DryJIN: Detecting Information Leaks in Android Applications

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

[1] Department of Computer Science and Engineering, UNIST, Ulsan, South Korea
{liberty,ysjeon}@unist.ac.kr
[2] Graduate School of Software, Soongsil University, Seoul, South Korea
th8548@soongsil.ac.kr, haehyun@ssu.ac.kr
[3] School of Computing Science, Simon Fraser University, Burnaby, Canada
steveyko@sfu.ca
[4] Department of Electrical and Computer Engineering, University of Maryland, Maryland, USA
yongkwon@umd.edu

Abstract. Android devices, handling sensitive data like call records and text messages, are prone to privacy breaches. Existing information flow tracking systems face difficulties in detecting these breaches due to two main challenges: the multi-layered Android platform using different programming languages (Java and C/C++), and the complex, event-driven execution flow of Android apps that complicates tracking, especially across these language barriers. Our system, DryJIN, addresses this by effectively tracking information flow within and across both Java and native modules. Utilizing symbolic execution for native code data flows and integrating it with Java data flows, DryJIN enhances existing static analysis techniques (Argus-SAF, JuCify, and FlowDroid) to cover previously unaddressed information flow patterns. We validated DryJIN's effectiveness through a comprehensive evaluation on over 168k apps, including malware and real-world apps, demonstrating its superiority over current state-of-the-art methods.

Keywords: System Security · Mobile Security · Static Analysis.

1 Introduction

Mobile devices handle a large amount of sensitive information, such as users' privacy-sensitive data (*e.g.*, pictures, SMS messages, financial information, medical records, etc.) [38], imposing a substantial privacy and security threat [39]. Specifically, sensitive information can be leaked through malicious or benign software exploited by malicious attackers. Over the years, we have witnessed devastating consequences of information leaks such as national-wide high-profile data breaches [6, 16, 35]. Despite various tracking techniques proposed for Android [6, 16, 20, 29, 30, 34, 35, 39], effectively detecting leaks remains a challenge.

Table 1. Comparison of sensitive information flows that can be detected by DryJIN and by other related work. Data flow types (F1–F6) are illustrated in Figure 1.

Source	Proxy	Sink	FlowDroid [6]	Argus-SAF [34]	JuCify [30]	DryJIN
Java	-	Java (F5)	✓	✓	✓	✓
	Native	Java (F2)	✗	✓	✓	✓
	Native	Native (F1)	✗	✓	✗	✓
Native	-	Native (F6)	✗	✗	✗	✓
	Java	Native (F4)	✗	✗	✗	✓
	Java	Java (F3)	✗	✗	✗	✓

Taint analysis, tracking information from a *source* (where sensitive information is created) to a *sink* (where it can leak to adversaries) [6, 14], comes in two forms: dynamic and static. *Dynamic* taint analysis [33, 36, 37] examines runtime but faces limitations from input quality and *coverage issues* [17], making worse by evasive techniques [2, 19]. *Static* taint analysis [6, 28, 34, 35], free from these issues, favored for tasks like malware detection and information leak detection. However, Android's complexity poses unique challenges to static analysis [31]:

1. Cross-language Modules: Android's modules implement by Java and C/C++, complicating data flow tracking across languages [24]. Most static analysis methods [6, 16, 30, 34, 35] struggle with inter-language module data flows.

2. Event-driven Execution: Android's event-driven model, using intents [10] and broadcast receivers [8], invokes handlers by the system [12]. While existing tools like FlowDroid [6], Argus-SAF[1] [34], and JuCify [30] address these flows, they often overlook critical ones (*e.g.*, types F3, F4, and F6), leading to undetected leaks in real-world apps [26], as shown in Table 1.

In this paper, we present DryJIN, a system that comprehensively tracks all types of information flows in Android, as depicted in Figure 1. Specifically, we first obtain data flows from the native code (*i.e.*, implemented by C/C++) by using a symbolic execution engine, angr [31], and collecting instructions that are dependent on sensitive information. These processes are subsequently converted into Jimple IR using the SOOT framework [21]. We also create Jimple IR for Java modules and integrate them. Finally, we employ Flowdroid's [6] data flow analysis to the unified Jimple IR from the native and Java modules.

We evaluated DryJIN using three datasets: (1) publicly available benchmarks [26, 30, 35], (2) 104,734 real-world malware obtained from VirusShare [15], and (3) 64,554 benign apps from AndroZoo [4]. The evaluation results using the benchmarks show that DryJIN outperforms state-of-the-art static taint analysis techniques. DryJIN detects all 46 true-positive cases, whereas Arugs-SAF [34] and JuCify [30] identify only 22 and 15, respectively. Moreover, we use DryJIN to detect information leaks on real-world malware and benign datasets collected in the wild from 2021 to 2022. Our results demonstrate DryJIN's advantage over existing techniques by identifying 268 cases of sensitive information leak among

[1] Originally JN-SAF, now integrated with Amandroid and known as Argus-SAF.

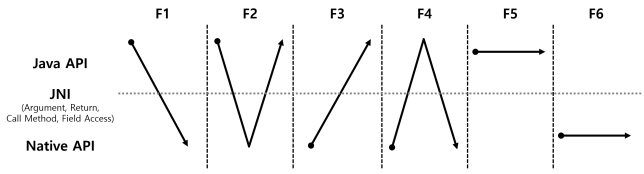

Fig. 1. Possible information flows in an Android app. Except for the type F5, the other types of information flows go through the native code [36].

247 real-world apps. In contrast, the state-of-the-art techniques like Argus-SAF failed to detect any, while JuCify detects only 2 leak of them.

Our work develops DryJIN, a system that leverages static taint analysis to detect sensitive information flows in Android apps, including those involving cross-language modules. Our approach integrates data flows from both native and Java code, using tools like angr and SOOT, and analyzes them with Flowdroid. Evaluation results demonstrate DryJIN's performance in detecting information leaks over existing techniques, proving effective across extensive real-world datasets. Our main contributions are summarized as follows.

1. We propose DryJIN that comprehensively detects information flows in Android, focusing on data flows between Java and native modules. All code and data will be available for future research at `https://github.com/ssu-csec/DryJIN`.
2. We measure and identify weaknesses of state-of-the-art static data flow tracking techniques for six information flow types between Java and native modules.
3. We revisit and improve information leak detection test suites for extensive coverage of information flows across Java and native modules and evaluate DryJIN using these suites and real-world datasets of 104,734 malicious and 64,554 benign apps from VirusShare and AndroZoo, respectively.

The paper is structured as follows: Section 2 covers the background on Android apps, Section 3 addresses the problems of existing approaches, while Section 4 provides a design overview of DryJIN. Section 5 discusses the evaluation results, Section 6 covers key discussion points, and Section 7 concludes the paper.

2 Background

In this section, we outline key concepts and information flow types for DryJIN.
Cross-language Support of Android. In Android development, Java/Kotlin and C/C++ components interact via the **J**ava **N**ative **I**nterface (JNI). This interface, supported by the Native Development Kit (NDK) [9], allows Java modules to execute native code, crucial for high-performance tasks and hardware-specific functions. Functions in native modules can exchange data with Java components through JNI, ensuring efficient interoperability between these different languages. However, this integration poses security challenges. Data transfer between Java and native code can potentially lead to sensitive data leaks in the

Table 2. The usage of native code in Android apps (h/ = have, i/ = invoke).

		# of Apps	h/ native library	i/ native method
VirusShare (Malicious apps)	2020	104,954	89,765 (85.5%)	47,267 (45.0%)
	2021	110,549	93,011 (84.1%)	50,480 (45.7%)
	2022	71,500	59,043 (82.6%)	54,254 (75.9%)
	Total	287,003	241,819 (84.3%)	152,001 (53.0%)
AndroZoo (Benign apps)	2020	174,737	93,988 (53.8%)	76,014 (43.5%)
	2021	93,126	58,472 (62.8%)	52,481 (56.4%)
	2022	17,363	12,866 (74.1%)	12,073 (69.5%)
	Total	285,226	165,326 (58.0%)	140,568 (49.3%)
	Total	**572,229**	**407,145 (71.2%)**	**292,569 (51.1%)**

alternate language module. Thus, closely monitoring JNI communication is key to safeguarding data in Android apps.

Native Code Usage in Android Apps. To understand how commonly is the native code used in real-world Android apps, we study apps interacting with native code. We collect two kinds of datasets. First, we download 285,226 benign apps listed in AndroZoo [4] from Google Play [23]. Second, we collect 287,003 malware apps from VirusShare [15]. From the datasets, we inspect every apps including any "ELF" files, which are native library binaries. After then, we count declarations and invocations related to the identified native libraries. Table 2 shows the result of our study. First, a significant number of Android apps contain native libraries: 71.2% in total, 84.3% from malware apps, and 58.0% from benign apps. Second, we observe that native code is more frequently used in malicious apps than benign ones. Third, we note a constant increase in the number of benign apps leveraging native code [1]. Fourth, we find 51.1% of the apps call a native method at least once. This reveals a 20.1% gap between apps that include native libraries and invoke native methods. Our manual inspection shows that the majority of them are heavily obfuscated apps, where static analysis techniques [5, 6, 30, 34] failed to identify the native method invocations.

Information Flows in Android Apps. In Android apps, the flow of information between different languages modules (Java/Kotlin and C/C++) is crucial. Taint analysis is designed to track significant information flows, starting from a source API to a sink API. For example, sensitive data such as the IMEI (International Mobile Equipment Identity) is obtained via a source API and processed in both Java and C/C++ modules before being sent out through a sink like a network socket API. Data flows in Android often include argument passing in functions like 'memcpy' instead of assignment directly and are complicated by event-driven mechanisms such as JNI functions, making tracking these flows challenging. Figure 1 depicts these information flows within an Android app. Unfortunately, missing even a single information flow leads to a failure in detecting sensitive information leak. To comprehensively capture information flow between heterogeneous modules, it is critical to precisely model information flow between them. However, previous approaches have limited analysis scopes (which we demonstrate in Section 5), and thus, missing sensitive data leaks.

3 Problem Statement of Existing Approaches

In this section, we discuss the challenges of detecting information leaks in Android, focusing on the limitations of existing techniques.

Existing Taint Analysis Approaches. In recent years, various taint analysis techniques like FlowDroid [6] and Amandroid [35] have been developed for Android app leaks, targeting Java/Kotlin apps with flow and context sensitivity. FlowDroid manages transitions with a dummy main model, whereas Amandroid uses environment modeling for data flows. Other studies [3, 27, 30, 33, 34] apply static analysis to cross-language (*i.e.*, C/C++) Android apps. JN-SAF [34] leads this field by modeling JNI functions and native activities for JNI/NDK-aware analysis. Unlike JN-SAF, JuCify [30] unifies these by abstracting native code into Java-level Jimple IR for integrated analysis. Dynamic analysis tools like TaintART [33] and NDroid [36] assess JNI-related flows in native code. TaintART targets OAT-compiled code, missing external libraries, while NDroid includes native libraries. However, both are bound to real execution paths, unlike static analysis's broader coverage. In this paper, we will focus solely on static analysis.

Limitations of Existing Approaches. FlowDroid [6], Argus-SAF [34], and JuCify [30] are the three state-of-the-art information leak detection techniques in Android. FlowDroid [6] handles various event handlers in the Android framework and conducts context, flow, field, and object-sensitive data flow analysis. However, it does not extend its analysis to native code and JNI functions. Argus-SAF [34] is an inter-language static analysis framework utilizing JNI function and NDK library models. However, it fails to capture data flows when Java methods, that are neither sources nor sinks, are invoked from native functions. JuCify [30] is a unified framework leveraging Jimple IR for native method data flow analysis. However it encounters three limitations: (1) it does not track data flows through native APIs, (2) it only reconstructs behaviors of native methods directly invoked via Java calls, restricting its coverage to only F2 in Figure 1, and (3) it struggles with inferring the return or parameter values in data flows, due to the opaque prediction approach.

Lack of Empirical Evaluation and Comparison. There is a lack of benchmarks that can provide a fair comparison among these techniques. Each static analysis framework is effective with its own benchmarks, while not for the others.

4 Our Approach

In this paper, we propose DryJIN that handles the limitations of the existing approaches to comprehensively detect the diverse types of information leaks. The design goal of DryJIN is to accurately detect all data flow types shown in Figure 1, by synthesizing the strengths and mitigating the weaknesses of Argus-SAF [34], JuCify [30], and FlowDroid [6]. DryJIN targets data flows overlooked by previous techniques, especially those originating from native sources or terminating at native sink APIs. It models control and data flow of native code via JNI/NDK functions as in Argus-SAF [34] with JuCify's [30] unified IR approach.

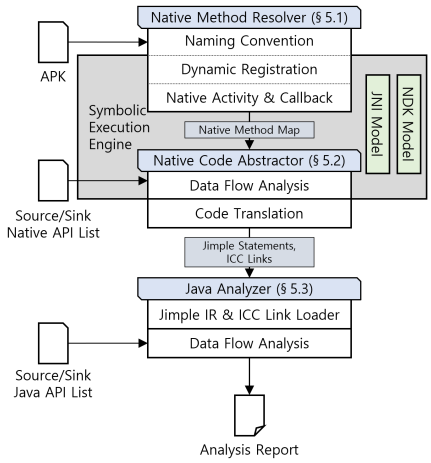

Fig. 2. Overview of DryJIN analysis process.

4.1 Resolving Native Methods

This task involves mapping method signatures to function addresses of native methods in libraries, initiating code abstraction will be discussed in Section 4.2. The Android runtime links native methods to Java code either by following the JNI naming convention or through the "`RegisterNatives`" JNI function [25].

DryJIN identifies methods following the JNI naming convention by searching for functions with names starting with "`Java_`". Then, DryJIN resolves the function names according to the JNI naming convention to derive the associated class names, Java method names, and parameters. These enable locating the corresponding method definitions in the 'classes.dex' file compiled code from Java source codes. Lastly, DryJIN records the native methods' addresses in native libraries for subsequent steps (Section 4.2). For native methods registered through the `RegisterNatives` function, typically invoked in the `JNI_OnLoad` method, DryJIN leverages the provided Java class object and an array containing Java method names, signatures, and native function pointers [11]. Key to this process is extracting the Java method signature and native function pointer from the array, which enables linking each native method to its Java counterpart.

In Android, *native activity* facilitates app development in native code for Android API access. The default entry point of native activity is `android_main` or `ANativeActivity_onCreate` [34]. Identifying sensitive information flows, especially through Inter-Component Communication (ICC) in Android's event-driven framework, is also essential. To tackle method linking and native activity, we employ JN-SAF's [34] symbolic execution, which models JNI functions and native activity for effective resolution of native methods. DryJIN captures the addresses of these methods to analyze native code data flows.

4.2 Abstracting Data Flows in Native Code

After identifying the addresses of native methods (including native activities), DryJIN starts the symbolic execution from each address to analyze data flows in the native code. DryJIN then abstracts these data flows into Jimple IR. For this step, we enhance two approaches proposed by Argus-SAF [34] and JuCify [30] with angr [31] for the symbolic execution engine.

We utilize Argus-SAF's [34] JNI functions and NDK library model for NDK/JNI-aware data flow analysis within symbolic execution. However, Argus-SAF's model *lacks* support for many native APIs as sources and sinks, only recognizing the `android_log_print` API. To address this, we manually added native sink and source APIs from the NDK API list [7] including libc and system call. Additionally, we enhanced JuCify [30] to transform Jimple IR, capturing abstracted data flows in native code. DryJIN annotates data (*e.g.*, symbolic variable) with types and originated from JNI functions and native source APIs, enabling precise tracking during symbolic execution. This overcomes JuCify's limitations in inferring method parameters. Furthermore, DryJIN captures ICC links from argument data of invoking ICC-related methods, filling gaps in handling ICC overlooked by JuCify and pass into FlowDroid as will be discussed in Section 4.3.

4.3 Detecting Sensitive Data Leaks

To detect sensitive information leaks, DryJIN employs FlowDroid [6] with the Jimple IR statements and ICC links generated from the analysis result of data flows. To this end, we modify FlowDroid as follows. First, FlowDroid uses the `dummy main method` to generate a call graph, emulating the lifecycle of Java components excepting for native activities. Therefore, DryJIN's FlowDroid adds nodes for callbacks of native activities and link edges from the `dummy main method`. By doing so, FlowDroid performs data flow analysis including native activities. Next, to handle the invocation of native APIs, DryJIN creates a new class called "`DummyNative`" contains native APIs as its methods. Lastly, DryJIN's FlowDroid loads the Jimple IR statements into native methods and the ICC links to identify the data passing through ICC-related method invocations. Consequently, the output of FlowDroid includes sensitive information leaks from a source API to a sink API located in both native and Java modules.

5 Evaluation

We evaluate DryJIN on the following 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?

Implementation. We implement DryJIN according to the design as discussed in Section 4 by employing Argus-SAF [34], JuCify [30], and FlowDroid [6]. In

Table 3. Evaluation results on benchmark test suites. Blue represents true positives and negatives, red indicates false negatives, and green denotes false positives.

Category	(Information Flow Type in Figure 1) Benchmark (# of Benchmarks = 51)	# of Leaks (# of Total leaks = 46)	Tool (# of Leaks detected) Argus-SAF [34]	JuCify [30]	DryJIN
Argus-SAF [34]	(F1) icc_javatonative.apk	1	1	0	1
	(F2) icc_nativetojava.apk	1	1	0	1
	(F1) native_complexdata.apk	1	1	0	1
	(N/A) native_complexdata_stringop.apk	0	0	0	0
	(F1) native_dynamic_register_multiple.apk	1	1	0	1
	(F2) native_heap_modify.apk	1	1	0	1
	(F1) native_leak.apk	1	1	0	1
	(F1) native_leak_array.apk	1	1	0	1
	(F1) native_leak_dynamic_register.apk	1	1	0	1
	(N/A) native_method_overloading.apk	0	0	0	0
	(F1) native_multiple_interactions.apk	1	1	0	1
	(F1) native_multiple_libraries.apk	1	1	0	1
	(N/A) native_noleak.apk	0	0	0	0
	(N/A) native_noleak_array.apk	0	0	0	0
	(N/A) native_nosource.apk	0	0	0	0
	(F1) native_pure.apk	1	1	0	1
	(F1) native_pure_direct.apk	1	1	0	1
	(F1) native_pure_direct_customized.apk	1	1	0	1
	(F2) native_set_field_from_arg.apk	2	2	1	2
	(F2) native_set_field_from_arg_field.apk	2	2	0	2
	(F2) native_set_field_from_native.apk	2	2	0	2
	(F2) native_source.apk	1	1	1	1
	(N/A) native_source_clean.apk	0	0	0	0
Jucify [30]	(F2) delegation_imei.apk	1	0	1	1
	(F2) delegation_proxy.apk	1	0	4	1
	(F2) getter_imei.apk	1	0	1	1
	(F2) getter_imei_deep.apk	1	0	1	1
	(F2) getter_leaker.apk	1	0	2	1
	(F2) getter_proxy_leaker.apk	1	0	2	1
	(N/A) getter_string.apk	0	0	1	0
	(F2) leaker_imei.apk	1	0	2	1
	(N/A) leaker_string.apk	0	0	2	0
	(F2) proxy.apk	1	0	1	1
	(F2) proxy_double.apk	1	0	1	1
DroidBench [26]	(F2) JavaIDFunction.apk	1	0	0	1
	(F1) NativeIDFunction.apk	1	0	1	1
	(F2) SinkInNativeCode.apk	1	0	0	1
	(F1) SinkInNativeLibCode.apk	1	0	0	1
	(F2) SourceInNativeCode.apk	1	1	1	1
Our Benchmarks	(F2) ArgToSetField.apk	1	0	0	1
	(F2) CopyRegion.apk	1	0	1	1
	(F2) GetFieldToCallMethod.apk	1	0	1	1
	(F2) InterNativeMethod.apk	1	0	0	1
	(F4) JavaProxy.apk	1	0	0	1
	(F1) JavaToNative.apk	1	1	0	1
	(F2) NativeProxy.apk	1	0	0	1
	(F3) NativeSourceToCallMethod.apk	1	0	0	1
	(F3) NativeSourceToSetField.apk	1	0	0	1
	(F3) NativeToJava.apk	1	0	0	1
	(F6) NativeToNative.apk	1	0	0	1
	(F3) SourceInNativeLibCode.apk	1	0	0	1
		# of True-positives	**22**	**15**	**46**
		# of False-positives	**0**	**9**	**0**
		# of False-negatives	**24**	**31**	**0**

our implementation, we make enhancements totaling 2,661 SLoC in Python for Argus-SAF and 603 SLoC for JuCify, along with 890 SLoC in Java for FlowDroid.

Experimental Setup. Analysis run on Intel Xeon CPU server with 256 GB RAM, enforcing a one-hour limit per app (30 mins each for symbolic execution and FlowDroid) and a 32 GB memory cap. Exceeding these limits led DryJIN to halt the current step but continue with the next, using accumulated results.

Dataset. We evaluate DryJIN using three datasets: (Benchmarks) open test suites from previous work like DroidBench [26], Argus-SAF [34], and JuCify [30], along with 12 new benchmarks we developed; (Malware) 104,734 malicious apps from 2021 and 2022, sourced from VirusShare [15]; and (Benign Apps) 64,554 benign apps from the same period, collected from AndroZoo [4].

Table 4. Evaluation results on real-world apps. Information flow patterns are in Figure 1. An app can obtain multiple flow types of information leaks.

	Malware		Benign-ware	
	2021	2022	2021	2022
DryJIN				
# of Apps Used	50,480	54,254	52,481	12,073
# of Apps Successfully Analyzed	7,984	10,419	24,727	6,757
(%)	(16%)	(19%)	(47%)	(56%)
# of Detected Apps Leaking Information	3,865	4,635	7,947	3,205
(# of Apps generating the type F1 leak)	85	94	34	5
(# of Apps generating the type F2 leak)	4	6	0	0
(# of Apps generating the type F3 leak)	2	5	0	1
(# of Apps generating the type F4 leak)	0	0	0	0
(# of Apps generating the type F5 leak)	3,763	4,512	7.905	3,198
(# of Apps generating the type F6 leak)	9	14	8	1
Argus-SAF				
# of Apps Used	92	106	42	7
(# of Apps generating the F1, F2, F3, F4, and F6 type leaks)				
# of Detected Apps Leaking Information	0	0	0	0
(# of Apps generating the type F1 leak)	0	0	0	0
(# of Apps generating the type F2 leak)	0	0	0	0
(# of Apps generating the type F3 leak)	0	0	0	0
(# of Apps generating the type F4 leak)	0	0	0	0
(# of Apps generating the type F6 leak)	0	0	0	0
JuCify				
# of Apps Used	92	106	42	7
(# of Apps generating the F1, F2, F3, F4, and F6 type leaks)				
# of Detected Apps Leaking Information	0	2	0	0
(# of Apps generating the type F1 leak)	0	0	0	0
(# of Apps generating the type F2 leak)	0	2	0	0
(# of Apps generating the type F3 leak)	0	0	0	0
(# of Apps generating the type F4 leak)	0	0	0	0
(# of Apps generating the type F6 leak)	0	0	0	0

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

We utilized benchmarks consisting of 12 cases: 5 test cases illustrating information flows F1 to F6 (excluding F5, which is identifiable through Java analysis alone) and 7 cases addressing previously neglected information flows via JNI functions. Table 3 shows DryJIN's evaluation, successfully detecting leaks across 51 benchmarks with no false positives. We compared DryJIN with Argus-SAF [34] and JuCify [30] for accuracy. Argus-SAF detected leaks in its benchmarks but missed JuCify's, finding only one leak in DroidBench and our tests. JuCify found leaks in its benchmarks with nine false positives but only two in Argus-SAF's, DroidBench, and ours. DryJIN outperformed these techniques, particularly in DroidBench and our benchmarks, highlighting our design's effectiveness in identifying Android information leaks. Yet, the issue of evaluation discrepancies, as highlighted by Pauck et al. [26], persists in this field.

Answer to RQ1: DryJIN successfully detects 100% of information leaks in 51 benchmarks that cover information flows types without any false positive, 48% and 63% higher than Argus-SAF and JuCify, respectively.

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

To evaluate DryJIN against real-world Android apps, we tested 104,734 malicious and 64,554 benign apps from 2021 and 2022, that invokes native methods at least once, as listed in Table 2. Table 4 shows DryJIN identified sensitive information leaks in 19,652 of the 167,488 apps tested, categorizing them into six types as depicted in Figure 1. Among information flow patterns, F5, not involving native methods, is most common. DryJIN effectively identified leaks in all types except F4, with F3, F4, and F6 being previously overlooked. This highlights the necessity of covering all information flows for comprehensive leak detection. Argus-SAF and JuCify detected zero and two leaks, respectively, in 247 apps with leaks (excluding F5-only leaks), suggesting potential flaws in their detection capabilities (Table 4). The reason for DryJIN's outperforming results lies in its adoption of a unified IR approach and comprehensive handling of native source sink APIs. However, DryJIN analyzed only 18% of malicious and 52% of benign apps, often due to obfuscation. Further details will be discussed in Section 5.3.

Case Study. In our real-world malware analysis, DryJIN detected 23 malicious apps leaking sensitive information through the F6-type data flow. We discuss a case study where a malicious app[2] that leaks the IMEI number. The malware contains `libgoogleapi.so` library in which invokes a JNI function to obtain a device's IMEI number. Then, it writes the data on a socket to send it over the network. Note that this case cannot be detected by the previous work because such data flow type (*i.e.*, F6 type) has not been considered in previous studies. We manually reverse-engineered the malware to check the behavior and confirmed that DryJIN correctly detected the leak.

Answer to RQ2: DryJIN can cover a wide range of information flow patterns in real-world apps, including F3 and F6, previously not considered in other research efforts. In contrast, Argus-SAF fails to identify any leaks, and JuCify only detects a single pattern (F2). However, DryJIN is currently unable to analyze a significant portion of real-world apps.

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

During our evaluation, we observe that DryJIN failed in analyzing 82% of malicious apps while it successfully analyzed more than 50% of benign apps. We discuss why DryJIN failed to analyze apps in the wild.

Root Cause Analysis. To find the root causes of failure cases, we inspected the analysis processes of each step. As Table 5 shows, DryJIN failed to analyze most benign and over half of malicious apps in **Step 1** (Section 4.1), mainly due to failure in finding native methods (74%) in malicious apps. Given apps invoke a native method at least once, we suspect they employ anti-analysis or obfuscation techniques to evade detection. Our investigation into randomly selected samples revealed they were obfuscated with tools like DexProtector [22]. For benign apps,

[2] SHA256: b062e6c65c08830297c39ce054ce457d3dfd26eab7f5cb53606ca2df17938322

Table 5. Analysis results on apps failed to analyze with DryJIN at each analysis step.

	Malware		Benign-ware	
	2021	2022	2021	2022
# of Apps Used	50,480	54,254	52,481	12,073
# of Apps Failed to Analyze	42,496	43,835	27,754	5,316
# of Apps Failed in Step 1	25,983	28,072	14,224	3,070
(# of Apps failed to load a native library)	3,440	7,388	5,251	1,113
(# of Apps failed to find a native method)	20,369	19,686	4,270	1,518
(# of Apps failed to run in time)	2,174	998	4,703	439
# of Apps Failed in Step 2	11,608	9,010	9,605	1,379
(# of Apps occurred errors in angr)	6,439	1,539	6,373	434
(# of Apps occurred the path explosion)	5,169	7,471	3,232	945
# of Apps Failed in Step 3	4,905	6,753	3,925	867
(# of Apps failed in loading a dex file)	1,366	1,798	204	83
(# of Apps failed to generate a call graph)	764	782	271	100
(# of Apps failed to run in time)	2,775	4,173	3,450	684

36% halted due to timeouts from large native libraries. Android apps are easier to reverse-engineer than other binaries due to Java decompilation by tools like jadx [32], prompting extensive use of obfuscation like string encryption and dynamic code loading [13]. Malicious apps often use these techniques to avoid static analysis, making obfuscation handling crucial in app analysis [18].

In **Step 2** (Section 4.2), apps faced `angr` errors or time-outs due to the path explosion, a common issue with symbolic execution-based static analysis frameworks [31]. In **Step 3** (Section 4.3), timeouts were the main failure reason (67%) for both malicious and benign apps, often due to their large sizes exceeding our set time limits. Additionally, dex file loading failures and call graph generation issues occurred, similar to Step 1, often caused by obfuscation techniques.

Answer to RQ3: The main reasons for analysis failure of many apps are: (i) applied anti-analysis or obfuscation techniques (Step1, Step3), (ii) time-outs caused by the large size of native libraries or apps (Step1, Step3), and (iii) path explosion problem of symbolic execution (Step2).

6 Discussion

Limitations. Despite DryJIN's advancement over existing methods, it has areas for improvement in real-world app analysis. Our prototype has limited coverage of ICC methods, missing leaks via certain ICC methods like implicit intents. Future enhancements will incorporate approaches like IccTA and RAICC to model ICC links [20, 29]. Additionally, DryJIN's reliance on symbolic execution faces challenges like path explosion, common to similar techniques [30, 34]. We plan to mitigate this by modeling well-known native libraries.

Threats to Validity. To verify DryJIN's accuracy, we *manually* inspected 50 malicious apps identified for leaks with tools like Jadx and IDA Pro, validating the abstracted data flows and the actual information leaks. While not all reported leaks by DryJIN were manually verified, we hypothesized that DryJIN had produced accurate results based on our evaluation result of the test suites.

7 Conclusion

In this paper, we observe that state-of-the-art detection techniques of information leak are largely missing the data flow tracking on native modules, and thus, we propose a system that can comprehensively trace information flows in Android, covering both Java and native modules. We revisited existing benchmark test suites and our evaluation results show DryJIN's effectiveness to detect sensitive information leaks in real-world apps compared to previous approaches. In future work, we plan to mitigate inherent challenges in symbolic execution by modeling of well-known native libraries.

Acknowledgment

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korea government, Ministry of Science and ICT (MSIT) (No. 2022-0-00563, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence).

References

1. Afonso, V., Bianchi, A., Fratantonio, Y., Doupé, A., Polino, M., de Geus, P., Kruegel, C., Vigna, G.: Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: The Network and Distributed System Security Symposium. pp. 1–15 (2016)
2. Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C.: When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In: Network and Distributed Systems Security (NDSS) Symposium 2020 (2020)
3. Alam, S., Qu, Z., Riley, R., Chen, Y., Rastogi, V.: Droidnative: Automating and optimizing detection of android native code malware variants. computers & security **65**, 230–246 (2017)
4. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories. pp. 468–471. MSR '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2901739.2903508, http://doi.acm.org/10.1145/2901739.2903508
5. Anthony Desnos, Geoffroy Gueguen, S.B.: Welcome to Androguard's documentation! (Accessed: Feb 12, 2023), https://androguard.readthedocs.io/en/latest/
6. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. SIGPLAN Not. **49**(6), 259–269 (jun 2014). https://doi.org/10.1145/2666356.2594299, https://doi.org/10.1145/2666356.2594299
7. Developer, A.: Android NDK API Reference (Accessed: Feb 16, 2023), https://developer.android.com/ndk/reference
8. Developer, A.: Application Fundamentals (Accessed: Feb 16, 2023), https://developer.android.com/guide/components/fundamentals

9. Developer, A.: Concepts (Accessed: Feb 16, 2023), https://developer.android.com/ndk/guides/concepts
10. Developer, A.: Intents and Intent Filters (Accessed: Feb 16, 2023), https://developer.android.com/guide/components/intents-filters
11. Developer, A.: JNI tips (Accessed: Feb 16, 2023), https://developer.android.com/training/articles/perf-jni
12. Developer, A.: Platform Architecture (Accessed: Feb 16, 2023), https://developer.android.com/guide/platform
13. Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., Zhang, K.: Understanding android obfuscation techniques: A large-scale investigation in the wild. In: Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I. pp. 172–192. Springer (2018)
14. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: Semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. pp. 576–587 (2014)
15. Forensics, C.: VirusShare.com (Accessed: Dec 13, 2022), https://virusshare.com/
16. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS. vol. 15, p. 110 (2015)
17. Huang, C.Y., Chiu, C.H., Lin, C.H., Tzeng, H.W.: Code coverage measurement for android dynamic analysis tools. In: 2015 IEEE International Conference on Mobile Services. pp. 209–216. IEEE (2015)
18. Kan, Z., Wang, H., Wu, L., Guo, Y., Xu, G.: Deobfuscating android native binary code. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 322–323. IEEE (2019)
19. Kirat, D., Vigna, G., Kruegel, C.: Barecloud: Bare-metal analysis-based evasive malware detection. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 287–301 (2014)
20. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: Iccta: Detecting inter-component privacy leaks in android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, pp. 280–291 (2015). https://-doi.org/10.1109/ICSE.2015.48
21. Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, L.: Static analysis of android apps: A systematic literature review. Information and Software Technology **88**, 67–95 (2017)
22. licel: DexProtector (Accessed: Feb 16, 2023), https://dexprotector.com/
23. LLC, G.: Android App Google Play (Accessed: Feb 17, 2023), https://play.google.com/store/
24. Oracle: Chapter 1: Introduction (Accessed: Feb 16, 2023), https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html
25. Oracle: Chapter 4: JNI Functions (Accessed: Feb 16, 2023), https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html#registering_native_methods
26. Pauck, F., Bodden, E., Wehrheim, H.: Do android taint analysis tools keep their promises? In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 331–341 (2018)

27. Qian, C., Luo, X., Shao, Y., Chan, A.T.: On tracking information flows through jni in android applications. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 180–191. IEEE (2014)
28. Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Karonte: Detecting insecure multi-binary interactions in embedded firmware. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1544–1561 (2020). https://doi.org/10.1109/SP40000.2020.00036
29. Samhi, J., Bartel, A., Bissyandé, T.F., Klein, J.: Raicc: Revealing atypical inter-component communication in android apps. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1398–1409 (2021). https://-doi.org/10.1109/ICSE43902.2021.00126
30. Samhi, J., Gao, J., Daoudi, N., Graux, P., Hoyez, H., Sun, X., Allix, K., Bissyandé, T.F., Klein, J.: Jucify: a step towards android code unification for enhanced static analysis. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1232–1244 (2022)
31. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)
32. skylot: jadx (Accessed: Feb 17, 2023), https://github.com/skylot/jadx
33. Sun, M., Wei, T., Lui, J.C.: 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. pp. 331–342 (2016)
34. Wei, F., Lin, X., Ou, X., Chen, T., Zhang, X.: 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. p. 1137–1150. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://-doi.org/10.1145/3243734.3243835, https://doi.org/10.1145/3243734.3243835
35. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Priv. Secur. **21**(3) (apr 2018). https://doi.org/10.1145/3183575, https://doi.org/10.1145/3183575
36. Xue, L., Qian, C., Zhou, H., Luo, X., Zhou, Y., Shao, Y., Chan, A.T.: Ndroid: Toward tracking information flows across multiple android contexts. IEEE Transactions on Information Forensics and Security **14**(3), 814–828 (2019). https://-doi.org/10.1109/TIFS.2018.2866347
37. Xue, L., Zhou, Y., Chen, T., Luo, X., Gu, G.: Malton: Towards On-Device Non-Invasive mobile malware analysis for ART. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 289–306. USENIX Association, Vancouver, BC (Aug 2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xue
38. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 1043–1054 (2013)
39. Zhu, D., Jin, H., Yang, Y., Wu, D., Chen, W.: Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data. In: 2017 IEEE symposium on computers and communications (ISCC). pp. 438–443. IEEE (2017)