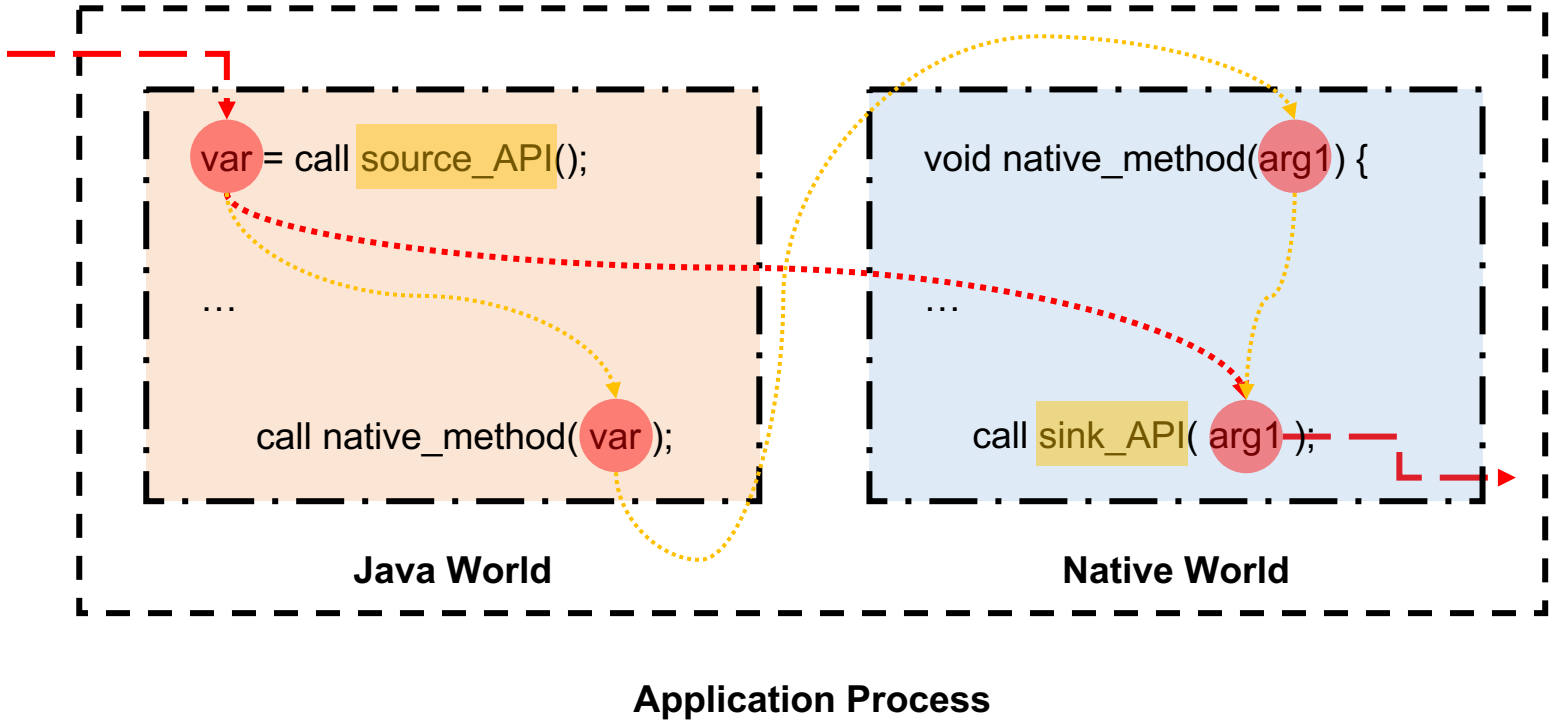
Android's Nature: Cross-language Module

❖ **Native library** is compiled codes by using **C/C++**.



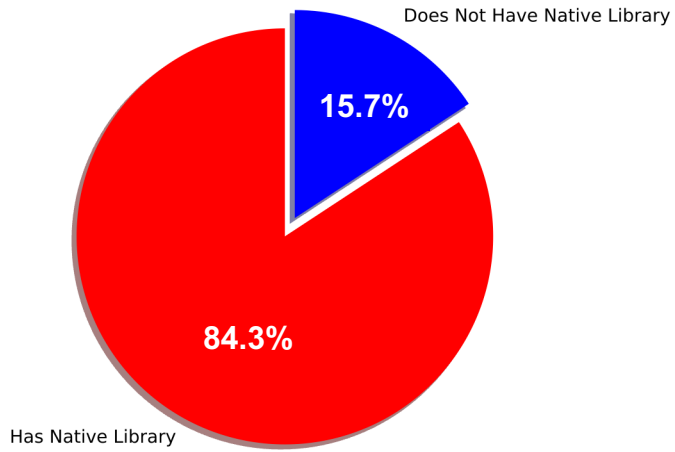
Android's Nature: Cross-language Module

❖ **Native library** is compiled codes by using **C/C++**.



Android's Nature: Cross-language Module

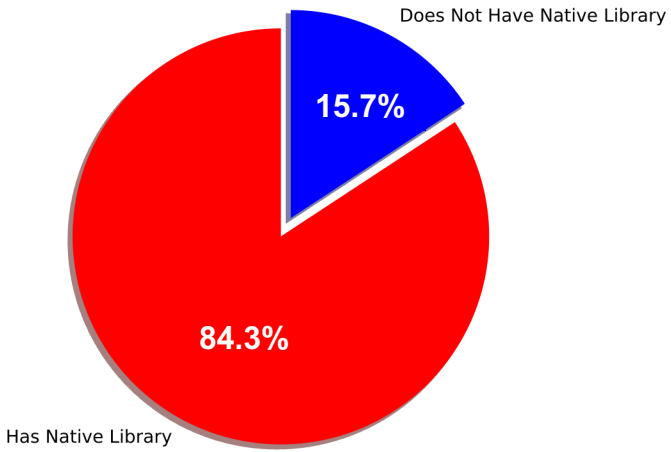
- **Native library** is compiled codes by using **C/C++**.
- It takes a large portion (84.3%) in **malware** market



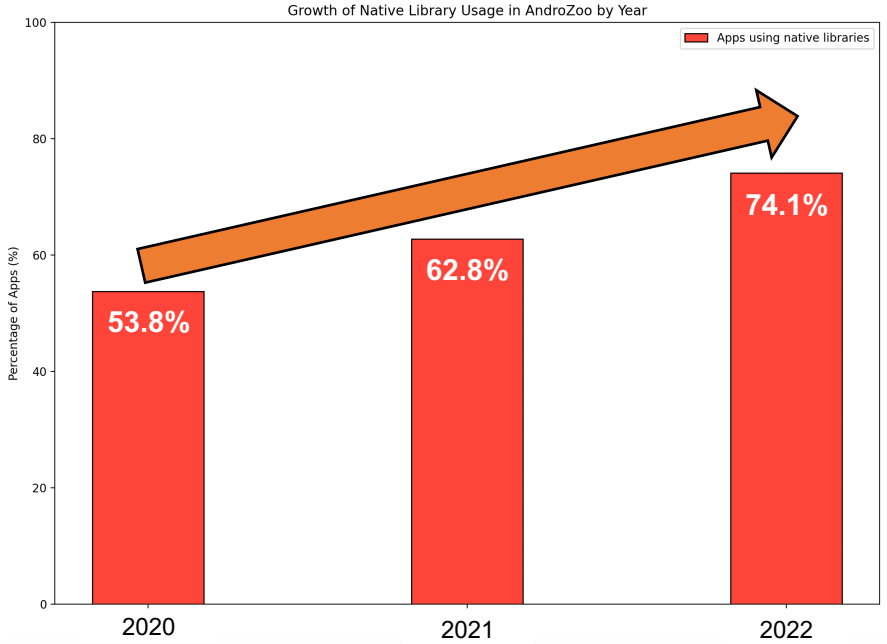
Malware

Android's Nature: Cross-language Module

- **Native library** is compiled codes by using **C/C++**.
- It takes a large portion (84.3%) in **malware** market and is growing in **benign-ware**.

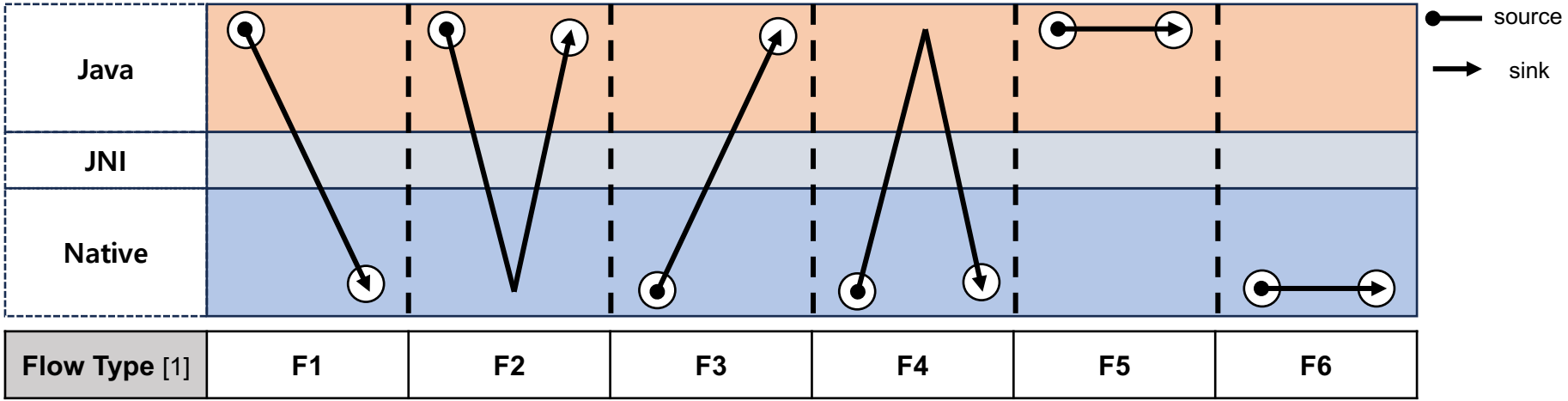


Malware



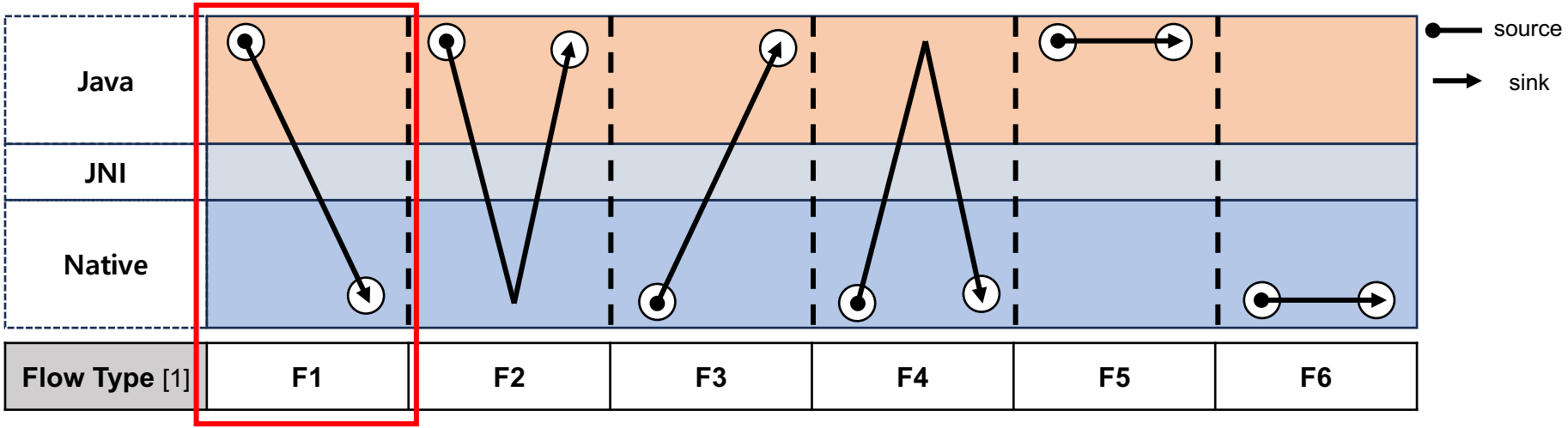
Benign-ware

Cross-language Attack Vectors on Information Flow



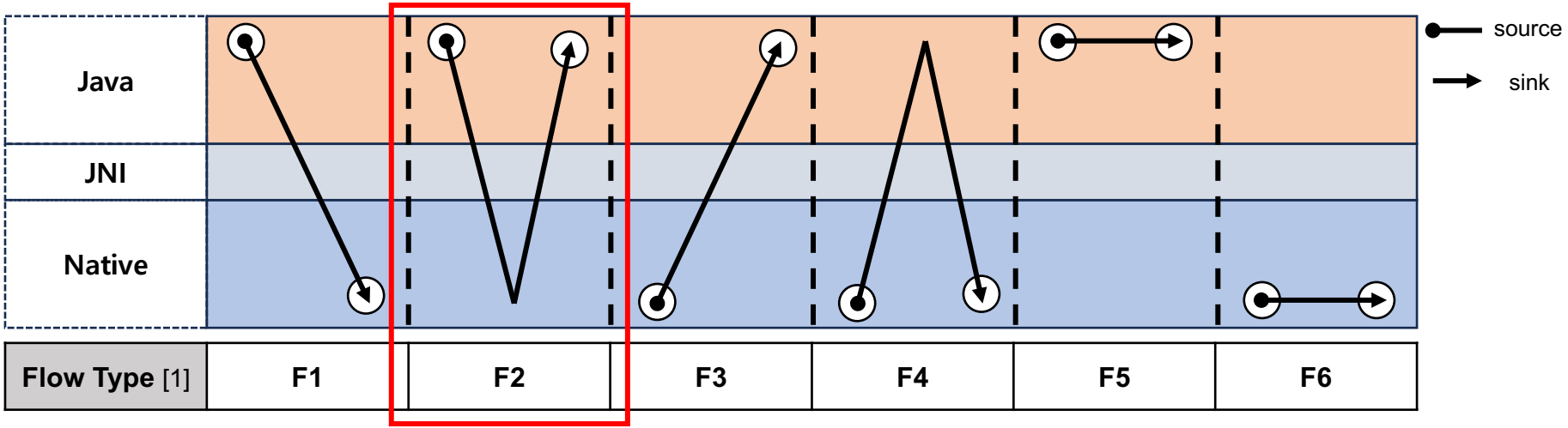
[1] Xue, Lei, et al. "NDroid: Toward tracking information flows across multiple Android contexts." *IEEE Transactions on Information Forensics and Security* 14.3 (2018): 814-828.

Cross-language Attack Vectors on Information Flow



[1] Xue, Lei, et al. "NDroid: Toward tracking information flows across multiple Android contexts." *IEEE Transactions on Information Forensics and Security* 14.3 (2018): 814-828.

Cross-language Attack Vectors on Information Flow



[1] Xue, Lei, et al. "NDroid: Toward tracking information flows across multiple Android contexts." *IEEE Transactions on Information Forensics and Security* 14.3 (2018): 814-828.

Problem Statement of Existing Approaches

❖ **FlowDroid (PLDI '14):** IFDS-based taint analyzer on java code.

Approach	F1	F2	F3	F4	F5	F6
FlowDroid	❌	❌	❌	❌	✅	❌

Problem Statement of Existing Approaches

- **FlowDroid (PLDI '14):** IFDS-based taint analyzer on java code.

- **Argus-SAF (CCS '18):** Summary-based taint analyzer on java code and native code.
  - Missing for native source APIs.
  - Capturing data flow in native code only the invocation of the source or sink java API.

Approach	F1	F2	F3	F4	F5	F6
FlowDroid	❌	❌	❌	❌	✅	❌
Argus-SAF	✅	✅	❌	❌	✅	❌

Problem Statement of Existing Approaches

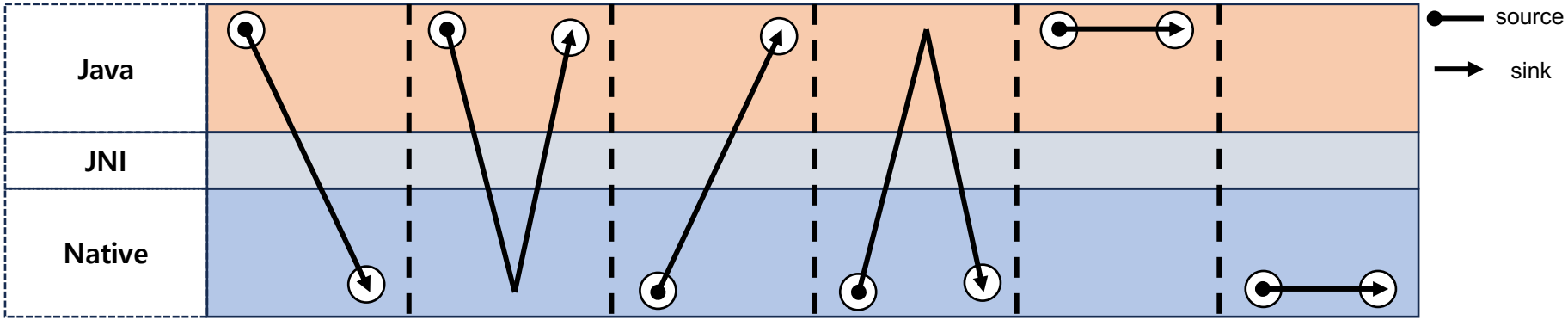
- **FlowDroid (PLDI '14):** IFDS-based taint analyzer on java code.

- **Argus-SAF (CCS '18):** Summary-based taint analyzer on java code and native code.
  - Missing for native source APIs.
  - Capturing data flow in native code only the invocation of the source or sink java API.

- **JuCify (ICSE '22):** Adapting native code into FlowDroid by translation.
  - Missing for native source and sink APIs.
  - Overlooking problem due to opaque argument permutation.

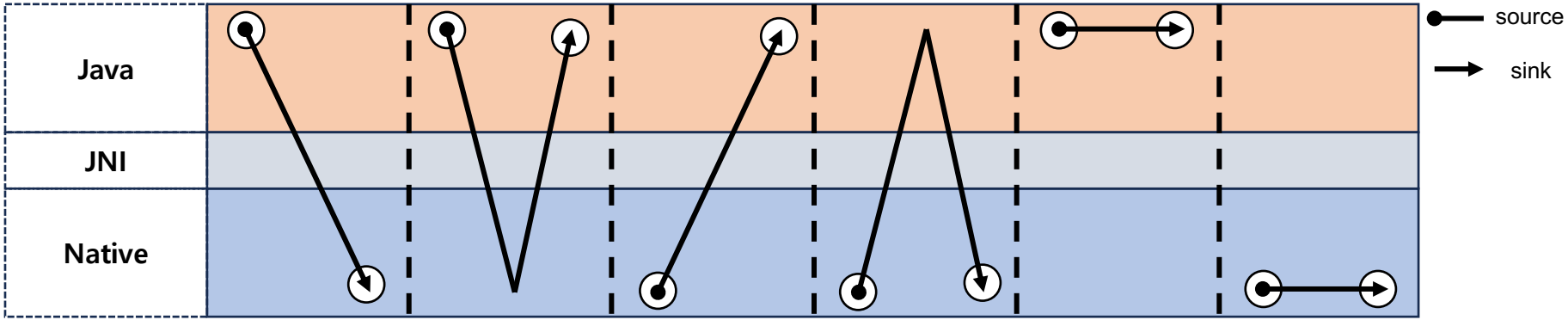
Approach	F1	F2	F3	F4	F5	F6
FlowDroid	❌	❌	❌	❌	✅	❌
Argus-SAF	✅	✅	❌	❌	✅	❌
JuCify	❌	✅	❌	❌	✅	❌

Problem Statement of Existing Approaches



Approach	F1	F2	F3	F4	F5	F6
FlowDroid	❌	❌	❌	❌	✅	❌
Argus-SAF	✅	✅	❌	❌	✅	❌
JuCify	❌	✅	❌	❌	✅	❌

Problem Statement of Existing Approaches



Approach	F1	F2	F3	F4	F5	F6
FlowDroid	❌	❌	❌	❌	✅	❌
Argus-SAF	✅	✅	❌	❌	✅	❌
JuCify	❌	✅	❌	❌	✅	❌
DryJIN	✅	✅	✅	✅	✅	✅

Overview of DryJIN

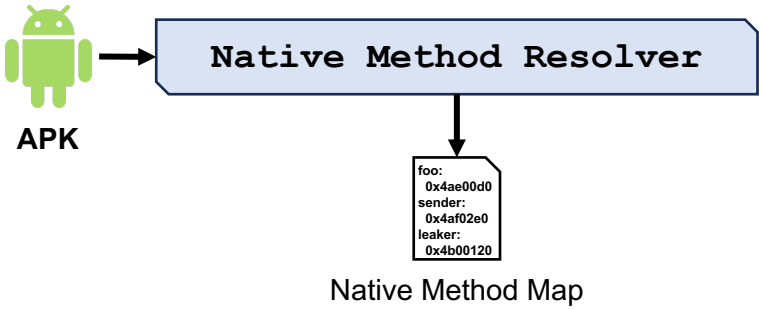
APK

Native Method Resolver

Native Code Abstractor

Java Analyzer

Overview of DryJIN - Native Method Resolver

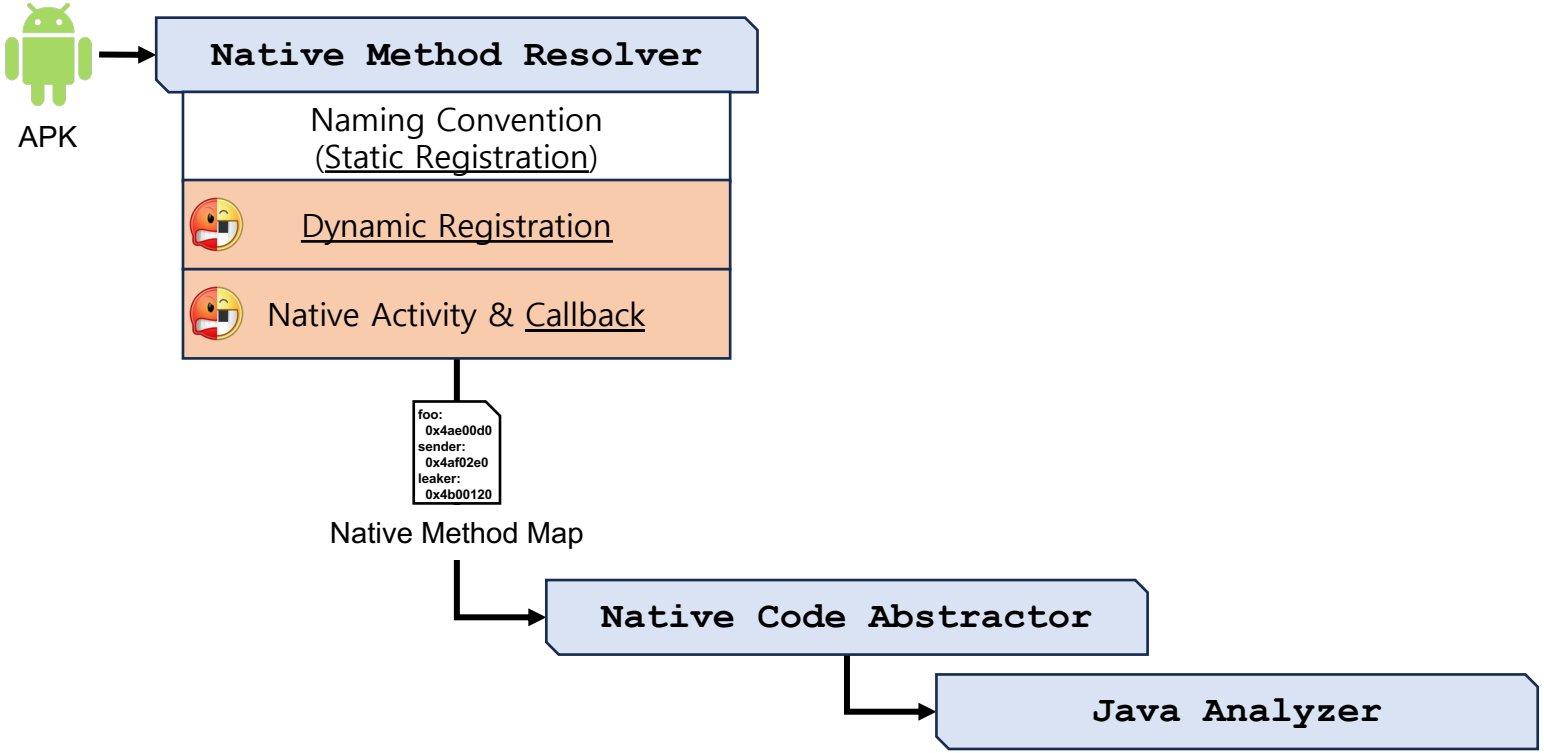


- Find **native methods** and its address within a native library.

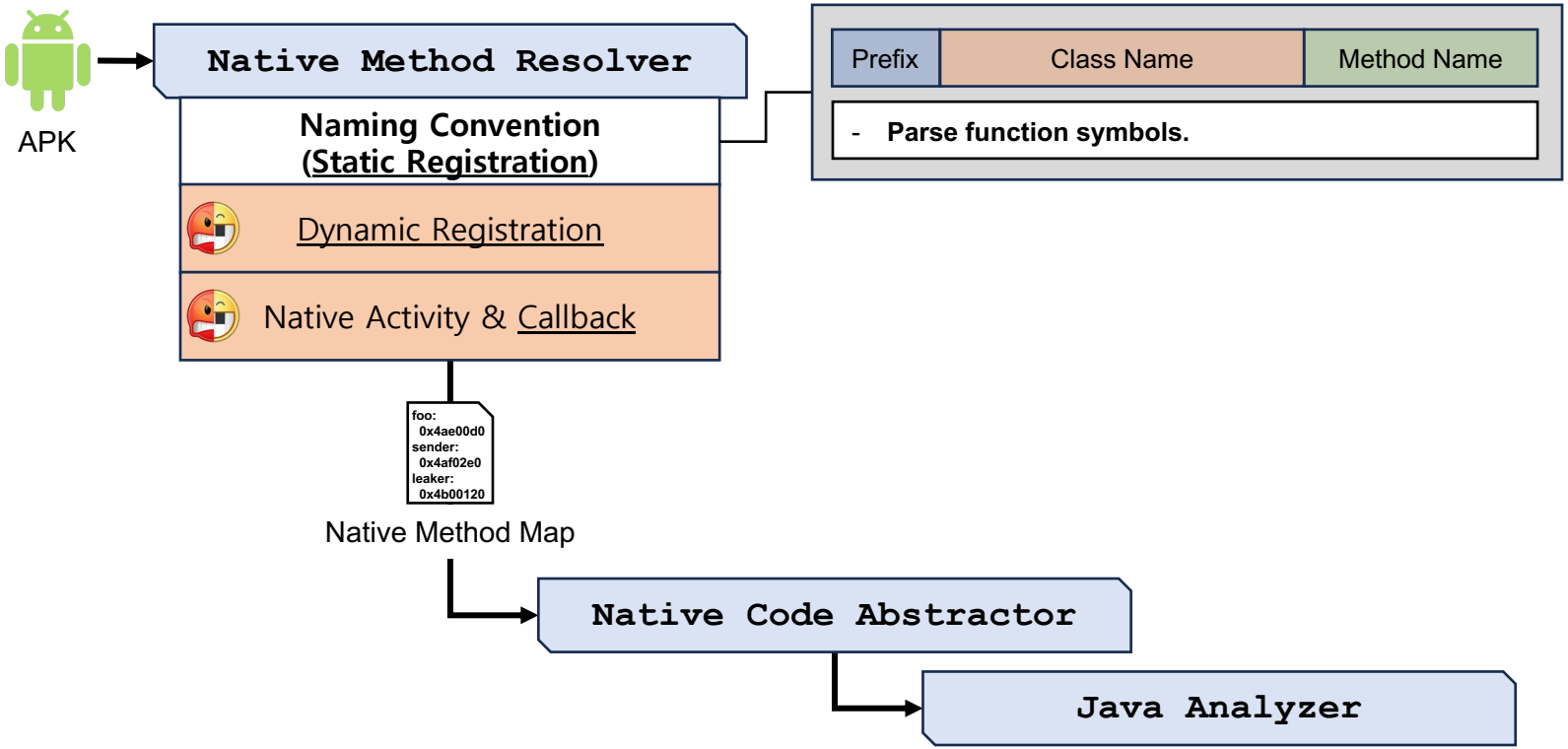
Native Code Abstractor

Java Analyzer

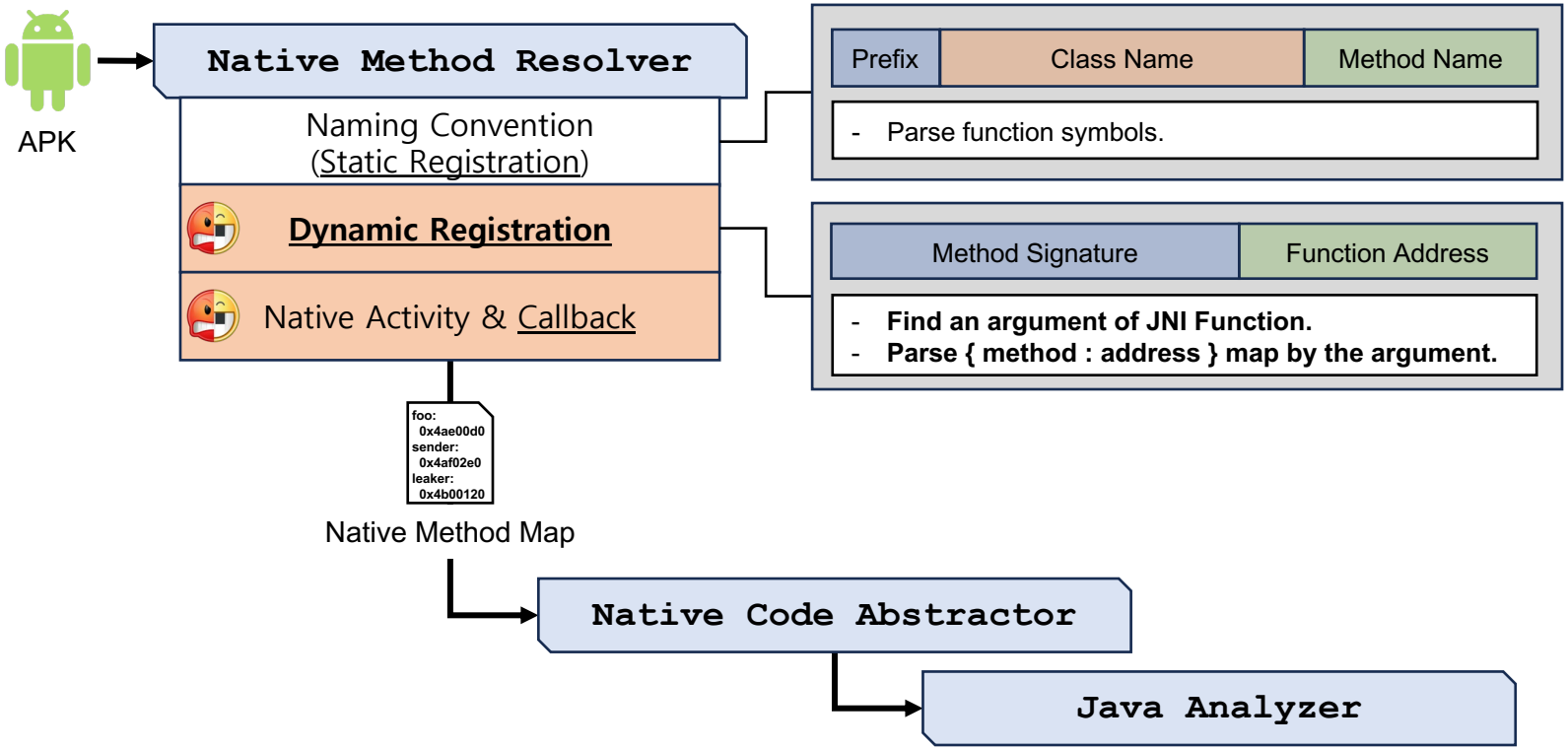
Overview of DryJIN - Native Method Resolver



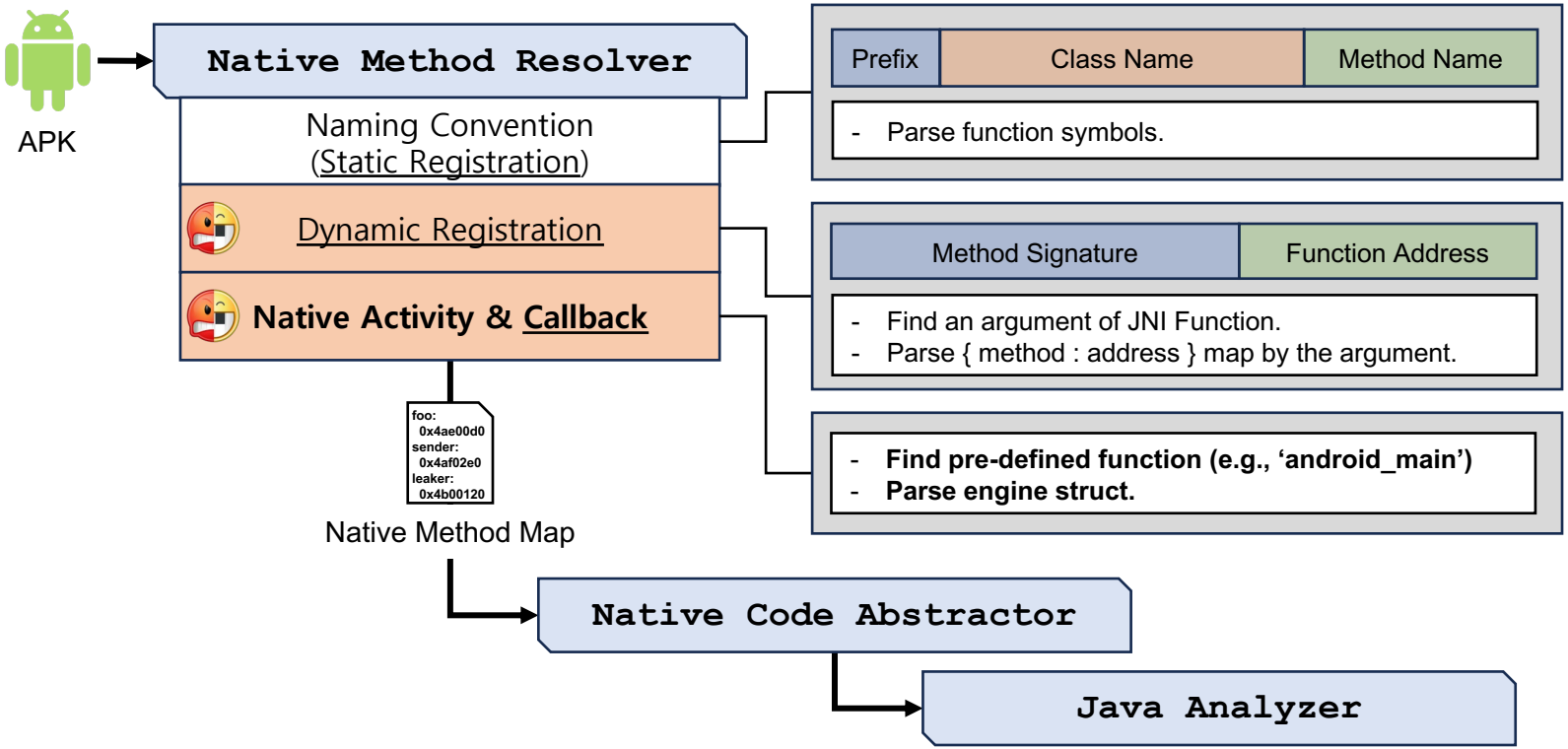
Overview of DryJIN - Native Method Resolver



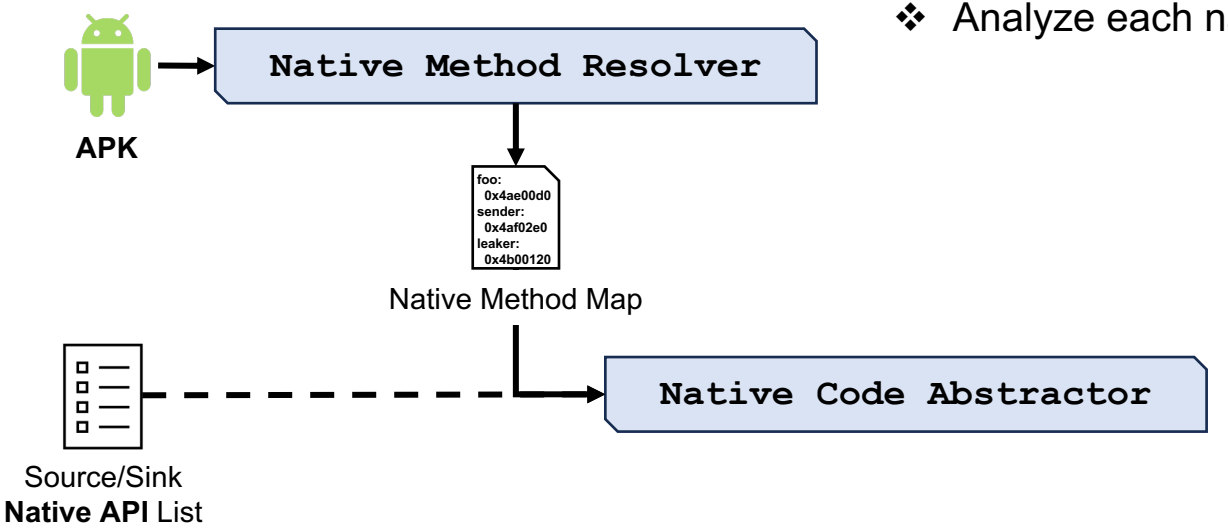
Overview of DryJIN - Native Method Resolver



Overview of DryJIN - Native Method Resolver



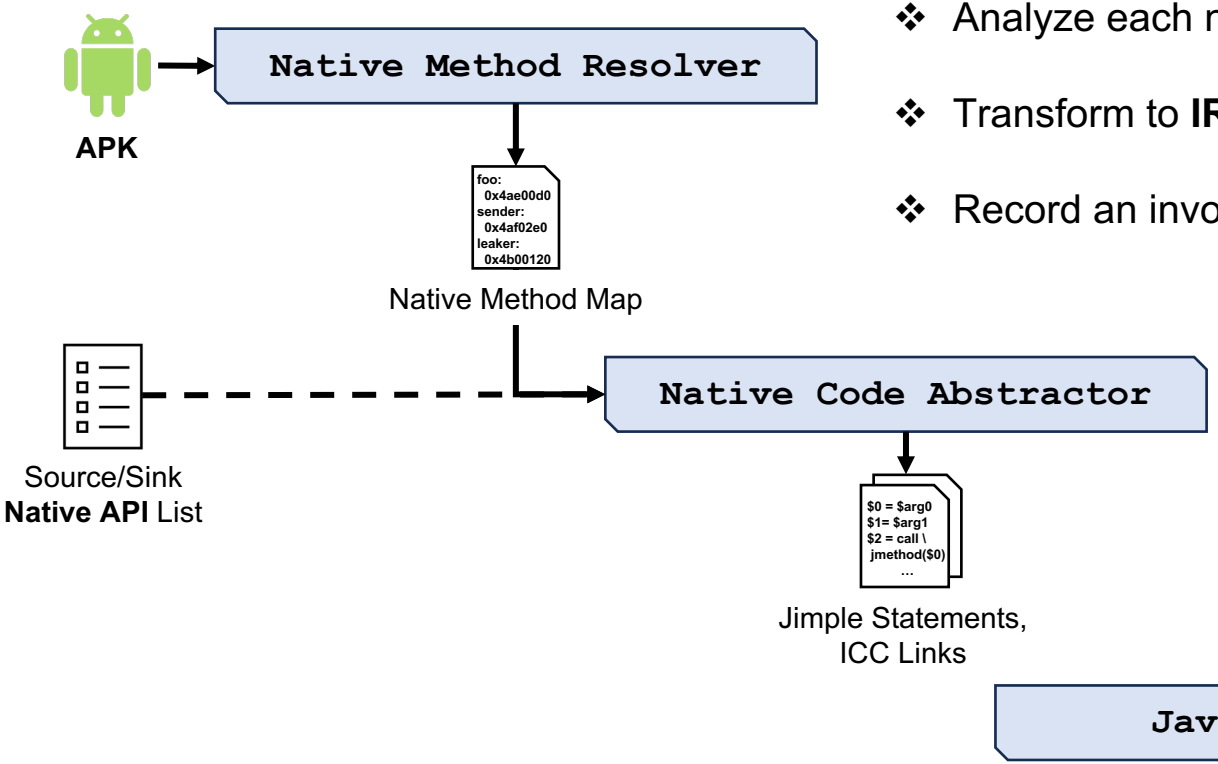
Overview of DryJIN - Native Code Abstractor



- Analyze each native method from **the address**

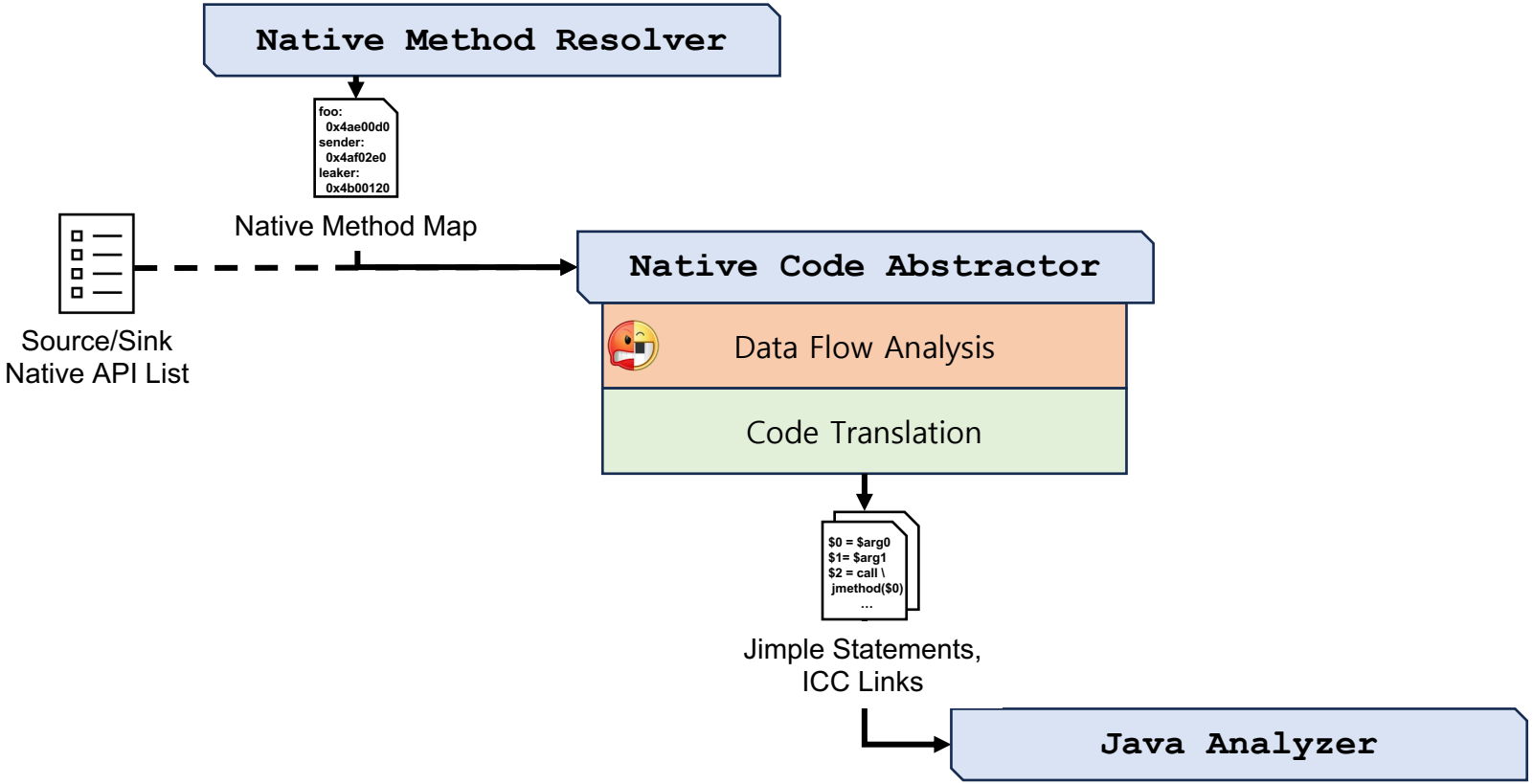
Java Analyzer

Overview of DryJIN - Native Code Abstractor

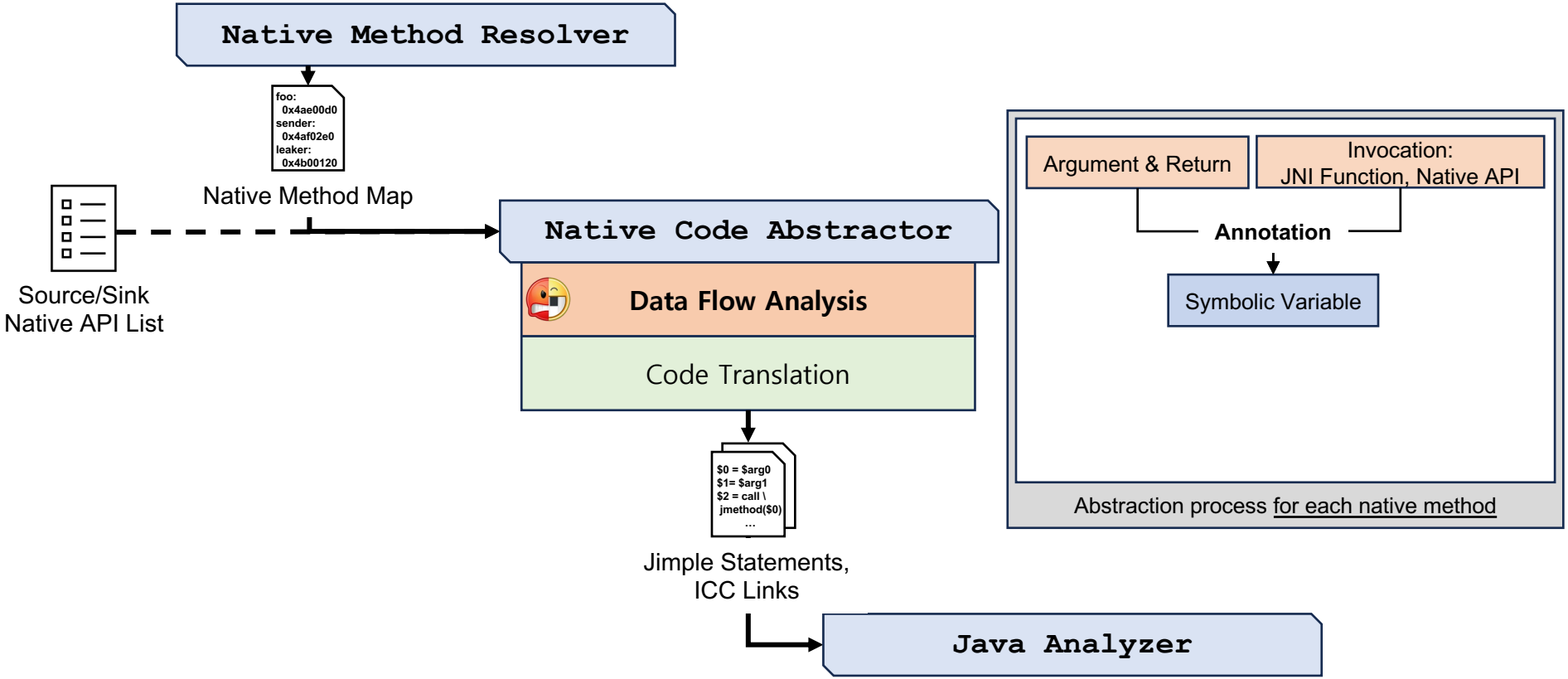


- Analyze each native method from the address
- Transform to **IR** being used for Java Analysis
- Record an invocation of **ICC**

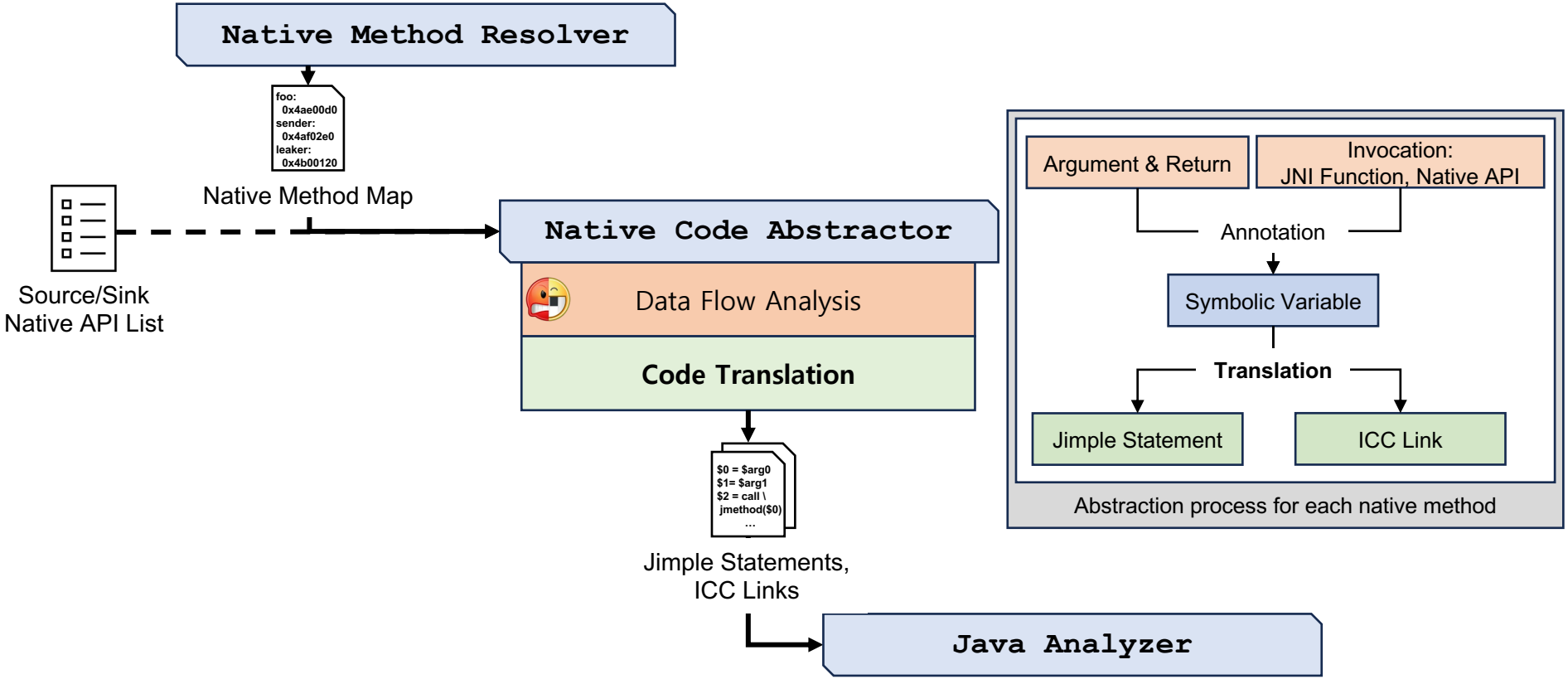
Overview of DryJIN - Native Code Abstractor



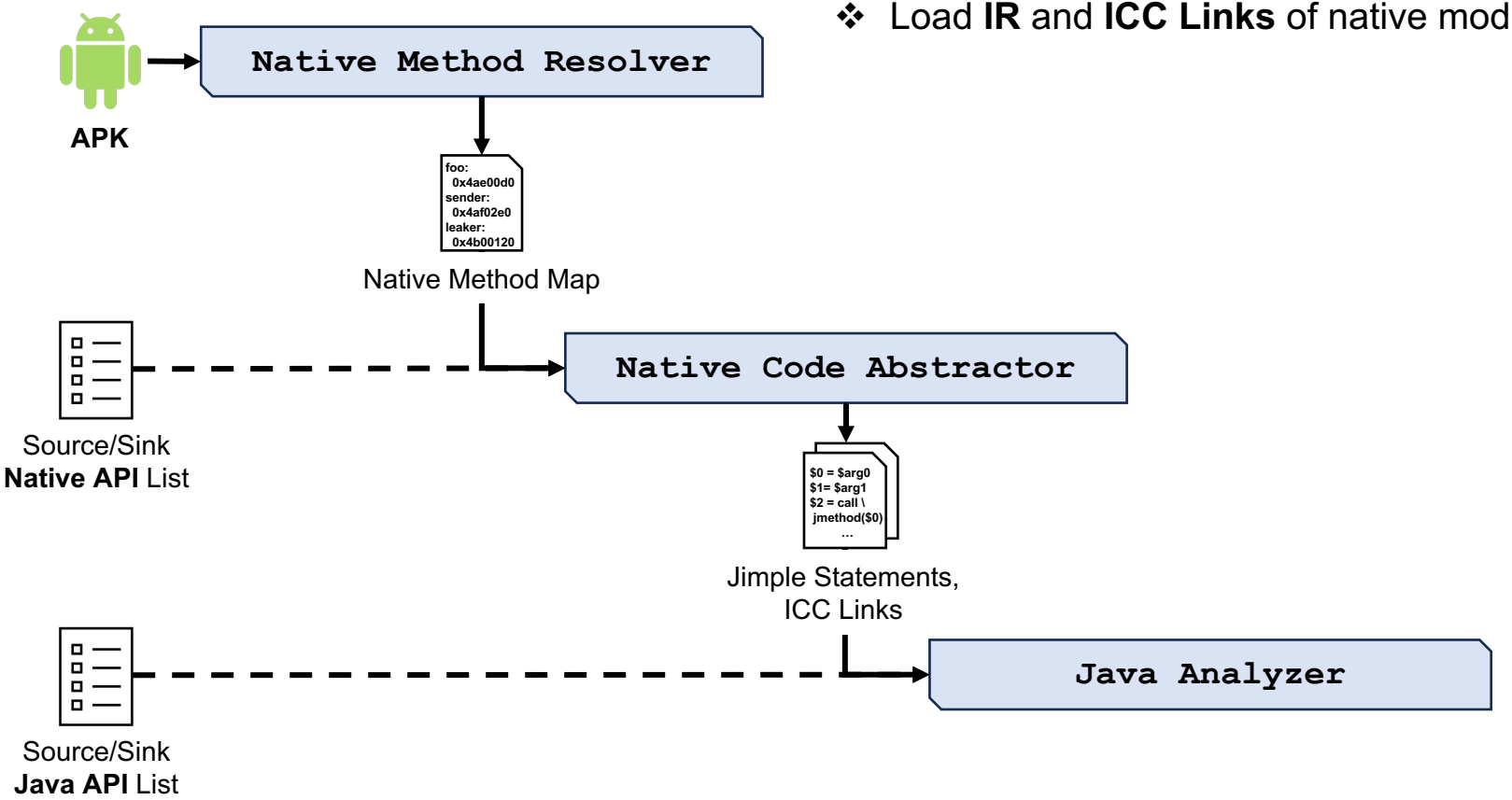
Overview of DryJIN - Native Code Abstractor



Overview of DryJIN - Native Code Abstractor

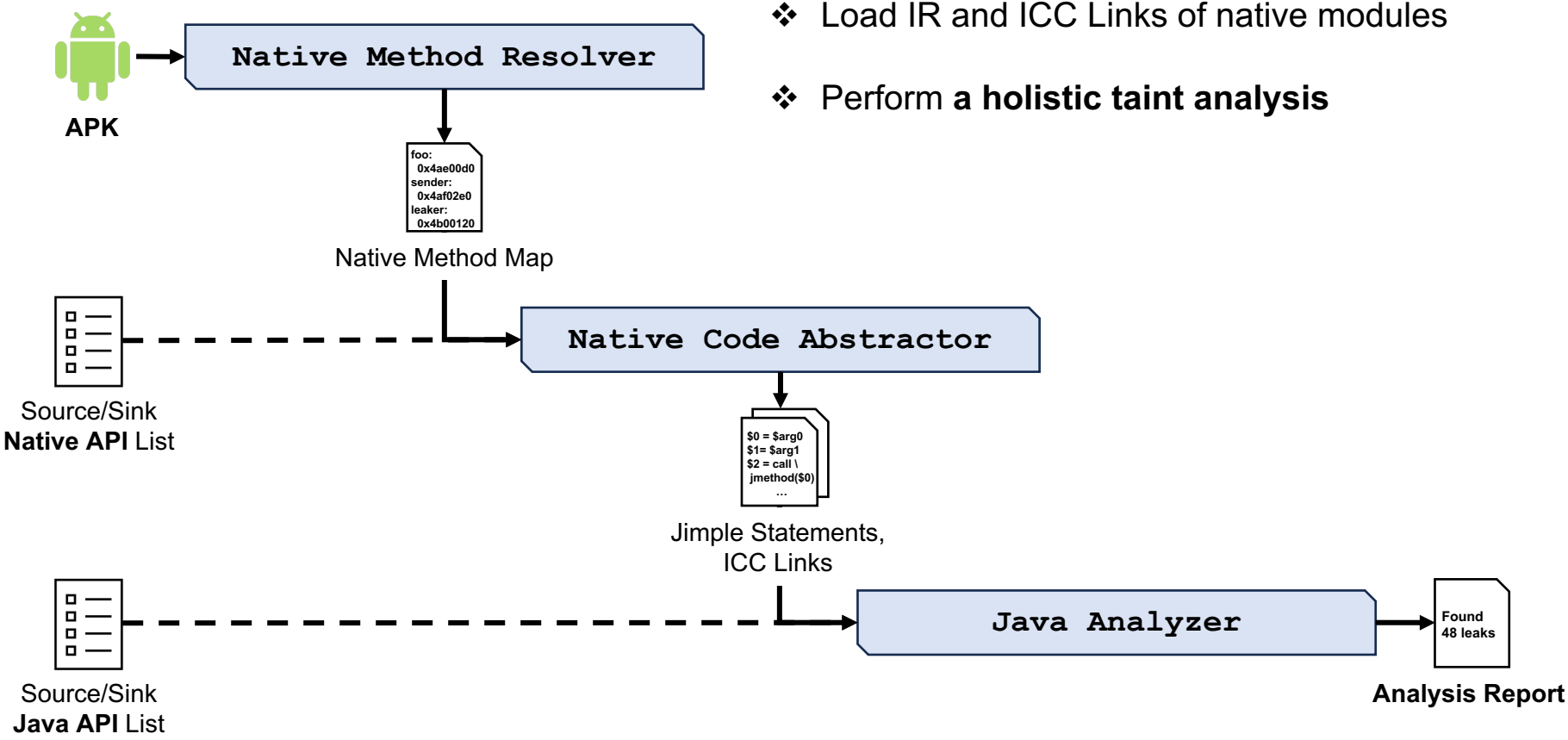


Overview of DryJIN - Java Analyzer



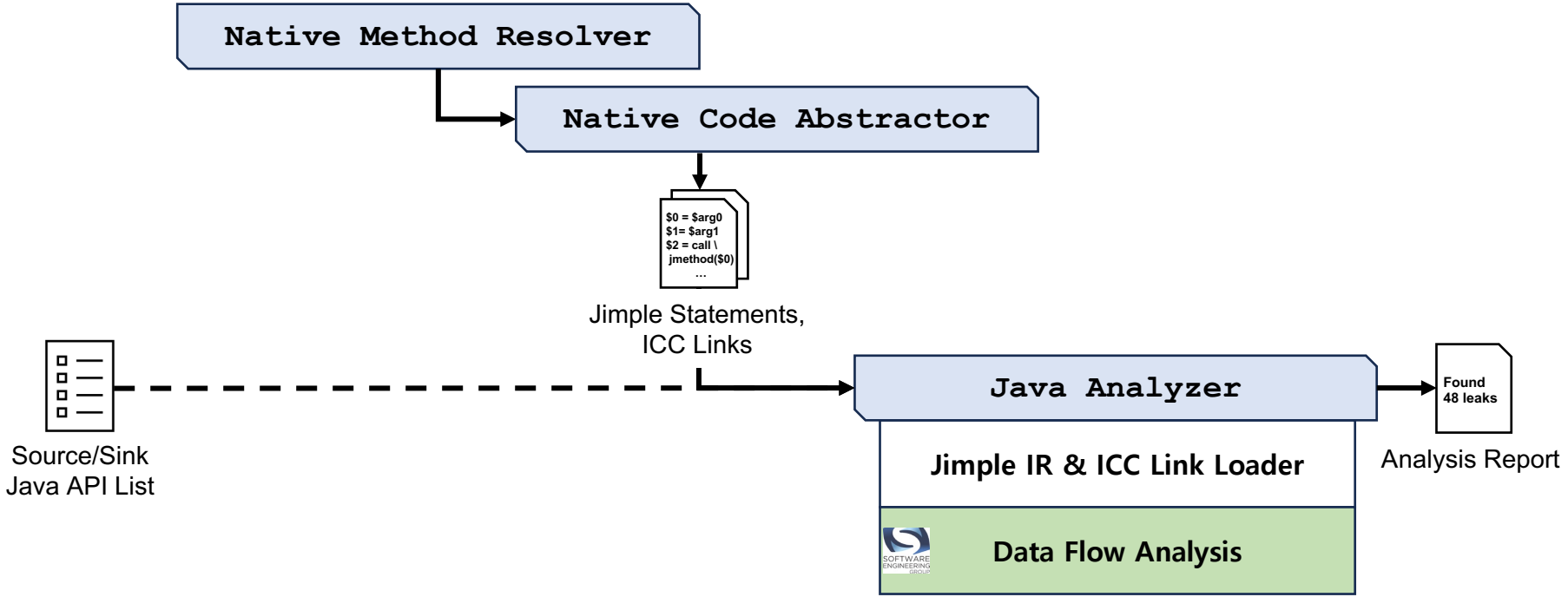
- Load **IR** and **ICC Links** of native modules

Overview of DryJIN - Java Analyzer



- Load IR and ICC Links of native modules
- Perform **a holistic taint analysis**

Overview of DryJIN - Java Analyzer



Research Questions

- RQ 1. How does DryJIN perform on **benchmark test suites**?
- RQ 2. Can DryJIN be used for analyzing **real-world apps**?
- RQ 3. When and why did DryJIN encounter **difficulties** in analyzing apps?
- Comparison Tools: **Argus-SAF**, **JuCify**

RQ 1. How does DryJIN perform on benchmark test suites?

- **Additional benchmarks** to handle native flows completely.

RQ 1. How does DryJIN perform on benchmark test suites?

- **Additional benchmarks** to handle native flows completely.
- **Other tools:** effective results only for its own benchmark.

Test Suites			Argus-SAF		JuCify		DryJIN	
Category	**Benchmarks**	**Leaks**	**Precision (%)**	**Recall (%)**	**Precision (%)**	**Recall (%)**	**Precision (%)**	**Recall (%)**
Argus-SAF	23	20	**100**	**100**	100	11.8	100	100
JuCify	11	9	100	0	**81.8**	**100**	100	100
DroidBench	5	5	100	20	100	40	100	100
DryJIN	12	12	100	8.3	100	16.7	100	100
Total	51	46	100	32.1	95.5	42.1	100	100

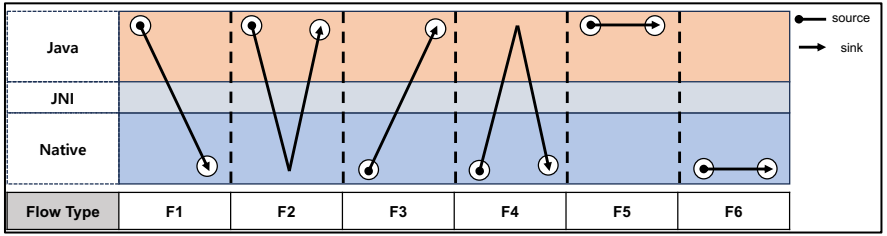
RQ 1. How does DryJIN perform on benchmark test suites?

- **Additional benchmarks** to handle native flows completely.
- **Other tools:** effective results only for its own benchmark.
- **DryJIN:** outperformed results for all benchmarks.

Test Suites			Argus-SAF		JuCify		DryJIN	
Category	**Benchmarks**	**Leaks**	**Precision (%)**	**Recall (%)**	**Precision (%)**	**Recall (%)**	**Precision (%)**	**Recall (%)**
Argus-SAF	23	20	**100**	**100**	100	11.8	100	100
JuCify	11	9	100	0	**81.8**	**100**	100	100
DroidBench	5	5	100	20	100	40	100	100
DryJIN	12	12	100	8.3	100	16.7	100	100
Total	51	46	100	32.1	95.5	42.1	100	100

RQ 2. Can DryJIN be used for analyzing real-world apps?

❖ **DryJIN:** 268 leak cases in the wild without java-only leak (i.e., F5).



DryJIN	Malware		Benign-ware	
	2021	2022	2021	2022
# of Apps Used	50,480	54,254	52,481	12,073
# of Detected Apps Leaking Information	3,865	4,635	7,947	3,205
(Java > Native) F1 Leak	**85**	**94**	**34**	**5**
(Java > Java) F2 Leak	**4**	**6**	0	0
(Native > Java) F3 Leak	**2**	**5**	0	**1**
(Native > Native) F4 Leak	0	0	0	0
(Java > Java) F5 Leak	3,763	4,512	7,905	3,198
(Native > Native) F6 Leak	**9**	**14**	**8**	**1**

RQ 2. Can DryJIN be used for analyzing real-world apps?

- **JuCify:** 2 leak cases as java-to-java leak through native flow (i.e, F2).
- **Argus-SAF:** misses for all cases.

	Malware		Benign-ware	
	2021	2022	2021	2022
# of Apps Used	92	106	42	7
Argus-SAF				
F1 Leak	0	0	0	0
F2 Leak	0	0	0	0
F3 Leak	0	0	0	0
F4 Leak	0	0	0	0
F5 Leak	0	0	0	0
F6 Leak	0	0	0	0
JuCify				
F1 Leak	0	0	0	0
F2 Leak	0	**2**	0	0
F3 Leak	0	0	0	0
F4 Leak	0	0	0	0
F5 Leak	0	0	0	0
F6 Leak	0	0	0	0

Case Study: ‘libgoogleapi.so’

- Loading **a native library**: ‘libgoogleapi.so’.

```
void __fastcall __noreturn Java_com_android_googleapi_tzg_ApiServices_start(JNIEnv *a1, jobject a2)
{
  char *cwd_len; // r0
  char cwd; // [sp+10h] [bp-80h]

  env = a1;
  obj = (int)a2;
  native_clazz = ((int (*)(void))(*a1)->GetObjectClass)();
  j_memset(&cwd, 0, 100);
  cwd_len = j_getcwd((int)&cwd, 100);
  j___android_log_print(3, "setting", "%s, %s", cwd_len, &cwd);
  imei = (int)getInfoByMethodName((int)"getIMEI");
  j___android_log_print(3, "setting", "imei-%s", imei);
  server_ip = getInfoByMethodName((int)"getServerip");
  server_port = *(_DWORD *)getIntByMethodName("getServerPort");
  uid[0] = getInfoByMethodName((int)"getUID");
  j___android_log_print(3, "setting", "server:-%s:%d:%s", server_ip, server_port, uid[0]);
  while ( 1 )
  {
    start(imei, (int (__fastcall *)(int, int, int))processor);
    j_sleep(10);
  }
}
```

Case Study: ‘libgoogleapi.so’

- Loading **a native library**: ‘libgoogleapi.so’.
- Calling **a native method** after launching the app.

```
void __fastcall __noreturn Java_com_android_googleapi_tzg_ApiServices_start(JNIEnv *a1, jobject a2)
{
  char *cwd_len; // r0
  char cwd; // [sp+10h] [bp-80h]

  env = a1;
  obj = (int)a2;
  native_clazz = ((int (*)(void))(*a1)->GetObjectClass)();
  j_memset(&cwd, 0, 100);
  cwd_len = j_getcwd((int)&cwd, 100);
  j___android_log_print(3, "setting", "%s, %s", cwd_len, &cwd);
  imei = (int)getInfoByMethodName((int)"getIMEI");
  j___android_log_print(3, "setting", "imei-%s", imei);
  server_ip = getInfoByMethodName((int)"getServerip");
  server_port = *(_DWORD *)getIntByMethodName("getServerPort");
  uid[0] = getInfoByMethodName((int)"getUID");
  j___android_log_print(3, "setting", "server:-%s:%d:%s", server_ip, server_port, uid[0]);
  while ( 1 )
  {
    start(imei, (int (__fastcall *)(int, int, int))processor);
    j_sleep(10);
  }
}
```

Case Study: ‘libgoogleapi.so’

- Loading **a native library**: ‘libgoogleapi.so’.
- Calling **a native method** after launching the app.
- Invoking **a java source API** to obtain IMEI.

```
public String getIMEI() {
    String v0;  // return ci.b.getSystemService("phone").getDeviceId();
    try {
        v0 = ci.b();  // return ci.b.getSystemService("phone").getDeviceId();
    }
    catch(Exception v1) {
        Log.v("", "", ((Throwable)v1));
    }

    return v0;
}
```

Source (Java):
Call java method

```
1  void __fastcall __noreturn Java_com_android_googleapi_tzg_ApiServices_start(JNIEnv *a1, jobject a2)
2  {
3    char *cwd_len; // r0
4    char cwd; // [sp+10h] [bp-80h]
5
6    env = a1;
7    obj = (int)a2;
8    native_clazz = ((int (*)(void))(*a1)->GetObjectClass)();
9    j_memset(&cwd, 0, 100);
10   cwd_len = j_getcwd((int)&cwd, 100);
11   j___android_log_print(3, "setting", "%s, %s", cwd_len, &cwd);
12   imei = (int)getInfoByMethodName((int)"getIMEI");
13   j___android_log_print(3, "setting", "imei-%s", imei);
14   server_ip = getInfoByMethodName((int)"getServerip");
15   server_port = *(_DWORD *)getIntByMethodName("getServerPort");
16   uid[0] = getInfoByMethodName((int)"getUID");
17   j___android_log_print(3, "setting", "server:-%s:%d:%s", server_ip, server_port, uid[0]);
18   while ( 1 )
19   {
20     start(imei, (int (__fastcall *)(int, int, int))processor);
21     j_sleep(10);
22   }
23 }
```

Case Study: ‘libgoogleapi.so’

- Loading **a native library**: ‘libgoogleapi.so’.
- Calling **a native method** after launching the app.
- Invoking **a java source API** to obtain IMEI.
- Starting a thread to **log** and **send** it.

```
void __fastcall __noreturn Java_com_android_googleapi_tzg_ApiServices_start(JNIEnv *a1, jobject a2)
{
  char *cwd_len; // r0
  char cwd; // [sp+10h] [bp-80h]

  env = a1;
  obj = (int)a2;
  native_clazz = ((int (*)(void))(*a1)->GetObjectClass)();
  j_memset(&cwd, 0, 100);
  cwd_len = j_getcwd((int)&cwd, 100);
  j___android_log_print(3, "setting", "%s, %s", cwd_len, &cwd);
  imei = (int)getInfoByMethodName((int)"getIMEI");
  j___android_log_print(3, "setting", "imei:%s", imei);
  server_ip = getInfoByMethodName((int)"getServerip");
  server_port = *(_DWORD *)getIntByMethodName("getServerPort");
  uid[0] = getInfoByMethodName((int)"getUID");
  j___android_log_print(3, "setting", "server:-%s:%d:%s", server_ip, server_port, uid[0]);
  while ( 1 )
  {
    start(imei, (int (__fastcall *)(int, int, int))processor);
    j_sleep(10);
  }
}
```

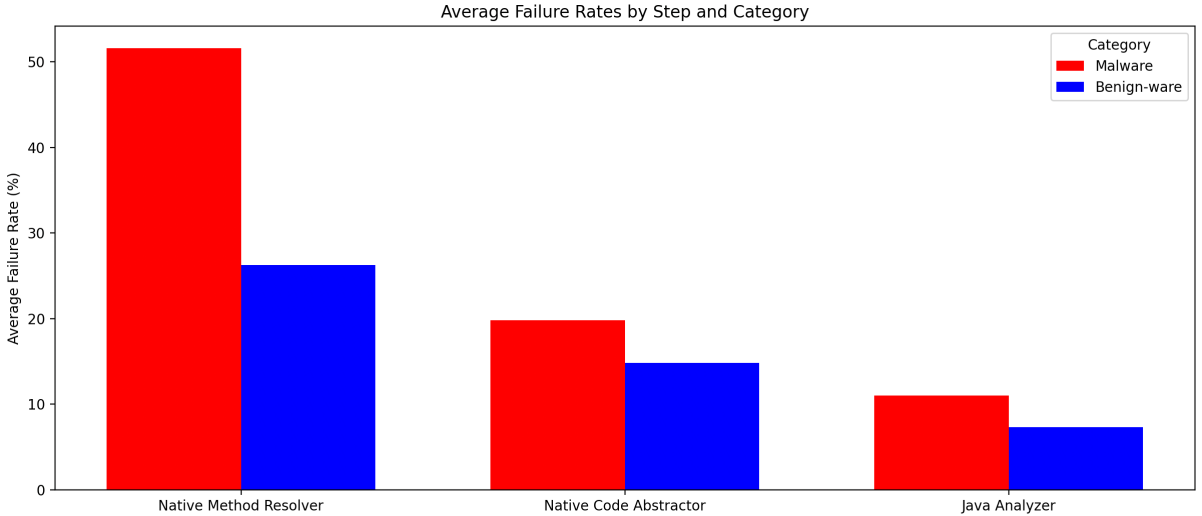
```
public String getIMEI() {
    String v0;  // return ci.b.getSystemService("phone").getDeviceId();
    try {
        v0 = ci.b();  // return ci.b.getSystemService("phone").getDeviceId();
    }
    catch(Exception v1) {
        Log.v("", "", ((Throwable)v1));
    }

    return v0;
}
```

Source (Java):
Call java method

```
  while ( 1 )
  {
    j_memset(&cmdline, 0, 12);
    j_memset(&buf, 0, 1024);
    cmd_len = j_read(sock_fd, &buf, 1024);
    if ( cmd_len <= 7 )
      break;
    cmdline = buf;
    dword_5C27C = buf[4];
    if ( cmd_len != 8 )
    {
      size_ = size;
      buf2 = j_malloc(size + 1);
      buf2_size = size_;
      dword_5C280 = (int)buf2;
      buf2[size_] = 0;
      j_memcpy(buf2, &buf2_, buf2_size);
    }
    info = (_BYTE *)processor_func(cmdline, dword_5C27C, dword_5C280);
    if ( !info )
    {
      info = j_malloc(5);
      *(_DWORD *)info = 1819047278;
      info[4] = 0;
    }
    j___android_log_print(3, "setting", "bytes:%s", info);
    size_ = j_strlen(info);
    v15 = j_write(sock_fd, &size_, 4);
    if ( v15 < 0 )
    {
      j___android_log_print(3, "setting", "s
      return 0;
    }
    if ( j_write(sock_fd, info, size_) < 0 )
      return 0;
    j_free(info);
  }
```

Sink (native):
Log print
C&C write

RQ 3. When and why did DryJIN encounter difficulties in analyzing apps?



Native Method Resolver

- Loading a native library
- Finding a native method

Native Code Abstractor

- Path explosion
- Internal error of symbolic execution engine

Java Analyzer

- Loading a DEX code
- Generating a java call graph

Summary

- Privacy leaks in Android are common.
- Current solutions lack data flow tracking in native modules.
- Comprehensive information flow tracing with native APIs in Android.
- Successfully detect 268 real-world information leaks.
- Planing to Address further challenges by modeling well-known native libraries.

Thank you

[Open Source]

DryJIN GitHub Repository
https://github.com/ssu-csec/DryJIN (Publicly available soon!)

Minseong Choi
E-mail : liberty@unist.ac.kr

Yubin Im
E-mail : th8548@soongsil.ac.kr

Steve Ko
E-mail : steveyko@sfu.ca

Yonghwi Kwon
E-mail : yongkwon@umd.edu

Yuseok Jeon
E-mail : ysjeon@unist.ac.kr

Haehyun Cho
E-mail : haehyun@ssu.ac.kr