IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 6 January 2023, accepted 26 January 2023, date of publication 30 January 2023, date of current version 3 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3240724

RESEARCH ARTICLE

Execution Recording and Reconstruction for Detecting Information Flows in Android Apps

HIROKI INAYOSHI, SHOHEI KAKEI, AND SHOICHI SAITO
Nagoya Institute of Technology, Nagoya 466-8555, Japan
Corresponding author: Hiroki Inayoshi (h.inayoshi.849@nitech.jp)

ABSTRACT Security researchers utilize taint analyses to uncover suspicious behaviors in Android apps. Current static taint analyzers cannot handle ICC, reflection, and lifecycles dependably, increasing the result verification cost. On the other hand, current dynamic taint trackers accurately detect execution paths. However, they depend on specific Android versions and modified devices, reducing their usability and applicability. In addition, they require app exercise every time running the taint analysis. This paper presents a new dynamic taint tracker called T-Recs, tracking information flows by recording and reconstructing the app execution. First, before the taint analysis, the app's runtime data are obtained by instrumenting logging code into the app's bytecode and running the app to be independent of specific Android versions and devices. Then, T-Recs performs the taint analysis accurately with the logged data and separately from the app exercise. This paper is an extended version of our work published. Previously, T-Recs' accuracy was mainly evaluated in privacy leak detection. The results show that T-Recs outperforms compared analyzers, which are FlowDroid (w/ and w/o IC3), Amandroid, DroidSafe, and TaintDroid (w/ and w/o IntelliDroid). This paper also involves DroidRA and IccTA. This paper shows that T-Recs detects ICC- and reflection-related leaks missed by FlowDroid in popular Google Play apps. The other static analyzers fail to analyze most of the apps. These experiments also indicate an advantage of T-Recs: its users can re-execute T-Recs' taint analysis without re-exercising the app. T-Recs' app-runtime overhead and parallel execution performance were also evaluated, and the results are acceptable.

INDEX TERMS Android security, information flow, privacy leak detection, taint analysis.

I. INTRODUCTION

Protecting smartphone users has attracted increasing interest due to the appearance of suspicious apps and third-party SDKs. App analysts apply automatic analysis techniques to large-scale datasets of Android apps to detect suspicious behaviors. For example, Zhao et al. uncovered backdoor and blocklist secrets in apps published on Google Play and Baidu Market and pre-installed apps [1]. They used a static taint analysis tool called FlowDroid [2] to uncover the secrets. Static taint analysis is popular in analyzing a large-scale dataset because of its scalability.

However, static taint analysis has the problem of detecting incorrect execution paths, increasing the cost of verifying experiment results. Recent reviews of the literature on static taint analysis showed the limitations of the analysis [3], [4]. Zhang et al. evaluated currently-available static taint analysis tools: FlowDroid, Amandroid [5], and DroidSafe [6] with DroidBench [7] apps supported by the tools and real-world apps. The results show that the tools are inaccurate and cannot be used for analyzing real-world apps dependably. Handling inter-component communication (ICC), reflective calls, and component lifecycles are significant challenges. For example, FlowDroid supports ICC detection with IccTA [8] and IC3 [9], which are shown to be unreliable. The increase in false positives (FPs) complicates analysis-result verification, causing an increase in analysis cost. For example, Zhao et al. manually analyzed 70 out of over 16,000 detected apps to estimate the accuracy, and the result is 87.14% (i.e., nine apps are FPs) [1]. Three of the FPs were caused by conflicting constraints along the execution path. Such a manual analysis, especially finding path constraints, is complex and requires

The associate editor coordinating the review of this manuscript and approving it for publication was SK Hafizul Islam.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

significant effort. Also, increasing the complexity of the analysis algorithms increases the analysis time, precluding the analysis from completing in a reasonable time.

On the other hand, dynamic taint analysis only analyzes the executed paths, and there is no chance of detecting incorrect execution paths. For example, TaintDroid [10] generates no FP in the privacy leak evaluation with 30 popular real-world apps. However, current dynamic taint analysis tools depend on specific devices and versions of Android OS. It decreases their usability [11] and the range of analyzable apps. In addition, the app exercise must be performed every time the taint analysis runs. Therefore, when the analyst or researcher changes the parameters or features of the taint analysis and re-analyzes the same app, the app exercise also needs to be executed, which incurs extra costs.

This paper presents a new dynamic analysis system called T-Recs, with which users can start analyzing apps immediately after plugging an unmodified device into their computer. T-Recs first records the target app's runtime information at almost instruction-by-instruction. Then, it accurately reconstructs the app execution on the computer with the logged information to track information flows and detect information leaks, whereas overcoming the device dependency issue. The computer is outside the Android framework in contrast to TaintDroid, which is implemented on the Android framework. Also, the logs are stored in the analysis computer so that the taint analysis can be executed independently of the app exercise.

T-Recs consists of five components: parser, instrumentator, logger, reconstructor, and exerciser. The parser, the instrumentator, and the logger procure the app's runtime information with as minimal bytecode instrumentation as possible. Then, the reconstructor reproduces the app execution based on the parsed and logged information. The exerciser addresses how to trigger the target behavior in apps, a general challenge in dynamic analyses. App exercise depends on the apps and the data to be tracked. For example, tracking user inputs requires input-related exercises. The exerciser triggers ICC, callbacks, and lifecycles in the DroidBench apps because the static taint analysis has limitations for handling them.

This paper is an extended version of the conference paper [12]. Previously, T-Recs' accuracy, analysis time, and success rate were evaluated in privacy leak detection compared to currently available taint analyzers, which are FlowDroid (w/ and w/o IC3), Amandroid, DroidSafe, and TaintDroid (w/ and w/o IntelliDroid [13]). The results show that T-Recs outperforms the compared tools in detection accuracy. T-Recs also achieves reasonable analysis time and success rate. T-Recs has been made available to the community.

This paper further evaluates T-Recs. DroidRA [14], [15] (w/ FlowDroid, Amandroid, and DroidSafe) and IccTA are additionally included. Also, this paper shows that T-Recs detects ICC- and reflection-related leaks missed by FlowDroid in popular Google Play apps collected in 2016 and 2021. To identify and count the leaks, a debugging feature was added to the reconstructor. Then, only the reconstructor was executed. These experiments were conducted once the overall analysis was finished, indicating that T-Recs' taint analysis (i.e., the reconstructor) can be re-executed without re-exercising the app, which is one of T-Recs' advantages. T-Recs' app-runtime overhead (i.e., overhead for apps to be installed, launch, cause leaks, and be uninstalled) and parallel execution performance were also evaluated in comparison with the other trackers. The results are acceptable, and running T-Recs in parallel can easily shorten the analysis time. This paper also provides a more detailed description of T-Recs.

Here is the summary of our contributions:

- We developed a mechanism for accurately reconstructing the app execution outside the Android framework based on the app's runtime information logged on a device.
- We implemented the mechanism into a new dynamic taint analysis system called T-Recs with nearly 17,000 lines of Python and Smali code.
- We demonstrated T-Recs' leak detection performance compared to FlowDroid, Amandroid, DroidSafe, DroidRA, IccTA, TaintDroid, and IntelliDroid with DroidBench, 254 popular apps from Google Play in 2016 and 2021, and SDK-version-varied apps from Google Play and Anzhi [16].
- The importance of tracking ICC- and reflection-related flows is highlighted by T-Recs, detecting these flows in ten apps and related leaks in six apps among 96 apps from Google Play in 2016 and also these flows in 52 apps and associated leaks in 29 apps among 158 apps from Google Play in 2021.
- The additional experiments indicate that T-Recs's cost of re-executing taint analysis is small, taking 34 minutes (17% of the whole) for the 96 apps collected in 2016 and one hour and 40 minutes (11% of the total) for the 158 apps collected in 2021.

The rest of this paper is organized as follows. Information leaks and taint analysis are explained in Section II. Our approach is presented in Section III, and its implementation is described in Section IV. Our evaluation is reported in Section V, and the results are discussed in Section VI. Related work is explained in Section VII. Lastly, our conclusion is stated in Section VIII.

II. BACKGROUND

This section first discusses an information-leaking app's code. Then, it explains information flow tracking and current taint analysis approaches.

A. INFORMATION-LEAKING APP's CODE

An app can leak sensitive information with, for example, the Smali code in Fig. 1. In Smali, one class is defined per file, similar to Java. The class name is printed in the first line of

```
1  .class LLeaker;
2
3  imei:Ljava/lang/String;  // field
4
5  .method public constructor <init>()V
6
7      invoke-direct {p0}, Landroid/app/Application;-><init>()V
8  .end method
9
10 .method public callback1()V
11
12     invoke-virtual {v0}, Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;  // source
13
14     move-result-object v1  // tainted
15
16     if-eqz v1, cond_0
17
18     iput-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field setter
19
20     :cond_0
21 .end method
22
23 .method public callback2()V
24
25     iget-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field getter
26
27     invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I  // sink
28 .end method
```

FIGURE 1. **Smali code leaking the sensitive information.**

the Smali file. It is prefixed with a capital *L* and suffixed with a semicolon. In Fig. 1, *Leaker* is the class name. A field of *Leaker* is defined in Line 3. The left part of the colon is the field name (i.e., *imei*) and the right part is the data type (i.e., *Ljava/lang/String;*). The data type is *java.lang.String* in Java, and slashes are used instead of dots in Smali.

Leaker has three methods defined in Lines 5-28. A method definition begins from a line starting with *.method* to next *.end method* line. A constructor is defined in Lines 5-8; *callback1()*, 10-21; and *callback2()*, 23-28. The *V* attached to the method names' ends indicates the data type of the return value, and *V* means void in Java. The *invoke* instructions, such as *invoke-direct* in Line 7, *invoke-virtual* in Line 12, and *invoke-static* in Line 27, are used to call a method. The *move-result-object* instruction assigns the return value of the most recent *invoke* instruction to the destination register (e.g., *v1* in Line 14). The *if-eqz* instruction is one of the branch instructions, and the path to the label is taken if the operand register value is 0 (e.g., if *v1*'s value is 0 in Line 16, Line 18 is skipped, and Line 20 is subsequently executed). The *iput-object* and *iget-object* instructions in Lines 18 and 25 are a setter and a getter of instance fields, respectively.

B. INFORMATION FLOW TRACKING

The information flow starts when the code obtains a device's hardware identifier (i.e., IMEI) in Lines 12-14 in the method *callback1()*. The code sets the value to the field *imei* in Line 18. Then, the code moves the value from the field *imei* to the register *v1* in Line 25 and leaks it by calling *Log.i()* in Line 27 in the other method *callback2()*.

Assume that *callback1()* is called, a taint tracker can assign a taint tag to the register *v1* in *callback1()* and propagates the taint from *v1* to the field *imei*. A challenge is to determine whether *callback1()* and *callback2()* are executed in this order. Execution of the methods depends on ICC, user-interface events, and the app's lifecycle. If a tracker overapproximates the call flows, the leak is falsely detected (i.e., FP). Alternatively, if a tracker underapproximates, the leak is missed (i.e., false-negative (FN)).

C. STATIC TAINT ANALYSIS

Static taint analysis requires no Android device and processes apps without running them. A significant challenge is to obtain precise Android models to find correct execution paths (e.g., the execution order of *callback1()* and *callback2()* in Fig. 1). Also, the execution order of instructions changes based on system properties, such as OS version and IMEI (e.g., Line 16 in Fig. 1).

Considerable effort has been devoted to Android-modeling techniques; however, the limitations are demonstrated [4]. Zhang et al. showed that currently-available static taint analysis tools produce many FPs and FNs in DroidBench apps that contain ICC- and lifecycle-related code similar to Fig. 1. They also evaluated the tools with real-world apps and concluded that none of them was reliable. The increase of FPs increases the analysis cost and complicates the verification of analysis results. Also, increasing the complexity of analysis algorithms multiplies the analysis time, making it challenging to complete the analysis in a reasonable time.

D. DYNAMIC TAINT ANALYSIS

Dynamic taint analysis uses the target app's runtime semantics. For example, TaintDroid [10] performs taint tracking within the Dalvik virtual machine interpreter, and the Android models are not used. Since only the executed paths are analyzed, FPs, due to misestimating call flows and control flows, do not occur. The computation required for the Android modeling is no longer necessary, and the analysis time does not depend on the modeling.

However, Reaves et al. [11] discuss that TaintDroid is the most difficult to set up in comparison with static analysis tools they audited because TaintDroid requires the user to build the Android source code. Also, a supported device is not always available. Other current dynamic taint analyzers could be easier to set up; however, they have been barely examined in the community and are not effortlessly usable, which are discussed in Section VII.

TaintDroid's other drawback is that the app exercise needs to be executed every time running the taint analysis because the app exercise and the taint analysis are performed simultaneously in TaintDroid. Therefore, when the analyst changes the parameters of the taint analysis (e.g., the data to be tracked) and re-analyzes the same app, the app exercise also needs to be repeated, which incurs extra costs. It also distresses researchers who add new features to the taint analysis and evaluate them.

Another issue is the app exercise itself. Since a dynamic taint analysis only analyzes executed part of the app's code, triggering the target behavior in the app is necessary. Monkey [17], a popular UI/application exerciser, exercises the app randomly. Random exercise is inefficient in triggering a leak in practice because sometimes a leak is only triggered by a particular sequence of UI operations. For example, some of the DroidBench apps require a specific sequence of operations to trigger leaks as shown by *callback1()* and *callback2()*,

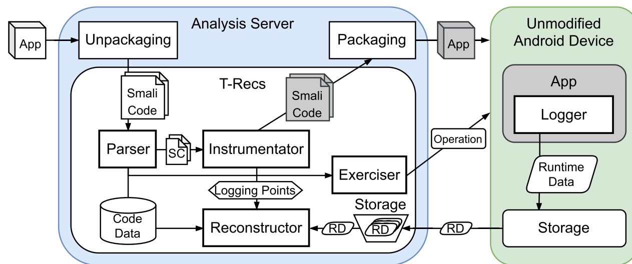


FIGURE 2. Overview of our approach.

which must be called in this order to cause the leak (Fig. 1). Finding such operation sequences using random input can take much time or fail to trigger leaks.

III. APPROACH

This section presents a new taint analysis system addressing the model accuracy, device dependency, and re-analysis cost issues.

A. OVERVIEW

The system is designed to be automatic for analyzing large-scale datasets. It performs dynamic taint analysis outside the Android framework to accomplish accuracy, usability, and small re-analysis costs. There are the following challenges:

- How to implement a mechanism to provide app runtime information outside the Android framework in a way that is effective for many real-world apps.
- What kind of app runtime information should be used to accurately reconstruct the app execution and track information flows outside the Android framework.
- Since this is a dynamic analysis, how to automatically exercise the app to trigger the target behavior should be addressed.

The system consists of five components that address these challenges. Fig. 2 shows the overview of the system. After the user plugs the unmodified Android device into the analysis server, the setup is finished, and the analysis server first unpackages the app and extracts the app's Smali code.

- **Parser** extracts the app's information from the Smali code to reduce the information to be logged.
- **Instrumentator** injects the logger into the app code. It also provides the logging point information to the reconstructor.
- **Logger** saves the app runtime information at the app's bytecode level. It is independent of specific Android devices and versions and requires no device modification, such as rooting. The logs are eventually stored in the analysis server.
- **Reconstructor** reproduces the app execution, including call-, control-, and dataflows on the analysis server based on the parsed and logged data. The logs are saved in the storage of the analysis server so that the reconstructor can be executed separately from the logger.

```
1  .class LLeaker;
2
3  imei:Ljava/lang/String;  // field
4
5  .method public constructor <init>()V
6        invoke-static {}, LTRecsLog;->Log_1_5()V
7
8        invoke-direct {p0}, Landroid/app/Application;-><init>()V
9        invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_7_p0(Landroid/app/Application;)V
10 .end method
11
12 .method public callback1()V
13       invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_10_p0(LLeaker;)V
14
15       invoke-virtual {v0}, Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;  // source
16
17       move-result-object v1
18       invoke-static/range {v0 .. v0}, LTRecsLog;->Log_1_14_v0(Landroid/telephony/TelephonyManager;)V
19       invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_14_v1(Ljava/lang/String;)V
20
21       if-eqz v1, cond_0
22
23       iput-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field setter
24
25       :cond_0
26 .end method
27
28 .method public callback2()V
29       invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_23_p0(LLeaker;)V
30
31       iget-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field getter
32       invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_25_v1(Ljava/lang/String;)V
33
34       invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I  // sink
35       invoke-static/range {v0 .. v0}, LTRecsLog;->Log_1_27_v0(Ljava/lang/String;)V
36       invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_27_v1(Ljava/lang/String;)V
37 .end method
```

FIGURE 3. Example of the instrumentation applied to the code in Fig. 1. The red-colored lines are logging points injected into the code.

- **Exerciser** cooperates with the reconstructor to automatically trigger leaks caused by ICC, callbacks, and lifecycles in the DroidBench apps.

The rest of this section describes each component.

B. PARSER

The parser extracts class, method, field, and instruction information from the app's Smali code. The parser maps each class and method (e.g., class *Leaker* and methods *init()*, *callback1()*, and *callback2()* in Fig. 1) to distinguish between in-app and API methods, and the instrumentator uses the results. The parser also distinguishes fields (e.g., *imei* in Fig. 1) implemented in subclasses and superclasses because the classes can have the same name fields, and the fields are not distinguishable based on their names. The parser also extracts fields' default values hard-coded in the app code.

C. INSTRUMENTATOR

The instrumentator injects the logging code (i.e., the logger) into the target app's Smali code. In this section, first, logging points are described. Then, logging-method construction is explained. Next, the type-conflict problem and a solution are discussed. Lastly, DEX-related problems and solutions are described.

1) LOGGING POINTS

Fig. 3 shows an example of the instrumentation. The red-colored lines are logging points in the app code where the logging method invocations are injected. The logging methods are static and are called by the *invoke-static/range* instruction. An argument register is passed to the logging methods with brackets for logging the register's value. For example, *p0* is passed to the logging method by *{p0.. p0}* in Line 9, and *p0*'s value will be logged. The subsequent *LTRecsLog;→Log_1_7_p0* specifies the called class and

method names of the logger. *TRecsLog* is the class name and *Log_1_7_p0* is the method name. The method name consists of the target class identifier, the original instruction line number in the code before the instrumentation (Fig. 1), and the register name, which are *1*, *7*, and *p0* respectively. The class identifier is assigned to each class (i.e., each Smali file) in the app. The following *(Landroid/app/Application;)* is the data type of the argument *p0*. *V* at the end is the data type of the method's return value and is void because the logging methods return nothing. The logging method in Line 6 has no argument, and no value is logged at the point. The logging points are as follows:

- immediately after field-getter instructions to save values loaded into the destination registers because fields can be modified outside the app code (e.g., Line 32 in Fig. 3). Also, logging at static-field operators informs the reconstructor about the timing of the *clinit* invocation.
- right after *monitor-enter*, *const-class*, and *check-cast* instructions and catch labels.
- immediately after method calls (e.g., Lines 9, 18, 19, 35, and 36 in Fig. 3) and at the head of each method in the app code (e.g., Lines 6, 13, and 29 in Fig. 3). The logs are used to determine method call relationships accurately, which is explained in Section III-E3. Argument values are also recorded at the head of the method and used for argument mapping. The logger skips value logging for constructors because constructors have an uninitialized object reference as its base object at the method head (e.g., Line 6 in Fig. 3). The return value is recorded for API method calls at the corresponding *move-result* (e.g., Line 19 in Fig. 3) and used for the return value mapping, described in Section III-E3. The value of reference-data-type arguments is also recorded after the method call since the method may modify the arguments.

The instrumentator considers reducing the instrumentation code volume for app runtime performance. Results of arithmetic and logic operations and conditional branches (e.g., Line 21 in Fig. 3) are not logged to reduce the amount of instrumentation code. Also, no logging code is injected to the end of each method (e.g., Lines 10, 26, and 37 in Fig. 3). Instead, the reconstructor simulates the operations on the server based on reproduced register values.

2) LOGGING-METHOD CONSTRUCTION

Fig. 3 indicates that a static logging method is constructed for every instrumented instruction. It avoids using local variables in the instrumented methods to reduce the impact on the original code.

Suppose all the logging points call the same logging method. In that case, each logging point must provide information, including the class identifier, line number, and register name to the logging method (i.e., each logging point needs a local variable to keep the information). However, introducing an additional local variable for the logging to a method can fracture the original code. A method can use registers *v0* to *v65535* for the method's local variables and parameters in Smali. The method's local variables are first assigned to registers from *v0*. Then, The method's parameters are assigned to registers. For example, if a method has 14 local variables and 2 parameters, the local variables use registers *v0* to *v13*, and the parameters use *v14* and *v15*. Assuming that an additional local variable is used for the logging, the local variables now use registers *v0* to *v14*, and the parameters use *v15* and *v16*. The second parameter's register is changed from *v15* to *v16*, which is not acceptable because whereas the first 16 registers *v0* to *v15* can be operated by all the instructions, *v16* and later registers are limited that only specific instructions can operate the registers. As a result, the instrumentator must rewrite the original code's instructions related to the second parameter, which is complex and better to be avoided. Therefore, the logging is designed to use no local variables in the instrumented methods. Since sharing a logging method among multiple logging points requires an additional local variable, a logging method is constructed for every logging point.

```
1  :try_start_0
2  invoke-static {}, LClass1;->method1()I;  // return an integer value
3
4  move-result v1              // originally set the return value to v0
5  invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_4_v0(I)V
6  move v0, v1
7
8  int-to-float v0, v0                       // convert integer to float
9  goto :goto_0                              // jump to 14
10 :try_end_0
11 .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0
12 :catch_0          // if an exception occurs from 1-10, jump to here
13 invoke-static/range {}, LTRecsLog;->Log_1_11()V
14 :goto_0
15 invoke-static {v0}, LClass2;->method2(F)V;    // v0 must be float
```

FIGURE 4. Instrumented code of exception handling. The modified part and injected code are red-colored.

3) TYPE-CONFLICT PROBLEM

The instrumentation code must not cause errors while the app is running. Solving the type-conflict problem discussed in [18] is challenging. Instrumenting an app can make data types of a register potentially conflicted within a method of the app.

Fig. 4 shows an example of the instrumentation applied to an exception-handling code. The modified parts of the code are red-colored. A try block starts at *:try_start_0* in Line 1 and ends at *:try_end_0* at Line 10. If no exception occurs between Line 1 and 8, the *goto* instruction changes the program counter to *:goto_0* in Line 14, and *invoke-static* instruction in Line 15 is subsequently executed after the try block. In the original code, only *invoke-static* in Line 2 can cause an exception in the try block. Thus, the catch block would never be executed after the *invoke-static* in Line 2 is successfully executed, and the instructions in Lines 4, 8, and 9 are necessarily executed. In other words, in the original code, after an integer value is assigned to *v0* in Line 4, the value is certainly converted to float by *int-to-float* in Line 8.

On the other hand, if an exception occurs in the try block, the execution point is changed to *:catch_0* in Line 12. Then, instructions in Lines 13-15 are subsequently executed. In the

code after the instrumentation, *invoke-static/range* in Line 5 can cause an exception in the try block, indicating that *int-to-float* in Line 8 can be skipped, and *v0* holds an integer value when the instructions after Line 12 are executed. In that case, the type-conflict problem occurs because *v0*'s value must be float for *invoke-static* in Line 15. In this way, the instrumentation can introduce a new exceptional flow leading to the type-conflict problem.

Since a verifier module of the Android runtime system always assumes that *invoke* causes exceptions, it detects the type conflict and terminates the app execution. To address the problem, Balachandran et al. developed a register-type separation technique, which rewrites the whole code of the app [18]. On the other hand, our approach replaces the destination register of *move-result* in a try block (e.g., *v0* at Line 4) with an unused register. We call the approach temporary-register technique. For example, *v1* is used as a temporary register instead of *v0* in Line 4, and the logger saves the *v1*'s value as the *v0*'s value in Line 5. After the logging, the value is moved from *v1* to *v0* in Line 6 to maintain the semantics. The technique is also applied to *move-exception*.

4) DEX-RELATED PROBLEMS

The instrumentator should avoid the 64K problem [19], which restricts a DEX file to containing 65,536 method references at most. After the logging methods are injected, the instrumentator counts the number of method references in Smali files in each DEX file, which is a directory when the app is unpackaged. Then, the instrumentator rearranges the Smali files in a DEX directory into multiple DEX directories if the DEX directory contains more method references than the maximum.

The instrumentator also detects long-distance jumps between an if-statement and its jump destination. When an if-statement uses a 16-bit address to specify the jump destination, the jump distance could exceed the limit because of the injected code, and the app repackaging would fail. The instrumentator detects such jumps and replaces the jumps with a *goto/32* statement that uses a 32-bit address.

D. LOGGER

The logger is injected into the app code by the instrumentator. Then, T-Recs repackages, installs, and launches the app on the Android device, and the logger is also executed. The logger targets primitive-data-type values, class object representations, string values of String type classes, array representations, and array elements' values.

The logger converts the class object representations into strings to write them into a log file. However, the formats of string representations depend on the class's *toString()* implementation. Also, the *toString()* should not be used when the class overrides the method because calling the method can affect the app's behavior (e.g., it could change a field value). Therefore, the logger explicitly invokes *getClass().getName()* and *hashCode()*. The logger uses a ring buffer to keep logs in the memory and reduce the number of writing to the disk for the app runtime performance with a large number of logging.

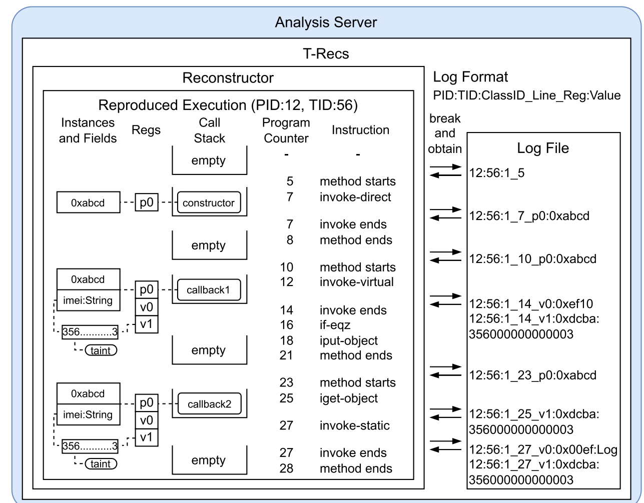


FIGURE 5. Example of app-execution reconstruction.

After the app is exercised for a time specified by the analysts, T-Recs terminates and uninstalls the app. Then, T-Recs collects the log file from the device and saves it in the analysis server.

E. RECONSTRUCTOR

The reconstructor reproduces the app execution, consisting of call-, control-, and dataflows, based on the information obtained by the parser, the instrumentator, and the logger. Simultaneously, the reconstructor propagates taints to track information flows.

Fig. 5 shows an example of reproducing the execution of the code in Fig. 3. The log file is obtained by executing the code in Fig. 3 and is stored in the storage of the analysis server so that the reconstructor can be performed independently of the logger. The log format is *PID:TID:ClassID_LineNumber*. PID and TID are the identifiers of the process and thread that executed the logging code. ClassID is the identifier of the class. LineNumber is the instrumented instruction's line number in the original code shown in Fig. 1. The log is followed by *_RegisterName:RegisterValue* if the instrumented instruction has an operand register to be logged.

The reconstructor reproduces the execution with PID 12 and TID 56 in Fig. 5. The reproduced execution includes currently-executed instruction, program counter, call stack, registers, instances, and fields. The call stack is empty at first, and the reconstructor starts by obtaining the first log *12:56:1_5*. The reconstructor sets the program counter 5 and simulates the instruction. Since program counter 5 is the head of *constructor*, the reconstructor creates its stack frame and pushes it to the call stack. Then, the reconstructor increments the program counter. The next instruction is *invoke-direct* at program counter 7, which is another logging point, and the reconstructor breaks the reproduction and obtains the next log *12:56:1_7_p0:0xabcd*. The reconstructor creates register

p0 and instance *0xabcd* based on the log. The reconstructor increments the program counter, reaches the end of the method, and removes the method's stack frame. Since the call stack is now empty, the reconstructor breaks the reproduction and obtains the next log, and the next method's reconstruction starts at program counter *10* in *callback1()*. By repeating these steps, the reconstructor reproduces the execution. The rest of this section explains how the reconstructor simulates register values, control flows, and call flows, and the taint propagation is described at the end.

1) REGISTER VALUES

The reconstructor reproduces register values based on logged object identifiers, strings, and primitive-data-type values. It considers registers' data types: primitive-data-type, reference-data-type, and class references.

Primitive-data-type values are reproduced based on the logs and data extracted by the parser. Boolean values, true and false, are represented by numeric values, 1 and 0, respectively, for using the values with branches (e.g., *if-eqz*). Unary and binary operations, such as numerical and logical calculations, with primitive-data-type values, are simulated by the reconstructor. The reconstructor explicitly uses the same bit length to obtain the same results of calculations as the actual execution.

Arrays and classes are reference-data-type, and registers of the data type hold object references. In the reconstructor, simulated registers hold references to array and class instances as same as the actual execution (e.g., register *p0* holds a reference to instance *0xabcd* in Fig. 5). Null values are represented by the numeric value 0, which is compatible with branches (e.g., *if-eqz*). An array's elements are logged and used to simulate array operations. The reconstructor also supports multidimensional arrays. The reconstructor manages fields of classes and handles static fields as a global area. When the app accesses an uninitialized field, the reconstructor simulates default values, 0 for numeric values and null for objects.

When multiple threads write and read the same field simultaneously, the reconstructor detects the timing of each operation based on logs at the *monitor-enter*. Figure 6 shows an example of instrumented code with *monitor-enter* and *monitor-exit* instructions. These instructions enable an app to perform exclusive control to maintain consistency when multiple threads use the same data in, for example, a field. In this example, *thread1()* sets a value to field *Leaker.imei* at Line 7, and *thread2()* gets the value from the field at Line 19. Since these instructions are placed between *monitor-enter p0* and *monitor-exit p0*, they are executed one at a time. The reconstructor can detect their execution order based on the logs generated at Lines 5 and 17, and the data flow from *v1* in Line 7 to *v1* in Line 19 is accurately reproduced. On the other hand, if an app does not use exclusive control, the reconstructor cannot reproduce the execution order of instructions in multiple threads accurately. However, in such case, the impact of incorrect reproduction might be small because the app's developer also disregards the execution order.

```
1  .method public thread1()V
2      invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_1_p0(LLeaker;)V
3
4      monitor-enter p0
5      invoke-static {}, LTRecsLog;->Log_1_3()V
6
7      iput-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field setter
8
9      monitor-exit p0
10     return-void
11 .end method
12
13 .method public thread2()V
14     invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_11_p0(LLeaker;)V
15
16     monitor-enter p0
17     invoke-static {}, LTRecsLog;->Log_1_13()V
18
19     iget-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field getter
20     invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_15_v1(Ljava/lang/String;)V
21
22     monitor-exit p0
23     return-void
24 .end method
```

FIGURE 6. Instrumented code with *monitor-enter* and *monitor-exit* instructions.

Class references are generated by *const-class* instructions and used by branches and method calls. The reconstructor simulates class references based on logged object representations.

2) CONTROL FLOWS

The reconstructor reproduces control flows in each method of the app, which consists of conditional branches (i.e., *if* and *switch*), unconditional jumps (i.e., *goto*), and exceptional flows (i.e., *try*, *catch*, and *throw*). App code is written in Dalvik executable (DEX) bytecode [20], which is register-based, and conditional branches operate on registers. Hence, the reconstructor simulates conditional branches based on the reproduced register values (e.g., program counter 16 in Fig. 5).

Simulating exceptional flows requires the detection of exception sources and exceptional-jump destinations. The reconstructor detects exception sources based on simulation results (e.g., ArrayOutOfBoundException by simulating arrays) and the logs (e.g., exception-causing calls by checking the completion of each call). The reconstructor checks a log that must appear right after a finished call, and if the log is not found, the reconstructor understands that an exception is caused during the call. For example, the reconstructor detects that the call at Line 15 in Fig. 3 causes no exception based on log *12:56:1_14_v0:0xef10* (Fig. 5). In the same way, the reconstructor detects whether a *check-cast* instruction throws an exception. The reconstructor also breaks the trace and checks the next log at throw instructions. The reconstructor detects exceptional-jump destinations based on the logs from catch blocks (e.g., the logging point at Line 13 in Fig. 4), which can be in a different method from the exception source.

3) CALL FLOWS

There are various patterns of method calls involving callbacks, lifecycles, ICC, reflection, threading, and constructors. It indicates that only one-to-one mapping of parameters and arguments fails to detect dataflows from a caller to a callee.

Also, the return value from a caller to a callee is not one-to-one because a callee can return a value to outside the app code, or a caller can receive a returned value from outside the app code.

The logs provide the reconstructor with the timings of starting and ending of each method invocation and the timings of starting methods. The reconstructor breaks the trace when it reaches a method invocation instruction and checks the next log. If the following log is generated at the next line of the invocation, the invoked method is an API, not implemented in the app. If the following log is a method head's, an in-app caller-callee relationship is detected. The reconstructor utilizes the logged register values to match parameters and arguments from the caller to the callee. When the callee is finished, the reconstructor matches the return value from the callee to the caller based on the logged register values.

When the app executes multiple threads, all the threads' logs are mixed in the log file. The reconstructor distinguishes threads using each log's process and thread identifiers. Suppose the next log has new process and thread identifiers. In that case, the reconstructor considers that a new thread is starting and matches the base object's representation to the previously-created instance's representations.

Detecting implicit control flow transitions facilitated by the callback mechanism in the Android framework is challenging for static taint analyzers [21]. Figure 7 shows an example of source code causing an implicit control flow transition. In the application space, *MainClass.main()* creates and passes a *Leaker* instance to *SystemClass1.method1()* in Lines 3 and 4. Then, in the framework space, *SystemClass1.method1()* invokes *appClass.callback1()* in Line 3, which is *Leaker.callback1()*, defined in Figure 1. *Leaker.callback1()* obtains IMEI, which can eventually be leaked. Therefore, a taint tracker must detect this control flow transition. The reconstructor cannot detect the relationship between *SystemClass1.method1()* and *Leaker.callback1()* because it occurs in the framework space. However, such a relationship is unnecessary because the reconstructor can detect the leak in Figure 1 based on logs generated at *Leaker.callback1()* and *Leaker.callback2()*. Also, based on PIDs and TIDs in the logs, the reconstructor can detect the exact time sequence of the method executions in a thread. On the other hand, if a callback method is executed in a different thread, the reconstructor identifies the time sequence as described in Section III-E1.

The reconstructor resolves the target class and method names of reflective calls to detect calls of taint sources and sinks. The reconstructor uses the argument values of the calls. Also, the reconstructor must consider that a taint source or sink can be called with an in-app class inheriting the class of the taint source or sink as the base object. The reconstructor resolves the called method's superclass based on class hierarchy information extracted by the parser.

There is a concern that the reconstructor may take a long time to analyze loops with a large number of iterations.

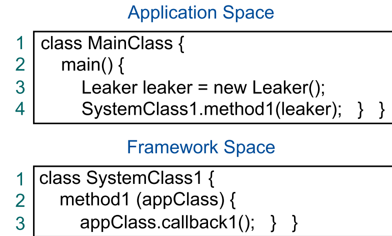


FIGURE 7. Simplified example of source code causing an implicit control flow transition.

In our preliminary investigation of the DroidBench apps, such loops were found in the method *computePi()* in PI1 from the category Emulator Detection (ED). The method has no parameters and no return value (condition 1). In addition, the method has no API invocation or field operation in the method body (condition 2). The method does not affect anything outside the method, and the reconstructor can safely skip the method. Therefore, by checking the two conditions, the reconstructor automatically detects such a method as an anti-analysis technique and skips it.

4) TAINT PROPAGATION

Taint propagation is required at DEX bytecode instructions [20] and across API method calls to track information flows. Taint propagation rules for DEX bytecode instructions are well developed in previous studies such as TaintDroid [10], and the reconstructor utilizes the same rules. TaintDroid assigns taints to registers, but the reconstructor assigns taints to simulated class instances. For example, in Fig. 5, the reconstructor detects the execution of the taint source at the program counter 14 and introduces the taint by assigning the taint mark to the string instance *356..3*, which will be stored in the field *imei* of the object *0xabcd*. When a reference to a class is moved between registers by register operations, the taint propagation is implicitly achieved, which is an advantage in simulating class instances. When a class instance field is operated through different registers holding the same reference (i.e., aliasing), the operations are implicitly applied to the same field of the same instance. For example, registers *p0* in *callback1()* and *p0* in *callback2()* reference the same object *0xabcd* with the field *imei* holding the tainted string in Fig. 5.

Previous studies developed conservative rules [22] and automatic model generators (e.g., StubDroid [23]) for propagating taints across API method calls. The reconstructor, performing on the analysis server, can be equipped with current approaches used by static and dynamic taint trackers. In this study, an approach that conservatively propagates taints is simply used. There are some dataflows that the conservative rules cannot track. The reconstructor considers Intent, Message, Bundle, Shared Preferences, Parcel, and files. The reconstructor uses API class names and matches values between setter and getter methods. By propagating the taint status, the reconstructor not only assigns the taint but also removes the taint. It can refine over-tainting caused by the conservative rules and reduce FPs.

TABLE 1. App exercise operations necessary to trigger leaks in the DroidBench apps.

Operation	Command	Necessary information
Launch app	monkey	app's package name
Kill app process	ps and kill	app's package name
Rotate screen	settings put system user_rotation	-
Press home button	input keyevent	-
Press back button	input keyevent	-
Start activity	am start	activity name
Send broadcast	am broadcast	receiver name
Start service	am start-service	service name
Tap screen	input tap	coordinates

F. EXERCISER

After the app is installed on the Android device, the exerciser performs app exercise operations to trigger leaks in the app. The exerciser focuses on how to exercise the DroidBench apps in this paper. Table 1 shows necessary operations to trigger leaks in the DroidBench apps.It also shows commands to perform each operation and necessary information, such as the app's package name and activity name, which are passed to the commands. Leaks in some of the DroidBench apps can be triggered by only launching them. On the other hand, some of the DroidBench apps require a sequence of operations specific to individual apps. Therefore, exercising the DroidBench apps using random input can take much time until a specific operation sequence is performed by chance.

The exerciser shortens the analysis time by triggering leaks in the DroidBench apps as quickly as possible. The exerciser iteratively runs the app and executes the reconstructor to detect information flows. The exerciser checks the reconstructor's result, and if the number of taint marks increases from the previous reconstruction result, the exerciser saves the current sequence of operations and uses it in the next turn.

Algorithm 1 shows the exerciser's pseudocode. The arguments are the app's data (*app*), a device with the app is installed (*device*), and the maximum number of operations to be performed (*max_op_num*). There are two loops in the procedure (Lines 7 and 11). In the outer loop, exercise operations that increase the taint marks (*taint_increasing_ops*) are performed in Lines 8-9, and performable operations (*performable_ops*) are obtained in Line 10. The performable operations include the operations shown in Table 1. For example, the coordinates of the app's UI buttons are detected, and the tap operation is prepared for each button. Then, in the inner loop, *taint_increasing_ops* are performed, and one of the performable operations is performed in Lines 15-19. The log is obtained in Line 20, and the reconstructor is executed with the log in Line 21. Based on the newly found leaks and the number of taint marks (*leaks* and *taint_num*), the exerciser stops the procedure (Lines 25-26), exits the inner loop (Line 27-33), or inserts *op* to *performable_ops*' head to retry the same operation (Line 34-35). Some DroidBench apps change their behavior depending on random numbers, and the number of taint marks can be different even for the same operation. Therefore, the exerciser retries the same operation as long as the number of taint marks decreases to trigger the app's information-flow-causing behavior. When the inner loop is finished, the outer loop also ends if the number of taint marks does not increase (Lines 38-39).

Also, determining when the exerciser exits is essential. The exerciser should not stop when a leak is detected, as some of the apps cause multiple leaks. The exerciser should also not run indefinitely, as reaching full coverage in runtime is very difficult. Therefore, in addition to the condition (Line 38-39),

Algorithm 1 App Exercise Procedure

```
 1: procedure exercise(app, device, max_op_num)
 2:     op_num ← 0
 3:     prev_taint_num ← 0
 4:     found_leaks ← empty list
 5:     taint_increasing_ops ← empty list
 6:     max_leak_num ← get_max_leak_num(app)
 7:     while true do
 8:         stop app and remove the logs on device
 9:         perform taint_increasing_ops
10:         generate performable_ops
11:         while performable_ops is not empty do
12:             if op_num > max_op_num then
13:                 return
14:             end if
15:             stop app and remove the logs on device
16:             perform taint_increasing_ops
17:             op ← pop an item from performable_ops
18:             perform op
19:             op_num ← op_num + 1
20:             log ← logs obtained from device
21:             leaks, taint_num ← reconstructor(log)
22:             if leaks not in found_leaks then
23:                 found_leaks ← leaks|found_leaks
24:                 leak_num ← length of found_leaks
25:                 if leak_num = max_leak_num then
26:                     return
27:                 else
28:                     taint_increasing_ops ← empty list
29:                     break
30:                 end if
31:             else if taint_num > prev_taint_num then
32:                 append op to taint_increasing_ops
33:                 break
34:             else if taint_num < prev_taint_num then
35:                 append op to performable_ops' head
36:             end if
37:         end while
38:         if taint_num ≤ prev_taint_num then
39:             break
40:         end if
41:         prev_taint_num ← taint_num
42:     end while
43: end procedure
```

the exerciser stops when one of the following conditions is satisfied. First, the exerciser limits the number of performed operations (Line 12). Second, the exerciser exits when all the leaks are found in the app (Line 25). The *max_leak_num* is calculated before the exercise (Line 6) and is the number of all possible combinations of taint sources and sinks, including reflective calls, in the app extracted by the parser.

In addition, the exerciser lets the reconstructor simulate triggering non-triggerable callback methods. A callback method *onLowMemory()* is barely called in the DroidBench apps because the Android OS executes it only with memory-consuming apps. In the DroidBench test cases, five apps are identified to contain the callback method. The callback method executes a taint source, sink, or both. The exerciser tells the reconstructor to trigger *onLowMemory()* apart from the actual runtime. The reconstructor reproduces the app execution without the app's runtime information (i.e., without breaking the reproduced execution at logging points) and detects information flows and leaks caused by *onLowMemory()*.

IV. IMPLEMENTATION

Python is used to implement all the components except the logger. The logger is implemented with the Smali language. The system is called T-Recs and is about 17,000 lines of code. T-Recs uses the Apktool [24] version 2.6.1 to unpackage and repackage the apps. The reconstructor uses Python's references, exceptions, and lists to reproduce references, exceptions, and arrays in the app execution. NumPy is used to simulate the same bit length of numeric values in the reconstructor. The instrumentator performs the temporary-register technique, explained in Section III-C2, only for methods in that two more registers are available.

The logger currently targets a limited depth of arrays, which is two-dimensional. The supported level of depth can be trivially expanded by modifying the logger to record more items in multidimensional arrays. However, the modification could affect the app-runtime performance.

As Section III-F explains, our exerciser is a prototype only for the DroidBench apps. The callback-method triggerer is implemented to reproduce the execution of *onLowMemory()*, the only method that cannot be triggered on Android devices in the DroidBench apps. Since taint sources must be executed before taint sinks to cause the leaks, the timings of the method execution are at each constructor's end and the whole reconstruction's end, which were determined based on the investigation of the DroidBench apps.

V. EVALUATION

This section presents T-Recs' evaluation with a test suite and real-world apps. First, this section explains datasets. Then, it describes compared tools and analysis results of each dataset. Lastly, it explains ethical considerations.

A. DATASETS

The following datasets were used.

1) DroidBench 3.0

DroidBench is a popular test suite initially published in 2014, covering a wide range of language- and Android-specific categories. DroidBench had 19 categories and 190 test cases in total when this paper was written. This paper focuses on 158 test cases in 13 categories supported by current static taint trackers: FlowDroid, Amandroid, and DroidSafe [4], [25] to evaluate how T-Recs outperforms the trackers in accuracy with the supported cases. The categories are Aliasing (A), Android Specific (AS), Arrays and Lists (AL), Callbacks (C), ED, Field and Object Sensitivity (FO), General Java (GJ), ICC, Lifecycle (L), Reflection (R), Reflection ICC (RICC), Threading (T), and Unreachable Code (UC).

2) POPULAR APPS FROM GOOGLE PLAY IN 2016

Analysis accuracy, time, and success rate were evaluated for detecting privacy leaks in real-world apps. Since TaintDroid detects leaks of sensitive information, such as IMEI and IMSI, a dataset in this evaluation must contain many apps that obtain and leak the information. Popular Google Play apps from the Agrigento dataset have such apps, 22 apps leaking IMEI and six apps leaking IMSI [26]. They were collected in June 2016. The app set contains 96 apps given by the authors of [26].

3) VARIED DATASET FROM GOOGLE PLAY AND ANZHI

The app set contains randomly-collected 19,943 apps from Google Play and 19,537 apps from Anzhi, 39,480 apps in total, via AndroZoo [27] in September 2021 for evaluating the success rates of the compared tools' essential phases. Anzhi was selected as the representative of third-party app markets because Anzhi was the market with the largest app collection after Google Play [27]. The Google Play apps support SDK versions from one to 28 (i.e., 16 codenames), and the Anzhi apps support SDK versions from one to 25 (i.e., 14 codenames). The datasets vary in supported SDK versions. Also, the distribution of apps' supported SDK versions differs between the markets, and the tools were evaluated with a wide range of SDK versions.

4) POPULAR APPS FROM GOOGLE PLAY IN 2021

The leak detection number and analysis time were evaluated with newer real-world apps. The app set contains 158 apps that appeared in the top chart list of free apps in Google Play in July 2021. Since these apps are recently developed, 98% of them have *androidx.** packages [28]. Leaks caused by the packages were ignored because the packages are official and can be considered benign.

B. PRIVACY LEAK DETECTION IN DroidBench 3.0

This section describes the evaluation results of DroidBench 3.0 to show T-Recs' superiority over current trackers in detection accuracy. It also discusses the analysis time.

TABLE 2. Results of DroidBench. The second column shows the expected #leaks. Gray cells highlight accurate results.

Test (#)	E	T-Recs			$FlowDroid_{IC3}$			FlowDroid			Amandroid			DroidSafe			$DroidRA_F$			$DroidRA_A$			$DroidRA_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
A (4)	1	1	0	0	1	1	0	1	1	0	1	1	0	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1	0	0	1	0	0
AS (11)	8	8	0	0	7	1	1	7	1	1	4	0	4	7	1	1	7	1	1	4	0	4	7	1	1	6	0	2	5	0	3	5	0	3
AL (10)	4	4	0	0	4	3	0	4	5	0	1	4	3	4	4	0	4	5	0	1	4	3	4	4	0	3	5	1	1	0	3	1	0	3
C (15)	18	18	0	0	15	2	3	15	2	3	2	1	16	18	4	0	15	2	3	2	1	16	18	4	0	16	2	2	1	0	17	1	0	17
ED (15)	16	16	0	0	16	0	0	16	0	0	15	0	1	11	0	5	16	0	0	15	0	1	11	0	5	16	0	0	9	0	7	9	0	7
FO (7)	2	2	0	0	2	0	0	2	0	0	2	0	0	2	2	0	2	0	0	2	0	0	2	2	0	2	0	0	2	0	0	2	0	0
GJ (25)	22	22	0	0	18	4	4	18	4	4	5	2	17	22	2	0	18	4	4	5	2	17	22	2	0	18	5	4	10	0	12	12	0	10
ICC (18)	27	27	0	0	18	0	9	14	0	13	23	9	4	21	2	6	14	0	13	23	9	4	21	2	6	19	1	8	18	0	9	20	0	7
L (24)	21	21	0	0	14	1	7	14	1	7	6	2	15	21	9	0	13	1	8	6	2	15	21	9	0	16	1	5	9	0	12	9	0	12
R (9)	9	9	0	0	8	0	1	8	0	1	1	0	8	4	0	5	8	0	1	6	0	3	6	0	3	1	0	8	9	0	0	9	0	0
RICC (10)	21	21	0	0	2	0	19	2	0	19	4	0	17	5	0	16	2	0	19	4	0	17	5	0	16	2	0	19	19	0	2	19	0	2
T (6)	6	6	0	0	5	0	1	5	0	1	1	0	5	4	1	2	5	0	1	1	0	5	4	1	2	3	0	3	6	0	0	6	0	0
UC (4)	0	0	0	0	0	3	0	0	3	0	0	4	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	0	0	0	0	0
Sum (158)	155	155	0	0	110	15	45	106	17	49	65	23	90	119	28	36	105	17	50	70	22	85	121	28	34	103	18	52	90	0	65	94	0	61

1) COMPARED TOOLS AND SETUP

Available static taint analysis tools were selected based on the study by Zhang et al. [4]: $FlowDroid_{IC3}$ [29], FlowDroid [29], Amandroid [30], DroidSafe [31], and DroidRA [32]. IC3 is obtained from the authors of [4]. In accordance with [4], DroidRA is used with FlowDroid, Amandroid, and DroidSafe, denoted by $DroidRA_F$, $DroidRA_A$, and $DroidRA_D$ respectively. The same tool options, taint source and sink definitions, and tool versions as the study [4] were used except for versions of DroidRA and FlowDroid (2.9.0). Also, the comparison includes tools targeting ICC: IccTA [33] and RAICC [34], [35]. In addition, it includes TaintDroid [36] and IntelliDroid [37], which leverage dynamic taint analysis.

Note that whereas the idea of the recording and reconstruction was initially realized in VTDroid [38], VTDroid was omitted from our evaluation. The decision is because VTDroid is specialized for specific flows, e.g., control dependencies and timing channels, which are not supported by the selected tools, including T-Recs.

The execution environment for T-Recs and the static analyzers is a ten-core (20 threads) 3.7GHz CPU and 128GB RAM. Devices of Zenfone 4 (Android 8.0.0) and Nexus 5 (Android 5.0.1) and an emulator of Nexus 9 (Android 8.0.0) were used for T-Recs to exemplify T-Recs' independency from specific Android devices and versions.

This section also involves TaintDroid with Nexus 4 (Android 4.3), the most popular and stable dynamic taint tracker for Android apps. In order to evaluate TaintDroid with DroidBench, TaintDroid was modified to support the taint sinks, which the original version of TaintDroid does not support (Appendix). Since TaintDroid does not have an app-exercise ability, a publicly-available hybrid analysis tool called IntelliDroid was used in combination with TaintDroid. IntelliDroid performs targeted execution and officially supports TaintDroid.

The exerciser was employed to exercise the apps for T-Recs automatically. As Section III-F explains, the exerciser requires a parameter that specifies the maximum number of exercise operations (Table 1) to be performed for an app on a device. The parameter value was determined based on a preliminary investigation. Each app was tested, and the number of exercise operations needed to trigger leaks in each app was obtained. The maximum was 27 for a case in the ICC category. Therefore, 30 was used as the maximum number of exercise operations in this evaluation.

2) DETECTION ACCURACY

Table 2 shows the result. The expected leak numbers are obtained from [25]. The result shows that only T-Recs is 100% accurate. The parser, the instrumentator, and the logger successfully processed all the apps, the exerciser automatically triggered all the leaks, and the reconstructor successfully detected all the leaks.

Notably, in ED, the instrumentator did not inject the logger into the *computePi()* method in an app called PI1 and successfully kept the method execution time within the threshold. In addition, the reconstructor detected *computePi()* as a method that does not affect the execution and skipped it, resulting in successful leak detection.

T-Recs' independence of the analysis environment helps T-Recs analyze apps in ED. One of the apps, for example, triggers a leak only when specific files exist on the device. Prior to the evaluation, we checked devices, Nexus 4 (Android 4.3), Nexus 5 (Android 5.0.1), Zenfone 3 (Android 6.0.1), Zenfone 4 (Android 8.0.0), Pixel 4 (Android 10), and Pixel 6 (Android 12). Among them, only Nexus 4 and 5 can trigger the leaks. Nexus 5 was included in this evaluation, and T-Recs successfully ran on the device and detected the leaks.

$FlowDroid_{IC3}$ and FlowDroid generate 15 and 17 FPs, respectively, in call-flow-related cases (AS, C, GJ, and L), control-flow-related cases (UC), and other cases (A and AL). UC is related to path sensitivity, which FlowDroid cannot consider. The dynamic taint trackers outperform the static taint trackers in this category. $FlowDroid_{IC3}$ and FlowDroid also produce some FNs. In particular, the tools produce nine and 13 FNs because of failure in intent tracking in ICC and 19 FNs in RICC because of failure to resolve reflective calls. The other static taint analysis tools, Amandroid, DroidSafe $DroidRA_F$, $DroidRA_A$, $DroidRA_D$, and IccTA, also produce a certain amount of FPs and FNs.

TaintDroid generates no FP, indicating that the tool is accurate. However, TaintDroid misses 65 leaks. IntelliDroid improves TaintDroid's result in three apps (four leaks) in GJ and ICC. Since some callbacks were successfully triggered, IntelliDroid should be adequate for more apps. The small number of improvements may be due to the quality of the tool, and increasing the quality may improve the result.

RAICC instruments none of the 158 apps. This is because the apps do not contain code targeted by RAICC. Since no leak detection is performed, the RAICC's result is excluded from Table 2. Section VII discusses RAICC further.

TABLE 3. Analysis time for the DroidBench apps.

Tool	Time
T-Recs	4 hours 58 minutes
FlowDroid$_{IC3}$	14 minutes
FlowDroid	4 minutes
Amandroid	47 minutes
DroidSafe	15 hours 37 minutes
DroidRA$_F$	18 hours 56 minutes
DroidRA$_A$	19 hours 56 minutes
DroidRA$_D$	33 hours 52 minutes
IccTA	1 hour 26 minutes
TaintDroid	36 minutes
IntelliDroid	59 minutes

3) ANALYSIS TIME

Table 3 shows the result. Each tool analyzed the apps one at a time. T-Recs' parser and instrumentator took 15 minutes, and the exerciser with the reconstructor took four hours and 43 minutes. T-Recs is the fifth slowest, but it is acceptable because it finishes within a reasonable time (one minute and 53 seconds per app). FlowDroid is the fastest, but the result would be different in real-world-app analysis because the benchmark apps have minimal code, and the static analysis time depends on the amount of code. On the other hand, T-Recs, TaintDroid, and IntelliDroid require an app execution time regardless of benchmarks or real-world apps. A result of real-world-app analysis is discussed in Section V-C4.

The analysis time of T-Recs can be shortened by improving the exerciser. The maximum number of exercise operations slightly influences the number of leaks detected and analysis time. If the maximum number of exercise operations is 40, T-Recs detects all the leaks and takes five hours and 13 minutes. It is 10% longer than the analysis with 30 as the maximum number of exercise operations. If the maximum number of exercise operations is 20, T-Recs fails to detect one leak in a case in the ICC category and takes four hours and 48 minutes. The analysis time is almost the same as one with 30 as the maximum number of exercise operations. Each analysis time is the average of three executions. On the other hand, T-Recs took 52 minutes in total with an ideal exerciser, which was manually created and contained a minimum set of operations to trigger all the leaks.

C. PRIVACY LEAK DETECTION IN POPULAR APPS 2016

This section compares T-Recs, FlowDroid, FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_F$, DroidRA$_A$, DroidRA$_D$, IccTA, TaintDroid, and IntelliDroid based on accuracy, time, and success rate for detecting privacy leaks in real-world apps. The tracking targets are those that the tools support: hardware identifiers (IMEI, IMSI, and ICCID), phone numbers, and location data.

1) COMPARED TOOLS AND SETUP

T-Recs uses the Pixel 3 with Android 9 and the computer explained in Section V-B1. Android 9 is the last version in which the hardware identifiers are accessible. For T-Recs and TaintDroid, each app is installed and launched on the Android devices, and each system waits for approximately 60 seconds and then uninstalls it. We decided not to exercise the apps based on the results of our preliminary experiment, indicating that apps in the dataset cause leaks by simple operations, such as starting an app. A one-hour timeout is used per app for each of T-Recs and the static analyzers.

Having flawless taint sink definitions for T-Recs and the static taint analyzers is challenging because there are numerous candidates, which are API methods that may cause leaks. Since taint sink definitions must be prepared to detect privacy leaks, they were created by ourselves. API methods of network-related libraries were chosen. Also, API methods that write data to transmit it to the network were selected. These sink definitions were used for T-Recs and the static taint analyzers, and it was believed that none of the tools is particularly advantageous to producing more TPs. However, our taint sink definitions cannot be used universally. Also, taint sink definitions vary depending on what code the analysts attempt to find and should be prepared by the analysts on their own. Therefore, it should be clear that this paper does not offer taint sink definitions (i.e., out-of-scope). On the other hand, TaintDroid's default sink definitions were used for TaintDroid and IntelliDroid.

2) DETECTION ACCURACY

Whereas establishing a ground truth is infeasible, correct leaks were obtained by manually searching the network dump for plaintexts (e.g., IMEI value 356000000000003 in Fig. 5) and names (e.g., IMEI) of the target information. We also searched for transformed data (e.g., XXxxxxx==) reported by T-Recs and TaintDroid. In the dynamic analysis, detecting no leak is correct if no leak occurred, and detecting a leak is correct if the leak occurred. Therefore, we only consider leaks occurring on both T-Recs' and TaintDroid's devices. Each alert of T-Recs and TaintDroid was compared with the network dump to verify that the leak occurred. If an alert includes transformed data, the app code was manually analyzed to confirm that the data contain the target information. The number of unique URLs in TaintDroid's alerts and the number of unique sink code locations in T-Recs' alerts were counted.

Table 4 shows the result. The expected #leaks indicates the number of unique URLs with that the sensitive information leaked. T-Recs does not generate FP for apps without leaks. Since the analyst needs to check only the 18 apps with leaks, the impact of the FPs is small, and we conclude that T-Recs is highly accurate. T-Recs has more FPs than TaintDroid because T-Recs' sink definition differs from TaintDroid. The conservative rules for API method calls explained in Section III-E4 may also be a factor. For the same reason, T-Recs generates more TPs than TaintDroid.

IntelliDroid only detects the three expected leaks in two apps, which are also detected by TaintDroid (Table 4). On the other hand, IntelliDroid misses many leaks that TaintDroid detects, demonstrating that introducing IntelliDroid into TaintDroid makes TaintDroid overlooks more leaks. Note that IntelliDroid finds a leak that does not occur in the

TABLE 4. Leak detection result. E indicates expected #leaks. ✗ indicates that the tool failed, and ✗$_{IC3}$ indicates that IC3 failed.

#App	E	T-Recs			TaintDroid			IntelliDroid			FlowDroid		FlowDroid$_{IC3}$		Amandroid		DroidSafe		DroidRA$_F$		DroidRA$_A$		DroidRA$_D$		IccTA	
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure
1	10	6	12	4	4	8	6	✗			✗		✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	8	5	3	3	4	0	4	0	0	8	6	2	✗		✗		✗		✗		✗		✗		✗	
1	5	5	4	0	0	0	5	0	0	5	0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	5	5	1	0	1	0	4	1	0	4	0	3	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	2	0	2	0	0	4	0	0	4	2	16	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	3	2	1	3	0	1	2	0	2	2	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	2	5	2	1	0	3	0	0	4	0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	4	13	0	3	0	1	✗			4	2	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	3	3	0	0	1	0	2	✗			3	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	3	3	9	0	3	0	0	0	0	3	1	4	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	2	0	0	2	2	0	0	0	0	2	1	0	✗$_{IC3}$		✗		✗		1	0	✗		✗		✗	
1	1	1	2	0	1	1	0	0	0	1	✗		✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	0	0	1	1	0	0	0	0	1	1	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	1	4	0	1	1	0	0	0	1	0	1	✗		✗		✗		✗		✗		✗		✗	
1	1	0	0	1	0	0	1	0	0	1	0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	1	0	0	1	3	0	0	0	1	0	1	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	1	0	0	1	0	0	✗			0	0	0	0	✗		✗		✗		✗		✗		✗	
1	1	1	2	0	0	0	1	✗			0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	✗			0	3	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	✗$_{IC3}$		✗		✗		0	1	✗		✗		✗	
1	0	✗			0	0	0	0	0	0	0	0	✗$_{IC3}$		0	0	✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	✗			0	0	✗		0	0	✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	3	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	✗		✗		0	1	✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		0	0
1	0	0	0	0	0	0	0	✗			0	0	0	0	✗		✗		✗		✗		✗		0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		0	0	✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	✗			0	1	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	✗			✗			0	0	0	0	✗		✗		✗		✗		✗		✗	
1	0	✗			0	0	0	0	0	0	0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		0	0
1	0	0	0	0	0	0	0	0	0	0	0	4	✗		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	✗			0	2	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗	
2	0	0	0	0	0	0	0	0	0	0	0	2	0	0	✗		✗		✗		✗		✗		0	2
2	0	0	0	0	0	0	0	✗			0	3	0	0	✗		✗		✗		✗		✗		✗	
2	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		✗		✗		✗	
3	0	0	0	0	0	0	0	0	0	0	✗		✗		0	0	✗		✗		✗		✗		0	0
3	0	0	0	0	✗			✗			0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
4	0	0	0	0	0	0	0	0	0	0	0	0	✗$_{IC3}$		0	0	✗		✗		✗		✗		✗	
4	0	0	0	0	0	0	0	0	0	0	✗		✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
8	0	0	0	0	0	0	0	✗			0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		✗	
16	0	0	0	0	0	0	0	0	0	0	0	0	✗$_{IC3}$		✗		✗		✗		✗		✗		✗	
96	59	43	57	16	27	14	32	3	0	56	20	58	0	0	0	0	0	0	1	2	0	0	0	0	0	4

environments of T-Recs and TaintDroid, showing its superiority over T-Recs and TaintDroid. However, this paper ignores the leak because improving code coverage of real-world apps is out-of-scope of this paper, as Section VI also discusses.

FlowDroid's alerts were also verified based on T-Recs' results because the two tools use the same sink definitions. In addition, taint sources suggested by each alert were compared with taint sources identified in the network dumps. We identified 20 TPs. On the other hand, as indicated by *unsure* in Table 4, the 58 alerts do not match with the network dumps. They are considered to be unsure leaks that could be either TP or FP because we had no resources for further high-cost verification. It was confirmed that T-Recs does not detect the unsure leaks because of the code coverage. All the unsure leaks are caused by codes outside the T-Recs' code coverage. The maximum, minimum, average, and median of the T-Recs' code coverages were 40.7%, 0.3%, 6.6%, and 4.3%, respectively. Note that 15 apps with zero code coverage were omitted from this calculation. In contrast, FlowDroid can analyze the entire code of each app, which is an advantage of static analysis. There is a trade-off between the coverage and accuracy, which is discussed in Section VI.

FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_A$, and DroidRA$_D$ detect no leaks. Note that the default definitions of taint sources and sinks are used for DroidSafe because changing the definitions requires modification of the source code of DroidSafe. Since DroidSafe with the default definitions fails to analyze all the apps, it would detect no leaks even if different source and sink definitions, such as the ones used by T-Recs and the other static analyzers, were used. Also, the developer clearly states that DroidSafe is unsuitable for analyzing Google Play apps [39]. DroidRA$_F$ finds one TP and two unsure leaks, which are also detected by FlowDroid. IccTA detects no TP and four unsure leaks. These seven tools are not very effective in analyzing real-world apps.

On the whole, FlowDroid misses many leaks that T-Recs and TaintDroid detect, and FlowDroid's recall is low. At the same time, FlowDroid generates 58 unsure alerts, suggesting high verification costs. Therefore, T-Recs and TaintDroid are more practical than FlowDroid for privacy leak detection. Also, the other tools generate almost no alerts and cannot be used for privacy leak detection dependably.

3) TRACKING ABILITY FOR ICC- AND REFLECTION-RELATED FLOWS

Static analysis can usually detect more leaks with higher FP rates than dynamic analysis. However, the result shows that the dynamic analyzers (i.e., T-Recs and TaintDroid) detect more leaks than the static analyzer (i.e., FlowDroid). T-Recs

TABLE 5. ICC- and reflection-related flows that we selected based on the DroidBench apps with that FlowDroid generates FNs.

Type	Description
1	*Activity.startActivity()* with a tainted argument
2	*Messenger.send()* with a tainted argument
3	Reflective call with a tainted argument
4	Reflective call with the tainted return value
5	Reflective call of a taint source

TABLE 6. #Apps and #codes in parentheses in which the five code types are found and whether T-Recs and FlowDroid detect the leaks caused by the five code types. The row "any" gives #apps and #codes in which at least one code type is found.

	T-Recs			FlowDroid		
Type	flow	TP	FP	TP	FN	incompleted
1	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
2	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
3	4 (8)	2 (5)	0 (6)	0 (0)	1 (3)	1 (2)
4	4 (8)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
5	7 (7)	5 (15)	0 (11)	0 (0)	4 (9)	1 (6)
any	10 (20)	6 (18)	0 (13)	0 (0)	5 (12)	1 (6)

detects 43 TPs; TaintDroid, 27 TPs; and FlowDroid, 20 TPs. One of the possible reasons is that FlowDroid failed to complete the analysis of apps in that the dynamic analyzers detect TPs. FlowDroid failed the analysis for two apps with one or more expected leaks (Table 4).

Another possible reason is the difference in information-flow-tracking abilities between FlowDroid and the other tools. Since FlowDroid mostly misses leaks in the DroidBench apps of ICC and RICC, we are focusing on the ICC and RICC cases. Selected code types are shown in Table 5 based on the DroidBench apps with that FlowDroid generates FNs (Table 2). T-Recs' reconstructor was modified to identify and count the occurrences of the five code types. The experiment was conducted once the overall analysis was completed (i.e., after the results in Table 4 were obtained). Since the logs obtained by the logger were kept, we only needed to re-execute the reconstructor. In other words, re-exercising the app is unnecessary when testing a new feature in the reconstructor, which is further discussed in Section VI.

Table 6 shows the number of apps and code points where the five code types are found by T-Recs (second column *flow*). It also shows the number of apps and code points where T-Recs detects leaks caused by the five code types (third and fourth columns). It also shows whether FlowDroid detects the TP leaks detected by T-Recs (fifth, sixth, and seventh columns). The result shows that four out of the five code types are found, and two of them cause leaks. Whereas six leaks of the third type and 11 of the fifth type are FP (i.e., the leaks are falsely detected), all the leak-detected apps are TP (i.e., no app is falsely detected by T-Recs). In contrast, FlowDroid fails to detect all of them. FlowDroid misses some of the leaks and fails to complete the analysis of some apps, as indicated by *incompleted*. Note that Table 6 excludes the result that FlowDroid detects none of the FP leaks detected by T-Recs. The results highlight the importance of tracking ICC- and reflection-related flows as the flows appear in the real-world apps as well as the DroidBench apps.

There can be other types of code with that FlowDroid generates FNs in the DroidBench apps. However, identifying the exact instructions preventing FlowDroid from tracking flows requires debugging FlowDroid, which is unfamiliar to us. Therefore, we focus only on ICC and reflection in this section and consider the choice to be sufficient to show how T-Recs detects leaks that FlowDroid misses.

TABLE 7. Analysis time for the privacy leak detection.

Tool	Time
T-Recs	3 hours 19 minutes
TaintDroid	2 hours 19 minutes
IntelliDroid	70 hours 6 minutes
FlowDroid	2 hours 34 minutes
FlowDroid_{IC3}	15 hours 33 minutes
Amandroid	82 hours 5 minutes
DroidSafe	20 hours 29 minutes
DroidRA_F	91 hours 32 minutes
DroidRA_A	95 hours 16 minutes
DroidRA_D	94 hours 55 minutes
IccTA	55 hours 39 minutes

4) ANALYSIS TIME

Table 7 shows the result. TaintDroid is the fastest, FlowDroid is second, and T-Recs is third. Their results are not largely different. They did not reach the timeout in analyzing any apps. In comparison, FlowDroid_{IC3} and the other tools took over 15 hours each. They failed many apps because of the timeout and are not suitable for the analysis of a set of real-world apps. For example, IC3 improves the performance of FlowDroid's ICC handling, but the result shows that IC3 cannot be finished in a reasonable time.

T-Recs' parser and instrumentator took 27 minutes; the app exercise, 138 minutes; and the reconstructor, 34 minutes. After the parser and the instrumentator process an app, the analyst can analyze the app to detect various information flows by reconfiguring and running the reconstructor and, if necessary, exercising the app on a device to acquire more code coverage. In other words, the parser and the instrumentator only need to be executed once for each app. Hence, reducing the analysis time for the parser and the instrumentator is a low priority. The two components have not been optimized in the current implementation by, for example, processing Smali files in parallel. Therefore, we simply determined to run the two components in 12 threads without conducting a further performance evaluation of them.

On the other hand, the reconstructor must be executed every time the analyst changes the configuration (e.g., taint source and sink definitions). Also, if the analyst needs more code coverage, re-trying the app exercise is necessary. Therefore, the rest of this section first discusses the analysis time taken by the reconstructor (Section V-C5), and then the analysis time for the app exercise is explained (Section V-C6).

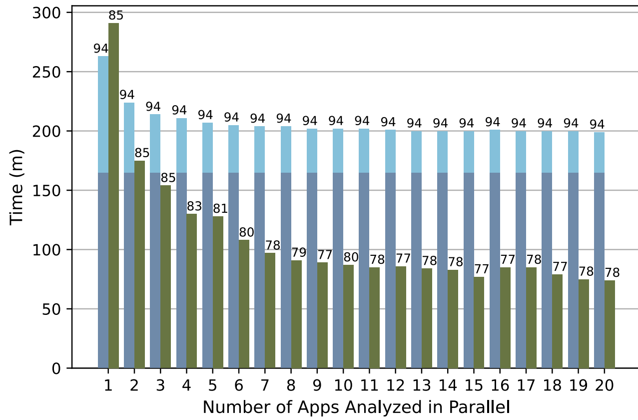


FIGURE 8. Analysis time of T-Recs and FlowDroid with different number of apps analyzed in parallel. The blue bars represent T-Recs' results, and the green bars represent FlowDroid's results. The labels on the bars show the number of successfully-analyzed apps.

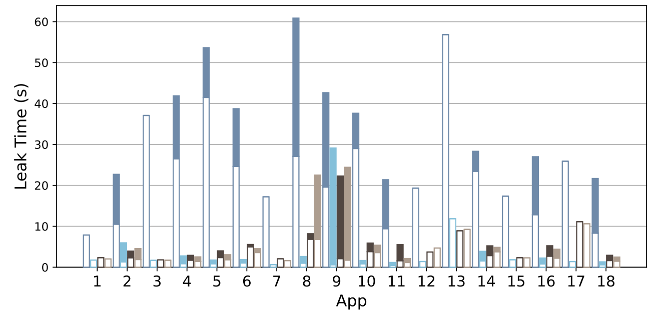


FIGURE 9. Time for apps to launch and cause leaks on Pixel 3 with and without T-Recs and Nexus 4 with and without TaintDroid, which are represented by the dark blue bar, the light blue bar, the dark brown bar, and the light brown bar from left to right, respectively.

5) T-Recs' AND FlowDroid's PARALLEL APP ANALYSIS TIME

The reconstructor's time depends on the number of apps analyzed in parallel, which is determined by the available RAM size of the computer used. A computer with 128GB RAM was used, and each tool's memory was fixed to 120GB by using a Docker container limited to 120GB RAM with disabling the swap. Since the computer's CPU has ten cores (20 threads), the reconstructor was executed by changing the number of apps analyzed in parallel from one to 20. FlowDroid was also tested for comparison. The analysis time may vary depending on the order of the apps to be analyzed. In this study, the tools analyzed the apps in alphabetical order by the app's name.

Fig. 8 shows the result. The blue bars show the analysis time taken by all phases of T-Recs. The dark blue parts represent the time for the parser, the instrumentator, and the app exercise. The light blue parts show the reconstructor's time. For example, 98 minutes for #apps = 1, 59 minutes for #apps = 2, and 34 minutes for #apps = 20, which is the shortest. The bars' labels represent the number of successfully-analyzed apps. The maximum is 94 because T-Recs fails two apps in phases prior to the reconstructor, and the reconstructor processes only 94 apps. On the other hand, the green bars indicate FlowDroid's result. For example, 291 minutes for #apps = 1, 175 minutes for #apps = 2, and 74 minutes for #apps = 20, which is the fastest. However, as the bars' labels show, the number of successfully-analyzed apps decreases as the number of apps analyzed in parallel increases. FlowDroid's maximum number of successfully-analyzed apps is 85 because FlowDroid fails 11 apps regardless of the RAM size. Hence, the maximum number of apps that FlowDroid can analyze in parallel without causing failure is three, and the time is 154 minutes.

For the results in Table 7, the number of apps analyzed in parallel was determined based on the maximum number that would not cause the analysis to fail due to lack of memory. T-Recs' reconstructor was executed in 20 threads, and FlowDroid was executed by analyzing three apps in parallel.

6) T-Recs' AND TaintDroid's APP-RUNTIME OVERHEADS

In dynamic analysis (i.e., T-Recs and TaintDroid), the app-runtime overhead affects the operation delay, which in turn makes the analysis time longer. If the app exercise time is too short, the app may be terminated before a leak occurs, and the leak would not be detected. Therefore, we investigated the app-runtime overheads of T-Recs and TaintDroid. We measured the time for apps to launch and cause the same leaks with and without the tools. Pixel 3 was used with and without T-Recs, and Nexus 4 was used with and without TaintDroid to compare with T-Recs.

Fig. 9 shows the leak time of the 18 apps that cause more than one leak. Pixel 3 with and without T-Recs and Nexus 4 with and without TaintDroid are represented by the four bars from left to right for each app. A transparent part of the bar indicates the time for the first leak in the app, and the colored part indicates the time for the last leak in the app (i.e., the time for occurring all the expected leaks in the app shown in Table 4). In a total of the 18 apps, T-Recs took 578.1 seconds; Pixel 3 without T-Recs, 74.9 seconds; TaintDroid, 104.0 seconds; and Nexus4 without TaintDroid, 113.1 seconds. T-Recs is 7.7 times slower than the original Pixel 3 and 5.6 times slower than TaintDroid. Using TaintDroid is faster than not using TaintDroid, indicating that TaintDroid has no overhead, and the result is consistent with the original paper [10], reporting that TaintDroid has negligible perceived latency with real-world apps. T-Recs took the longest time for app number 8, which was 60.9 seconds. TaintDroid took the longest time for app number 9, which was 22.3 seconds. The T-Recs' longest time is 2.7 times larger than TaintDroid.

The app-runtime overhead of T-Recs is caused due to the instrumentation, which is a trade-off for the tool's device independency. Preparing Android devices is easier for T-Recs than TaintDroid. Hence, T-Recs users can run the app exercising parallelly on multiple devices to shorten the analysis time. For example, assuming that the analyst sets the same exercising time for every app (i.e., 61 seconds per app for T-Recs and 23 seconds per app for TaintDroid), T-Recs' time would be shorter than TaintDroid if three or more devices were used in parallel for T-Recs. Hence, the app-runtime overhead of T-Recs is considered to be acceptable.

TABLE 8. Time (seconds) for apps to be installed and uninstalled on the devices with and without the tools.

	T-Recs	Pixel 3	TaintDroid	Nexus 4
Install	497	396	1714	1571
Uninstall	36	36	119	114

We also measured the time for apps to be installed and uninstalled, which affects the analysis time. Table 8 shows the total time for 90 apps with T-Recs, Pixel 3 without T-Recs, TaintDroid, and Nexus 4 without TaintDroid. We excluded two apps that T-Recs failed to complete the analysis and four apps that TaintDroid failed. The uninstallation times are the same for Pixel with and without T-Recs. It barely changes for Nexus 4 with and without TaintDroid. On the other hand, for the installation times, although using T-Recs is 26% slower than not using T-Recs, the T-Recs' result is 3.4 times faster than TaintDroid. This is mainly because T-Recs' device, Pixel 3, has a higher processing performance than TaintDroid's Nexus 4. The result demonstrates the device-independency advantage of T-Recs, in which the analyst can use new smartphones with high-performing processors.

7) ANALYSIS SUCCESS RATE

Table 9 shows the results. T-Recs' success rate is 98%, the highest among the compared tools. It failed one app in the instrumentation phase and one app in the installation phase. No type-conflict error occurred during app execution, and the success rate of the logger is 100%. The reconstructor also succeeded in detecting information flows.

FlowDroid's success rate is 89%. It stopped during the analysis due to some runtime errors with 11 apps. All failures occurred in the call graph construction phase before the flow detection process. These failures occur regardless of the taint source and sink definitions. FlowDroid$_{IC3}$ failed with 67 apps, mostly due to IC3 failures (e.g., exceeding the timeout). TaintDroid failed to install four apps due to incompatible SDK versions. TaintDroid uses Android 4.3 at the latest, and all apps that do not support this version cannot be analyzed. IntelliDroid's success rate is 75%, and the other tools' success rates are diminutive.

D. SUCCESS RATE OF ESSENTIAL PHASES IN VARIED DATASET

This section focuses on the tools' essential phases that are independent of tracked data to obtain the upper bounds of the tools' analysis success rate in general. Section V-D1 explains the essential phases, which vary from tool to tool, and Section V-D2 presents the result.

1) COMPARED TOOLS AND SETUP

Since IntelliDroid, FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_F$, DroidRA$_A$, DroidRA$_D$, and IccTA detect almost no leaks (Table 4), and their success rates are less than 80%

TABLE 9. #Apps successfully analyzed, #apps failed, and the analysis success rate for each tool in the privacy leak detection.

Tool	#apps succeeded	#apps failed	Success Rate
T-Recs	94	2	98%
TaintDroid	92	4	96%
IntelliDroid	72	24	75%
FlowDroid	85	11	89%
FlowDroid$_{IC3}$	29	67	30%
Amandroid	12	84	13%
DroidSafe	0	96	0%
DroidRA$_F$	4	92	4%
DroidRA$_A$	0	96	0%
DroidRA$_D$	0	96	0%
IccTA	8	88	8%

(Table 9), this section excludes them. Hence, this section compares T-Recs, FlowDroid, and TaintDroid. They may fail for reasons other than those described in Section V-C7. However, hidden errors could not be detected without knowing the tools' details. Also, errors in information flow tracking could depend on tracked data. In dynamic analysis, app exercise also depends on data being tracked. Therefore, this section focuses on the tools' essential phases, independent of taint source and sink definitions.

Since T-Recs' instrumentation and app-installation failed (Section V-C7), the percentage of successfully instrumented and installed apps was examined. The device used was Pixel 6 (Android 12), which was the latest device available at the paper was written and is much newer than TaintDroid's Nexus 4, highlighting the device-independency advantage of T-Recs. Considering that FlowDroid failed regardless of the taint source and sink definitions (Section V-C7), FlowDroid was executed without taint source and sink definitions to examine the percentage of analysis failures that occurred in the call graph construction phase. No timeout was used. Since TaintDroid's app-installation failed (Section V-C7), the percentage of apps that are successfully installed to TaintDroid was examined. TaintDroid failed because of the apps' supported SDK versions, which can be acquired by investigating the app files, but other factors may cause the installation failures. Therefore, the apps were actually installed one by one to TaintDroid without launching them.

2) RESULTS

The rate of successfully-processed apps for each Android codename is shown in Fig. 10. The Android codename (e.g., 1.0, 1.1, C, and D) indicates the minimum SDK version supported by an app configured by the app developers. The result shows that T-Recs evenly supports the 16 codenames (i.e., 28 SDK versions). T-Recs achieves at least 86.3% for any version, and the average is 96.6%. FlowDroid also achieves at least 87.5% for any version, and the average is 95.6%. On the other hand, TaintDroid fails for apps newer than version J (i.e., Android 4.3), and the average is 62.5%. TaintDroid is not applicable for apps developed for Android 4.4 or later

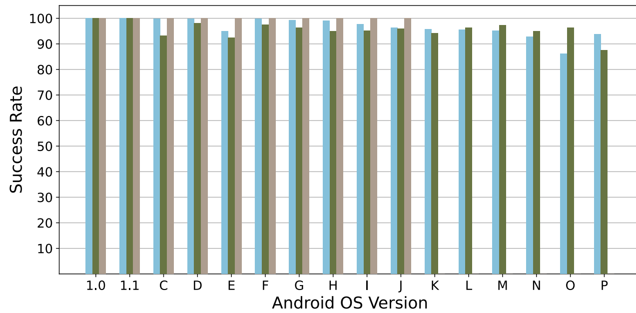


FIGURE 10. Success rates for the varied dataset. T-Recs is left blue bars, FlowDroid is center green bars, and TaintDroid is right brown bars.

versions after 2013. In this evaluation, T-Recs took 276 hours; TaintDroid, 116 hours; and FlowDroid, 108 hours.

E. ID LEAK DETECTION IN POPULAR APPS 2021

Recently-published popular apps were used to consider a more up-to-date situation than Section V-C because access to the hardware identifiers has been restricted since Android 10.

1) COMPARED TOOLS AND SETUP

This section compares T-Recs, FlowDroid, FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_F$, DroidRA$_A$, DroidRA$_D$, and IccTA. It omits TaintDroid and IntelliDroid because TaintDroid's success rate is low for newer apps (Section V-D2). The app set used in this section contains 139 apps (88%) with minimum SDK versions 19 (i.e., Android 4.4) and greater, which cannot be analyzed by TaintDroid and IntelliDroid. As a result, T-Recs was only compared with static analysis, which was disadvantageous for T-Recs because of the coverage difference, and static analysis should detect more leaks. However, showing that T-Recs can detect leaks in newer apps is still valuable.

T-Recs launched and waited for each app on Pixel 3 for two minutes with no exercise operation based on the assumption that ID leaks swiftly occur as well as the evaluation in Section V-C. A one-hour timeout is used per app for each tool.

Target taint sources are identifiers, including Build. SERIAL, MAC address, Android ID, Google Advertising ID, Instance ID, and Globally-Unique ID [40] in addition to the taint sources used in the privacy leak evaluation (Section V-C). Taint sinks are the same as the privacy leak evaluation. As described in Section V-C2, this evaluation also uses the default definitions of taint sources and sinks for DroidSafe.

2) RESULTS

Table 10 shows the number of apps and leaks detected by the tools. T-Recs and FlowDroid detect leaks in 55 and 60 apps, respectively. FlowDroid$_{IC3}$ detects six leaks in four apps, which are also uncovered by FlowDroid. The other tools generate no alerts.

T-Recs detects leaks in fewer apps than FlowDroid, as expected. In contrast, T-Recs detects 400 leaks, which is larger than FlowDroid's result. Based on the result in Table 4, T-Recs generates numerous FP leak alerts as well as TPs for leak-causing apps. Therefore, some of the 400 leaks could be FP. At the same time, however, Table 4 shows that T-Recs produces no FP for no-leak-causing apps. Hence, all 55 apps detected by T-Recs in Table 10 are likely TP. Interestingly, the detected apps and leaks overlap slightly between the tools. The result shows that T-Recs can track information flows and detect ID leaks, primarily undetected by FlowDroid, in recently-developed apps from Google Play.

On the other hand, Table 10 shows that FlowDroid detects 206 leaks that T-Recs does not detect (i.e., among 214 leaks detected by FlowDroid, the overlap is only eight leaks). It was confirmed that 205 leaks are caused by codes not covered by T-Recs. In contrast, the other leak is caused by codes within T-Recs' code coverage. However, T-Recs does not detect the leak because of the leak's sink. This sink is a writer that outputs data not to the network but to a file. Hence, detecting the leak is FP. T-Recs identifies that the information flow's destination is a file and does not detect it as a leak. Overall, FlowDroid detects leaks undetected by T-Recs principally because of the difference in their code coverages. The maximum, minimum, average, and median of the T-Recs' code coverages were 28.9%, 0.05%, 6.1%, and 4.6%, respectively. Note that 16 apps with zero code coverage were excluded from this calculation. In contrast, FlowDroid can analyze the whole code of each app, which is a trade-off for accuracy, and Section VI further discusses this.

Table 11 shows the analysis time taken by each tool. T-Recs took 14 hours and 35 minutes. The parser and the instrumentator took five hours and 13 minutes; the app exercise, seven hours and 42 minutes; and the reconstructor, one hour and 40 minutes.

3) TRACKING ABILITY FOR ICC- AND REFLECTION-RELATED FLOWS

Similar to Table 6 in the privacy leak evaluation (Section V-C3), Table 12 shows T-Recs' and FlowDroid's results for the five types of code related to ICC and reflection. It shows the number of apps and code points where the five code types are found (second column *flow*) and where the related leaks are detected (third column *leak*). It also shows whether FlowDroid detects the leaks (fourth, fifth, and sixth). The result shows that all five code types are found, and also leaks related to four code types are found. In contrast, FlowDroid detects none of the leaks. FlowDroid fails to complete the analysis for some of the apps, as indicated by *incompleted*. The results show one of the reasons why T-Recs' and FlowDroid's results barely overlap. The results emphasize the importance of tracking ICC- and reflection-related flows as the flows appear in the recently-published real-world apps.

As well as Section V-C3, this experiment was conducted after the results in Table 10 were obtained, and re-exercising the app was unnecessary. It is also discussed in Section VI.

TABLE 10. #Apps and #leaks in parentheses detected in the apps from 2021. "Overlap" indicates #apps and #leaks detected by both T-Recs and the other tool.

	T-Recs	FlowDroid	FlowDroid_{IC3}	Amandroid	DroidSafe	DroidRA_F	DroidRA_A	DroidRA_D	IccTA
	55 (400)	60 (214)	4 (6)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
Overlap	-	3 (8)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)

TABLE 11. Analysis time for the ID leak detection.

Tool	Time
T-Recs	14 hours 35 minutes
FlowDroid	19 hours 58 minutes
FlowDroid_{IC3}	22 hours 18 minutes
Amandroid	122 hours 33 minutes
DroidSafe	4 hours 33 minutes
DroidRA_F	149 hours 44 minutes
DroidRA_A	149 hours 45 minutes
DroidRA_D	149 hours 45 minutes
IccTA	42 hours 1 minute

TABLE 12. #Apps and #codes in parentheses in which the five code types are found and whether FlowDroid detects the leaks caused by the five code types. The row "any" gives #apps and #codes in which at least one code type is found.

Type	T-Recs flow	T-Recs leak	FlowDroid positive	FlowDroid negative	FlowDroid incompleted
1	1 (2)	1 (1)	0 (0)	1 (1)	0 (0)
2	2 (2)	0 (0)	0 (0)	0 (0)	0 (0)
3	40 (95)	25 (89)	0 (0)	20 (71)	5 (18)
4	43 (177)	16 (59)	0 (0)	12 (40)	4 (19)
5	22 (24)	6 (21)	0 (0)	4 (6)	2 (15)
any	52 (223)	29 (110)	0 (0)	24 (83)	5 (27)

F. ETHICAL CONSIDERATIONS

In our evaluation, the experiments were carried out only by us. We used Android devices bought and prepared for the experiments and used no actual personal information. We consider respect for app publishers. The bytecode instrumentation targets only code that does not change any data sent to remote servers so as not to affect the publishers' properties. Additionally, the results were carefully used only to evaluate the tools and were not used for other purposes.

VI. DISCUSSION

As mentioned in Section V-C3 and Section V-E3, T-Recs can re-execute the taint analysis (i.e., the reconstructor) without the app exercise after a new feature is added to the taint analyzer. In the privacy leak detection (Section V-C4), the reconstructor took 34 minutes, which is 17% of the whole. The reconstructor took one hour and 40 minutes, which is 11% of the whole in the ID leak detection (Section V-E2). Note that these times were measured after the new feature (i.e., the ICC- and reflection-related-flow counter) had been added. A large amount of time is saved because T-Recs' analysis time depends mainly on app exercise time. As another example, in the DroidBench evaluation (Section V-B3), the exerciser took four hours and 43 minutes (i.e., 95% of the whole). In addition, the app exercise time can be extended when the analyst targets other than privacy leaks because this paper currently focuses only on privacy leaks, which tend to take a short time to occur. Therefore, being able to re-execute the taint analysis without running the app exercise is a significant advantage of T-Recs.

The evaluation shows that T-Recs has higher usability in terms of ease of setup than TaintDroid. TaintDroid is only available on Nexus 4 or older devices. Accessibility of the devices is extremely poor, and also, devices have a limited lifespan due to, for example, battery swelling. T-Recs, on the other hand, allows analysts to freely select devices such as Pixel3 and Pixel6, depending on each evaluation's environmental requirements (e.g., Android OS version). These devices are readily available, and the analysis can be started speedily by simply connecting them to a computer.

In the evaluation, privacy leaks are the main target and can be easily triggered. The core of this paper is to perform dynamic taint analysis outside the Android device, and improving code coverage of real-world apps is out-of-scope. As long as there is a code coverage problem with dynamic analysis, it is unrealistic to replace static analysis with dynamic analysis in some cases, and the analyses should be selected according to the user's purpose. Static analysis should be used in exchange for higher verification costs when code coverage is a priority. T-Recs should be used when low verification cost (i.e., accuracy) is a priority.

In the DroidBench evaluation, the categories covered by the compared tools [4] were selected, and others, such as native code and inter-app communication, are out-of-scope. We believe that T-Recs can be used in combination with existing tools, such as JN-SAF [41] for summarizing flows in native code because T-Recs performs the taint analysis on the server.

Our manual analysis of the network dumps explained in Section V-C2 is limited, which could affect the accuracy of the data in Table 4. We only searched the network dumps for plain texts of the target information, their names, and transformed data found by T-Recs and TaintDroid. Therefore, some leaks could be missed, for example, leaks caused by an app performing complex encryption on IMEI without being detected by the tools. Such mistakes affect the following. First, there would be more FNs in T-Recs' and TaintDroid's results. However, the impact is considered to be small because modifying both tools' results by the same amount does not change the conclusion that T-Recs generates fewer FNs than TaintDroid. Second, if a missed leak had been counted as the unsure in FlowDroid's result, the number of unsure leaks would decrease, and the number of TPs would increase. However, detecting such encrypted leaks requires reverse engineering of the app. The cost is high, which is consistent

TABLE 13. Taint sinks and corresponding file paths we modified.

Sink	Path
android.telephony.SmsManager: void sendTextMessage(java.lang.String, java.lang.String, java.lang.String, android.app.PendingIntent, android.app.PendingIntent)	frameworks/opt/telephony/src/java/android/telephony/SmsManager.java
android.util.Log: int i(java.lang.String, java.lang.String) android.util.Log: int e(java.lang.String, java.lang.String) android.util.Log: int v(java.lang.String, java.lang.String) android.util.Log: int d(java.lang.String, java.lang.String)	frameworks/base/core/java/android/util/Log.java
java.lang.ProcessBuilder: java.lang.Process start()	libcore/luni/src/main/java/java/lang/ProcessBuilder.java
android.app.Activity: void startActivityForResult(android.content.Intent, int) android.app.Activity: void startActivity(android.content.Intent) android.app.Activity: void setResult(Iandroid.content.Intent)	frameworks/base/core/java/android/app/Activity.java
java.net.URL: java.net.URLConnection openConnection()	libcore/luni/src/main/java/java/net/URL.java
android.content.ContextWrapper: void sendBroadcast(android.content.Intent)	frameworks/base/core/java/android/content/ContextWrapper.java

with our argument that the verification cost of the unsure leaks is high in Section V-C2. Therefore, the effect of missing leaks is considered to be acceptable.

Apps may detect code rewriting and stop running to protect the app developers and users [42]. When analyzing such apps, the logger must be integrated into the Android OS instead of the app bytecode instrumentation. As a result, the logger would depend on the Android OS version, the same as the existing dynamic analysis systems. However, compared to implementing the taint logic itself, implementing only the logger would be less expensive and more practical.

Whereas TaintDroid performs variable-, method-, file-, and message-level tracking, T-Recs does not track inter-app messages and does not keep tracking tainted content in a file across different runs. Although this was not a problem in the evaluation, it could depend on the information being tracked. As well as TaintDroid, T-Recs disregards implicit flows [43].

VII. RELATED WORK

Various tools of static taint analysis for Android apps have been developed and assessed in the community [3], [4], [44]. Mordahl et al. [44] examined configurations in FlowDroid and DroidSafe. Pauck et al. [3] evaluated static taint trackers: Amandroid, DIALDroid [45], DidFail [46], DroidSafe, FlowDroid, and IccTA. They excluded unavailable and unsatisfied tools, such as SCanDroid [47] and DroidInfer [48], for competitive comparison. Zhang et al. [4] compared FlowDroid combined with IccTA, Amandroid, and DroidSafe under the same setup. They also included DroidRA, an instrumentation-based approach targeting reflective calls and used in combination with FlowDroid, Amandroid, and DroidSafe. We opted to follow the study and used the same tools and configuration parameters because the exact set of used benchmark applications and the answers are available [25].

RAICC [34] is one of the latest tools targeting ICC. Specifically, RAICC targets "atypical ICC methods", which allow to perform an ICC while it is not its primary purpose [34]. Since the 158 apps in DroidBench do not contain "atypical ICC methods", RAICC instruments none of the apps in the evaluation (Section V-B2). We also contacted its developers, and they confirmed that it is intended for RAICC to instrument no DroidBench apps. On the other hand, this paper mainly focuses on addressing the inaccuracy of current taint analyzers against DroidBench, which contains more common cases than "atypical ICC methods". Also, Barros et al. [49] developed a static analysis technique for handling ICC and reflective calls precisely. Their approach is implemented for Java and requires the target apps' source code. However, when analyzing apps from Google Play or third-party markets, their source code is not usually available.

As native code is being more frequently used in apps, researchers have been developing new static analysis techniques targeting native code, such as JN-SAF [41] and JuCify [50]. JuCify unifies call graphs of native code and bytecode, and the result can be used by FlowDroid and other static analyzers that do not support native code. Also, CTAN [51] further improves JN-SAF. As Section VI explains, native code is out-of-scope of this paper.

There are more tools [22], [52], [53], [54], which perform the bytecode-level dynamic taint analysis for Android apps other than TaintDroid. Although, in comparison with static taint analyzers, dynamic taint trackers have been barely reviewed in the community except for TaintDroid. We obtained the taint tracking module of ARTist from the authors. However, the authors mentioned that the module is aged and requires quite some adoptions to be used with ARTist. Therefore, the tool was not used in our evaluation. We also obtained TaintMan from the authors. However, we encountered difficulties in deploying the tool and decided to omit the tool. Since native code is out-of-scope, OS-level trackers [55], [56] were excluded. For these reasons, only TaintDroid was chosen for our evaluation.

Recently, researchers developed hybrid analysis techniques such as targeted execution [13] and program slicing [57], which can assist taint trackers in detecting more leaks. IntelliDroid performs targeted execution, which is efficient when attempting to run a specific code path. However,

it depends on static analysis and inherits the drawbacks of inaccurate models. It was evaluated in our DroidBench evaluation in Section V-B2, and its result only slightly increased from TaintDroid. Besides IntelliDroid, Harvester [57] can improve TaintDroid by triggering malicious code. However, Harvester was omitted because the user must coordinate its target logging points (i.e., not wholly automatic), and also, Harvester is not publicly available.

VIII. CONCLUSION

This paper presents a usable dynamic taint tracker called T-Recs, detecting information flows by recording and reconstructing the app execution. T-Recs addresses the limitations of current dynamic trackers, which are the dependency on the analysis environments and the re-analysis cost. T-Recs was implemented and evaluated with 158 apps from DroidBench, 96 and 158 popular apps from Google Play, and SDK-version-varied apps randomly collected from Google Play and Anzhi. The results show that T-Recs is making steady progress. Future work includes the evaluation of T-Recs with benchmarks excluded in this paper. T-Recs is going to be applied to real-world apps to uncover apps' suspicious behaviors that have not been recognized yet. T-Recs has been made publicly available at https://github.com/SaitoLab-Nitech/T-Recs.

APPENDIX.
TaintDroid MODIFICATION

TaintDroid only supports HTTP/HTTPS transmission as the taint sinks by default. Six files in the source code of TaintDroid were modified to support taint sinks in the DroidBench apps. Table 13 shows the added taint sinks and corresponding file paths. For more details, the modified code is available from us upon reasonable request.

REFERENCES

[1] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2020, pp. 1106–1120.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Edinburgh, U.K., Jun. 2014, pp. 259–269.

[3] F. Pauck, E. Bodden, and H. Wehrheim, "Do Android taint analysis tools keep their promises?" in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Lake Buena Vista, FL, USA, Oct. 2018, pp. 331–341.

[4] J. Zhang, Y. Wang, L. Qiu, and J. Rubin, "Analyzing Android taint analysis tools: FlowDroid, amandroid, and DroidSafe," *IEEE Trans. Softw. Eng.*, vol. 48, no. 10, pp. 4014–4040, Oct. 2022.

[5] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," *ACM Trans. Privacy Secur.*, vol. 21, no. 3, pp. 1–32, Apr. 2018.

[6] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2015, p. 110, doi: 10.14722/ndss.2015.23089.

[7] Secure Software Engineering. *DroidBench 3.0*. Accessed: Oct. 3, 2021. [Online]. Available: https://github.com/secure-software-engineering/DroidBench/tree/develop

[8] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. IEEE/ACM IEEE Int. Conf. Softw. Eng.*, Florence, Italy, May 2015, pp. 280–291.

[9] D. Octeau, D. Luchaup, and M. Dering, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proc. Int. Conf. Softw. Eng.*, Florence, Italy, May 2015, pp. 77–88.

[10] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Vancouver, BC, Canada, Oct. 2010, pp. 393–407.

[11] B. Reaves, J. Bowers, S. A. Gorski, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. Butler, W. Enck, and P. Traynor, "*droid: Assessment and evaluation of Android application analysis tools," *ACM Comput. Surv.*, vol. 49, no. 3, pp. 1–30, Oct. 2016.

[12] H. Inayoshi, S. Kakei, and S. Saito, "Plug and analyze: Usable dynamic taint tracker for Android apps," in *Proc. IEEE 22nd Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Limassol, Cyprus, Oct. 2022, pp. 24–34.

[13] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2016, pp. 21–24, doi: 10.14722/ndss.2016.23118.

[14] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of Android apps," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Saarbrücken, Germany, Jul. 2016, pp. 318–329.

[15] X. Sun, L. Li, T. F. Bissyandé, J. Klein, D. Octeau, and J. Grundy, "Taming reflection: An essential step toward whole-program analysis of Android apps," *ACM Trans. Softw. Eng. Methodology*, vol. 30, no. 3, pp. 1–36, Apr. 2021.

[16] *Anzhi*. Accessed: Oct. 10, 2021. [Online]. Available: http://www.anzhi.com

[17] Google. *UI/Application Exerciser Monkey*. Accessed: Sep. 20, 2022. [Online]. Available: https://developer.android.com/studio/test/monkey

[18] V. Balachandran, D. J. Tan, and V. L. Thing, "Control flow obfuscation for Android applications," *Comput. Secur.*, vol. 61, pp. 72–93, Aug. 2016.

[19] Google. *Enable Multidex for Apps With Over 64K Methods*. Accessed: Sep. 21, 2022. [Online]. Available: https://developer.android.com/studio/build/multidex

[20] Google. *Dalvik Bytecode*. Accessed: Sep. 21, 2022. [Online]. Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

[21] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2015, pp. 1–15, doi: 10.14722/ndss.2015.23140.

[22] J. Schüette, A. Küechler, and D. Titze, "Practical application-level dynamic taint analysis of Android apps," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, Sydney, NSW, Australia, Aug. 2017, pp. 17–24.

[23] S. Arzt and E. Bodden, "StubDroid: Automatic inference of precise data-flow summaries for the Android framework," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, Austin, TX, USA, May 2016, pp. 725–735.

[24] *Apktool*. Accessed: Sep. 21, 2022. [Online]. Available: https://ibotpeaches.github.io/Apktool/

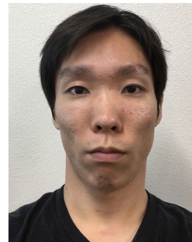
[25] J. Zhang, Y. Wang, L. Qiu, and J. Rubin. *Supplementary Materials*. Accessed: Oct. 3, 2021. [Online]. Available: https://resess.github.io/artifacts/StaticTaint/index

[26] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2017, pp. 1–15, doi: 10.14722/ndss.2017.23465.

[27] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, Austin, TX, USA, May 2016, pp. 468–471.

[28] Google. *AndroidX*. Accessed: Sep. 21, 2022. [Online]. Available: https://developer.android.com/jetpack/androidx

[29] Secure Software Engineering. *FlowDroid*. Accessed: Sep. 29, 2021. [Online]. Available: https://github.com/secure-software-engineering/FlowDroid
[30] Argus Group. *Argus-SAF*. Accessed: Feb. 13, 2022. [Online]. Available: https://github.com/arguslab/Argus-SAF
[31] MIT-PAC. *Droidsafe-SRC*. Accessed: Feb. 20, 2022. [Online]. Available: https://github.com/MIT-PAC/droidsafe-src
[32] SerVal Research Group. *DroidRA*. Accessed: Nov. 29, 2022. [Online]. Available: https://github.com/serval-snt-uni-lu/DroidRA
[33] L. Li. *IccTA*. Accessed: Dec. 2, 2022. [Online]. Available: https://github.com/lilicoding/soot-infoflow-android-iccta
[34] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein, "RAICC: Revealing atypical inter-component communication in Android apps," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Madrid, ES, USA, May 2021, pp. 1398–1409.
[35] J. Samhi. *RAICC*. Accessed: Nov. 29, 2022. [Online]. Available: https://github.com/JordanSamhi/RAICC
[36] *TaintDroid*. Accessed: Feb. 9, 2022. [Online]. Available: https://github.com/TaintDroid
[37] M. Wong. *IntelliDroid*. Accessed: Feb. 20, 2022. [Online]. Available: https://github.com/miwong/IntelliDroid
[38] H. Inayoshi, S. Kakei, E. Takimoto, K. Mouri, and S. Saito, "VTDroid: Value-based tracking for overcoming anti-taint-analysis techniques in Android apps," in *Proc. 16th Int. Conf. Availability, Rel. Secur.*, Vienna, Austria, Aug. 2021, pp. 1–6.
[39] MIT-PAC. *Droidsafe-src*. Accessed: Feb. 20, 2022. [Online]. Available: https://mit-pac.github.io/droidsafe-src
[40] Google. *Best Practices for Unique Identifiers*. Accessed: Sep. 22, 2022. [Online]. Available: https://developer.android.com/training/articles/user-data-ids
[41] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, ONT, Canada, Oct. 2018, pp. 1137–1150.
[42] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame, "You shall not repackage! Demystifying anti-repackaging on android," *Comput. Secur.*, vol. 103, Apr. 2021, Art. no. 102181.
[43] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Proc. 4th Int. Conf. Inform. Syst. Secur. (ICISS)*, Hyderabad, Andhra Pradesh, India, Dec. 2008, pp. 56–70.
[44] A. Mordahl and S. Wei, "The impact of tool configuration spaces on the evaluation of configurable taint analysis for android," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2021, pp. 466–477.
[45] A. Bosu, F. Liu, D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Abu Dhabi, UAE, Apr. 2017, pp. 71–85.
[46] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proc. 3rd ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, Edinburgh, U.K., Jun. 2014, pp. 1–6.
[47] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android applications," Dept. Comput. Sci., Univ. Maryland, College Park, MD, USA, Tech. Rep. CS-TR-4991, Nov. 2009. [Online]. Available: http://hdl.handle.net/1903/11847
[48] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for Android," in *Proc. ISSTA*, Baltimore, MD, USA, Jul. 2015, pp. 106–117.
[49] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving Java reflection and Android intents (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Lincoln, NE, USA, Nov. 2015, pp. 669–679.
[50] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "JuCify: A step towards Android code unification for enhanced static analysis," in *Proc. 44th Int. Conf. Softw. Eng.*, Pittsburgh, PA, USA, May 2022, pp. 1232–1244.
[51] S. B. Andarzian and B. T. Ladani, "Compositional taint analysis of native codes for security vetting of Android applications," in *Proc. 10th Int. Conf. Comput. Knowl. Eng. (ICCKE)*, Mashhad, Iran, Oct. 2020, pp. 567–572.
[52] M. Sun, T. Wei, and J. C. S. Lui, "Taintart: A practical multi-level information-flow tracking system for Android runtime," in *Proc. ACM Sigsac Conf.*, Vienna, Austria, Oct. 2016, pp. 331–342.
[53] M. Backes, S. Bugiel, O. Schranz, P. Von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Paris, France, Apr. 2017, pp. 481–495.
[54] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "TaintMan: An ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 1, pp. 209–222, Jan. 2020.
[55] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. 21st USENIX Secur. Symp.*, Bellevue, WA, USA, Aug. 2012, pp. 569–584.
[56] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. S. Chan, "NDroid: Toward tracking information flows across multiple Android contexts," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 3, pp. 814–828, Mar. 2019.
[57] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in Android applications that feature anti-analysis techniques," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2016, pp. 1–15, doi: 10.14722/ndss.2016.23066.



HIROKI INAYOSHI received the B.E. and M.E. degrees from the Nagoya Institute of Technology, Japan, in 2018 and 2021, respectively. He has been a Graduate Student with the Department of Computer Science, Nagoya Institute of Technology, since 2021. His research interests include dynamic program analysis, mobile security, and privacy. He is a member of IPSJ.

SHOHEI KAKEI received the B.E. and M.E. degrees from Gifu University, Japan, in 2011 and 2013, respectively, and the Ph.D. degree from Kobe University, Japan, in 2019. He has been an Assistant Professor with the Department of Computer Science, Nagoya Institute of Technology, Japan, since 2019. His current research interests include digital forensics, blockchain technology, and information security. He is a member of IEICE and IPSJ.

SHOICHI SAITO received the B.S. and M.E. degrees in engineering from Ritsumeikan University, in 1993 and 1995, respectively, and the Dr.Eng. degree, in 2000. He became a Research Associate at the Department of Computer and Communication Sciences, Wakayama University, in 1998. He was an Assistant Professor, in 2003, and an Associate Professor, in 2005. He was an Associate Professor with the Nagoya Institute of Technology, in 2006, where he has been a Professor, since 2016. His research interests include operating systems, security, and Internet. He is a member of ACM, IEEE CS, and IPSJ.