Journal of Computer Security 32 (2024) 291–317
DOI 10.3233/JCS-220044
IOS Press



# SIAT: A systematic inter-component communication real-time analysis technique for detecting data leak threats on Android

Yupeng Hu [a], Wenxin Kuang [a], Jin Zhe [d], Wenjia Li [b], Keqin Li [c], Jiliang Zhang [a] and Qiao Hu [a,*]

[a] *The Department of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan, China*
*E-mails: yphu@hnu.edu.cn, wenxinkuang@hnu.edu.cn, zhangjiliang@hnu.edu.cn, huqiao@hnu.edu.cn*
[b] *The Department of Computer Science, New York Institute of Technology, New York, USA*
*E-mail: wli20@nyit.edu*
[c] *The Department of computer science, State University of New York, New York, USA*
*E-mail: lik@newpaltz.edu*
[d] *China Tobacco Hunan Industrial Co., Ltd., Changsha, Hunan, China*
*E-mail: jinz1203@hngytobacco.com*



**Abstract.** This paper presents the design and implementation of a systematic Inter-Component Communications (ICCs) dynamic Analysis Technique (SIAT) for detecting privacy-sensitive data leak threats. SIAT's specific approach involves the identification of malicious ICC patterns by actively tracing both data flows and implicit control flows within ICC processes during runtime. This is achieved by utilizing the taint tagging methodology, a technique utilized by TaintDroid. As a result, it can discover the malicious intent usage pattern and further resolve the coincidental malicious ICCs and bypass cases without incurring performance degradation. SIAT comprises two key modules: *Monitor* and *Analyzer*. The *Monitor* makes the first attempt to revise the taint tag approach named TaintDroid by developing the built-in intent service primitives to help Android capture the intent-related taint propagation at multi-level for malicious ICC detection. Specifically, we enable the *Monitor* to perform systemwide tracking of intent with five abstraction functionalities embedded in the interactive workflow of components. By analyzing the taint logs offered by the *Monitor*, the *Analyzer* can build the accurate and integrated ICC patterns adopted to identify the specific leak threat patterns with the identification algorithms and predefined rules. Meanwhile, we employ the patterns' deflation technique to improve the efficiency of the *Analyzer*. We implement the SIAT with Android Open Source Project and evaluate its performance through extensive experiments on a particular dataset consisting of well-known datasets and real-world apps. The experimental results show that, compared to state-of-the-art approaches, the SIAT can achieve about 25% ∼200% accuracy improvements with 1.0 precision and 0.98 recall at negligible runtime overhead. Apart from that, the SIAT can identify two undisclosed cases of bypassing that prior technologies cannot detect and quite a few malicious ICC threats in real-world apps with lots of downloads on the Google Play market.

Keywords: Android malware, dynamic threats detection, inter-component communication, taint tags, threat patterns

*Corresponding author. E-mail: huqiao@hnu.edu.cn.

## 1. Introduction

In recent years, we have witnessed explosive growth in the number of mobile devices, and the large quantity of diversified mobile applications (apps) on those mobile devices have made our daily lives much more convenient and enjoyable. However, with the rapid growth of mobile apps, they have increasingly become the target of mobile malware authors, who generally develop and distribute mobile malware apps that aim at stealing and disclosing various types of sensitive and valuable information that is associated with either mobile user or device. As a result, malware has become one of the most significant security threats to mobile operating systems, especially Android.

In the Android system, the widely used Inter-Component Communication (ICC) [15] plays an essential role between the components of apps that are isolated in different sandboxes. Apps pass messages between each other by passing the *intents*, which are passive data structures holding the abstract descriptions of operations to be performed between components. Such a flexible method contributes a lot to functionality reuse and data sharing; however, it also exposes a vulnerable surface to several security threats [26]. In the context of ICC mechanism scenarios, apps whose developers overlooked security issues often suffer from risky vulnerabilities such as intent hijacking and spoofing [7], resulting in sensitive user data leak or privilege misuse by other apps, particularly mobile malware. Besides, two or more malicious apps with ICC paths could even collude on stealthy behaviors that neither of them could accomplish alone [5,12]. In these cases, malicious apps send and receive intents in a way that looks as if those are ordinary message exchanges. By this means, they can often easily bypass those classical malware detection approaches, which regularly inspect apps individually.

It is challenging to distinguish a normal or malicious ICC in a given security context. Many existing ICC-relative research works [27,31] focus on detecting vulnerabilities in benign apps. Benign apps do not have the malicious intent of ICC but may have some inherent design flaws. Recently, most of the research works that aim at identifying ICC paths with malicious purposes are in two categories: static analysis and running protection. A static analysis approach often extracts sensitive ICC paths by matching attributes and tracking data flow (e.g., IC3 [30], AmanDroid [39], DIALDroid [6]). However, even the state-of-the-art static analysis-based approaches suffer from many false positives because they cannot validate the specific data content through static analysis when facing the reflection and unreachable code. As a result, ignoring the validation of the data content in the static analysis will lead to an ICC path that does not occur in reality. Alternatively, runtime protection-based approaches (e.g., XManDroid [7] and SCLib [40]) either enforce mandatory access control according to the predefined policy set or ask about the End-user's decision for access permission to protect them from threats when apps communicate with each other using the ICC mechanism. However, those runtime protection-based approaches only pay attention to the information acquired before receiving intent, ignoring the actual behaviors of the receiver. Furthermore, they only determine whether to prohibit the mobile app from receiving intent according to various information, which makes these runtime protection techniques unable to identify malicious ICC paths for data leaks accurately.

Hereby, along with the explicit data flow, we pay more attention to identifying runtime ***intent-related implicit control flow*** that denotes a transfer of intent control across multiple components via ***complex ICC patterns*** other than an explicit component call. In this paper, we propose a systematic ICC real-time Analysis Technique (SIAT), which identifies malicious ICC patterns by tracking the data flows and the implicit control flow in the ICC processes at runtime by leveraging the taint tag approach TaintDroid [14]. *Unlike the traditional control flow tracking in static code analysis, SIAT tracks the implicit control*

*flow through the ICC process at runtime to identify the thorough malicious intent usage pattern.* Our key contributions are summarized as follows:

- We have concluded three kinds of typical data leak-related attacks and divided them into five malicious ICC patterns according to their behaviors.
- We have proposed SIAT,[1] the first systematic approach to accurately identify the malicious ICC patterns across apps with the help of the intent-related implicit control flows. This approach extracts the implicit control flows by examining both the sender and receiver sides for each intent communication at runtime.
- We have evaluated the performance of the SIAT through extensive experiments on both malware apps and benign apps composed of several well-known datasets and thousands of real-world Android apps.

The remainder of this paper is organized as follows. Related works are discussed in Section 2. Section 3.2 discusses the threat patterns SIAT focuses on. Section 4 provides our motivations and systematic methodology. Section 5 describes the overall architecture of SIAT. Section 6 presents the comprehensive performance evaluation that we have conducted for SIAT. Before drawing conclusion in Section 8, we introduce the in-depth discussion in Section 7.



## 2. Related work

App data leak threat issues have attracted a wealth of taint-based research efforts. [5] provides the first survey on inter-app communication threats, app collusion, and state-of-the-art detection tools in Android, providing a comprehensive assessment of the strengths and shortcomings of state-of-the-art approaches. The state-of-the-art approaches could broadly be divided into two categories: single-app analysis and app-pair analysis. And the analysis approaches of each category also are in two types: static analysis [42] and dynamic (a.k.a., runtime) analysis.

*Single-app analysis.* There are a few static single-app analysis approaches. CHEX [27] can identify the component hijacking vulnerabilities through static data flow analysis. Amandroid [39] focuses on analyzing inter-component data flows and tracking the interaction of the components. IccTA [25] addresses the major challenge of performing data flow analysis across multiple Android components for privacy leakage detection based on static taint analysis. The subsequent RAICC [35] reveals atypical ICC links in applications. It reflects the fact that their role is not primarily to start a component (as most ICC methods typically do in this paper) but rather to perform some action (e.g., set the alarm or send an SMS by starting a component with objects of type `PendingIntent` and/or `IntentSender`). DroidSafe [17] is a typical static information flow analysis tool that reports potential leaks of sensitive information in Android applications with accurate analysis stubs.

The dynamic single-app analysis approaches [9,14,20,22,41] monitor the app at runtime. As a data flow tracing method, TaintDroid [14] monitors the system at runtime and tracks the taint transmission to detect privacy leakage. IntentFuzzer [41] identifies the vulnerable interfaces by dynamically sending test intents to the components. IntentDroid [20] tests eight different vulnerabilities caused by unsafe handling of coming ICC intent data. DazeDroid [22] fully-automated extracts the components and fuzz all interfaces in apps.

[1] https://github.com/JinxKing/SIAT.

*App-pair analysis.* In the static app-pair analysis technologies, a precise and scalable end-to-end flow static analysis approach is introduced in [13] to identify the malware collusion risk in Android via fine-grained security risk classification policies. ApkCombiner [24] directly combines two apps into a single app and uses the single static data flow analysis method to identify sensitive ICC methods. COVERT [3] employs a compositional analysis method for finding inter-app vulnerabilities. JITANA [38] can analyze multiple android apps simultaneously. DIALDroid [6] analyzes each app and adopts the database to calculate the sensitive ICC path. PIAnalyzer [19] is a static approach for modeling specific vulnerabilities where other apps can intercept broadcasted PendingIntents. Although PRIMO [29] predicts the likelihoods of inter-app ICC occurrences via a formalism for ICC links based on set constraints, it cannot tackle links created by native code or Java reflection, and it is not designed for collusion detection.

Most dynamic app-pair analysis technologies enforce security policies only at the sender to protect users from inter-app threats. XManDroid [7] is the first approach proposed to prevent application-level privilege escalation through enforcing permission policies. FlaskDroid [8] provides a mandatory access control strategy simultaneously for both Android's middleware and kernel layers to prevent privilege escalation and collusive data leaks. SCLib [40] proposes an approach that performs inter-app mandatory access control for defending against component hijacking without modifying the Android system. SEALANT [23] combines static analysis and enforcing security policy to provide end-user protection.

Moreover, various works have explored detecting privacy leaks at the network level. For instance, ReCon [33] uses a machine learning classifier to identify leaks and can deal with simple obfuscation. AGRIGENTO [10] is resilient to obfuscation techniques, such as encoding, formatting, and encryption, by performing differential black-box analysis on Android apps. Existing technologies still suffer from some covert channels. Regarding ICCs, SIAT can handle the data obfuscation, encryption, or transmission via Secure Sockets Layer (SSL) based on our implicit control flow analysis.

In particular, there are some relevant works to SIAT. SpanDex [11] is integrated with TaintDroid and Android's Dalvik VM. Instead of ICCs, it is focused on securely tracking the password data flows within an app by monitoring explicit and implicit flows differently. Staicu [37] provides an empirical study of dynamic information flows for JavaScript at the language level and concludes that implicit flow tracking is needed for some privacy scenarios observable. In contrast, SIAT focuses on the control flow at the component level. *Unlike existing detection technologies, SIAT not only inspects the sender's but the receiver's intent-related behaviors by migrating the runtime approach TaintDroid to the systemwide tracing of intent data across multiple apps/components at runtime to figure out the real intent usage pattern of the related components. In this way, SIAT can significantly improve threat detection accuracy at the cost of negligible runtime overhead.*



## 3. Background and Data Leak Threats

### *3.1. Intent background*

Communications between components of mobile applications (a.k.a., apps) are achieved via sending and receiving intents, which are data structures holding an abstract description of operations to be performed and are generally used with methods to invoke activity, service, and broadcast receivers. Intentions can be divided into two types. One is the explicit intent, which specifies the target component by name, and the other is the implicit intent, which does not name the target (the component name field is blank) and is usually used to activate components in other applications. An intent filter is a key to

defining the behavior of intents, which works as an expression in an app's manifest file that specifies the type of intents the component would like to receive. Components that wish to receive implicit intents have to declare intent filters. *Although an intent filter offers a useful level of flexibility in the run-time binding of components, it is frequently overused or used inappropriately, with negative consequences for security.*

### *3.2. Data Leak Threat Patterns*

**Definition 1** (Malicious ICC for Data Leak)**.** Given a security context, a malicious ICC for data leak refers to a real ICC path among multiple components that leaks out sensitive data to an unauthorized party via transferring explicit/implicit intents.

As Section 6 details, we pay more attention to the malicious ICC paths incurred by malware through implicit control flows. These malicious ICCs mainly appear in three typical threat patterns, i.e., intent hijacking, intent spoofing, and intent collusion. As shown in Fig. 1, the following threat patterns that SIAT intends to identify in Section 6 lead to different malicious ICC behaviors that can steal or leak sensitive data.

*Intent hijacking.* Intent hijacking involves a malicious app receiving an intent not intended for it. As depicted in Fig. 1, in intent hijacking, the implicit intent may never reach the expected component, but an unauthorized app intercepts it. In *Victim* app, when the *Component A* sends intent to *Component B*, the *Malware*1 app can obtain the intent just by setting the attributes matched with the intent in the intent filter of the *Component C*. As a result, it is easy to cause data leakage during the intent hijacking. If the data (e.g., location, contacts) requires permission, and the intent does not restrict the receiver, *Malware*1 obtains the sensitive data without the necessary permission. In this case, the *Malware*1 escalates the privilege by hijacking the intent besides stealing the sensitive data.

*Intent spoofing.* Intent spoofing is an attack where a malicious app induces undesired behavior by forging an intent. Figure 1 illustrates how intent spoofing works. If the textitVictim app discloses that the *Component B* expects to receive intent from the *Component A* or some other components. Once textitComponent B does not have appropriate restrictions on the attributes of the intent filter, then textit-Malware1 may pretend to be textitComponent A and send intent to textitComponent B. In this case, it could trigger the corresponding action of textitComponent B to leak data.

*Intent collusion.* Intent collusion generally refers to the situation in which two apps cooperatively accomplish implicit malicious behaviors that a single app cannot achieve solely. As Fig. 1 shows, *Component D* in *Malware1* sends an intent with location data to *Malware2* via implicit intent (e.g.,`sendBroadcast(intent)`). Afterward, *Component E* receives the intent and sends out the location data through SMS messages. During the process of intent collusion, the malicious apps pretend to communicate normally with each other, which brings great challenges to identify those apps successfully as malware apps.

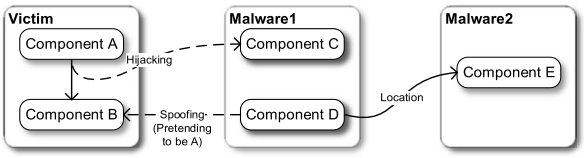



Fig. 1. The intent hijacking, spoofing and collusion patterns that bring forth ICC paths for data leaks will be identified by SIAT.

## 4. Motivations and methodology

*Motivations.* The actual risk of an ICC path intrinsically depends on the specific security context/semantics. It is challenging to tell the normal or malicious app by merely inspecting its ICC-related behaviors. For instance, if the *Component C* happens to have the attributes matching with the intent from the *Component B*, there might be an unintentional false positive hijacking case regarding conventional static analysis technologies. Ignoring that might result in high false positive rates. In contrast, a malicious ICC might seem normal. For example, in Fig. 1, if the receiver *Component E* cannot send out the location data via SMS message without required privileges, even though the *Component D* intends to, the ICC between them seems normal. Concerning the probabilistic matching between the implicit intent and intent filters (e.g., due to the mismatch of multiple intent filters and data types), existing methods are prone to false positives or false negatives in identifying the malicious ICCs in the threat patterns above.

*Methodology.* We need to differentiate the defined malicious ICC from normal ICC to solve the above problems. Therefore, we are dedicated to discovering the inherent logic behind implicit control flows via a systematic methodology. Through the ICC process, we perform a comprehensive taint analysis at the multi-level(i.e., app message-level, variable-level, method-level, and file-level) of both the sender and receiver sides. Our methodology almost has no false positives in runtime analysis, achieving significant recall owing to its systematic perspective. We showcase the solution of the *'coincidental malicious ICC'* in Section 6.1.2.

Let $M(I, F)$ be that the intent $I$ matches the intent filter $F$, i.e., a real satisfying match, and $E(I)$ be an explicit intent. As shown in Fig. 2, except for the explicit data flows (e.g., straightforward explicit intent), along with the $M(I, F)$ or $E(I)$ from source to sink methods [32], the proposed SIAT tries to track the thorough intent-related implicit control flows, i.e., $source \wedge (M(I, F) \vee E(I)) \wedge sink$.

We can discover the malicious ICC systematically based on these rules: (1) the threat patterns identified by Algorithm 1; (2) the sink leaking the sensitive data; (3) implicit/complex ICCs usage pattern which tending to circumvent our detection, consistent with the insight found in our manual verification in Section 6.2.



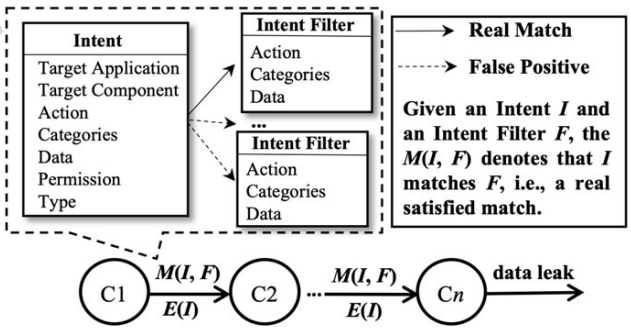



Fig. 2. The systematic methodology for implicit control flows identification. All possible methods the data could leave the device is a sink.

## 5. The SIAT

### 5.1. Technical challenges

*Technical challenges for architecture design.* A key challenge to overcome in architecture design is to design a sound architecture that can identify intent-related data and control flows without degrading detection accuracy and runtime performance.

*Technical challenges for the Monitor.* The critical challenge for the Monitor is finding a sound way to migrate TaintDroid to cooperate with Android for dynamic ICC path identification at the explicit data flows and the implicit control flows in a systematic perspective. The fundamental limitations of TaintDroid lie in two aspects, i.e., (1) it is a single-app analysis approach; (2) it can be circumvented through data leaks via implicit control flows. For example, in Fig. 7, the intent transferred via `SharedPreferences` can bypass the TaintDroid *due to the disposal of the tainted data in the put operation*. In addition, to monitor the implicit control flows in the ICC process, the two key challenges the *Monitor* have to deal with are: where the data in the intent by the sender initially comes from; and where the data in intent finally goes to in the receiver.

*Technical challenges for the Analyzer.* The critical challenge for *Analyzer* is building a complete and accurate ICC pattern with the taint logs.

### 5.2. The architecture of SIAT

SIAT works as a runtime safety guard to identify the malicious ICCs leaking data by analyzing the real-time data and control flows. The use scenes for real-world app guard are illustrated in Section 6.2. The primary objective of SIAT is performing systematic taint tracking by properly revising the runtime approach TaintDroid.

As shown in Fig. 3, to be practical, the primary design strategy of SIAT is to spread the complex detection workload to two different modules. Monitor and Analyzer are responsible for the runtime data collection and taint logs analysis in the background, respectively. Furthermore, combining the improved TaintDroid with Android via the well-defined intent service primitives, the SIAT provides a real-time

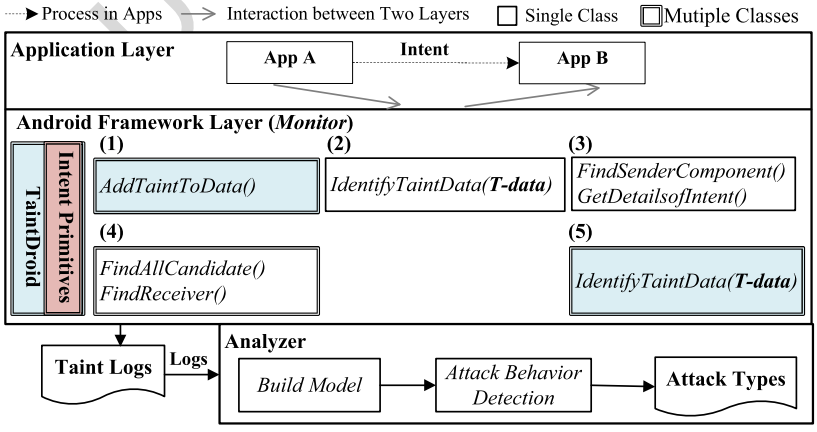



Fig. 3. The architecture of SIAT. Each logical function tracing the intent via the intent primitives denotes a set of taint or intent handling methods.

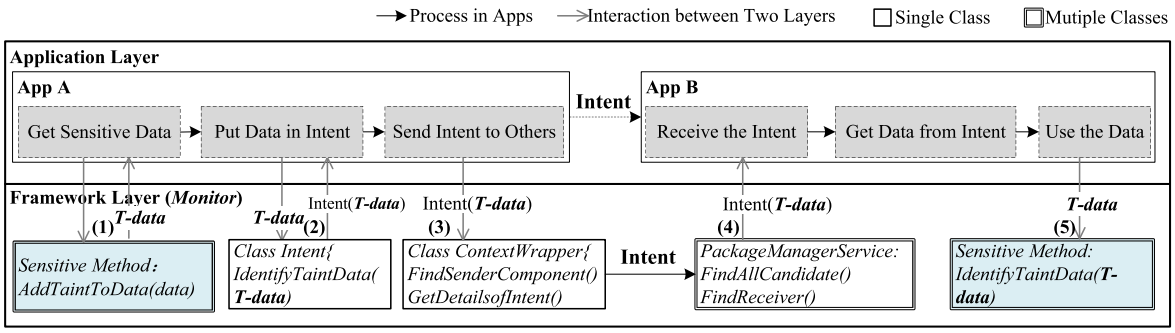



Fig. 4. The overall workflow (data flow and control flow) of *Monitor* in inter-component communication.

systematic tracing of privacy-sensitive data and visibility into how collaborative malicious behaviors occur via intent for data leaks. Meanwhile, we avoid revising their core logical structures when building explicit data flows and implicit control flows tracking on top of Android with TaintDroid to prevent functionality and performance degradation, as the evaluation results show in Section 6.

### 5.3. *Monitor*

*Monitor* is responsible for tracing and analyzing the flow of privacy-sensitive data at runtime by inspecting both the sender's and the receiver's intent. Figure 4 demonstrates its implementation of the five functionality steps in ICC workflow. The single and multiple classes denote the number of classes involved in implementing functions.

#### 5.3.1. *Key technique: TaintDroid migration*

Firstly, we need to develop the built-in intent service primitives in the main files of TaintDroid, enabling it to interact with Android at multi-level taint propagation (i.e., app message-level, variable-level, method-level, and file-level) without taint detection precision loss.

Secondly, we would like to leverage architectural features of components based inter-app communications, to enable the systematic tracking of implicit control flow with TaintDroid. We have deliberated on the extension of Android framework layer via code instrumentation engineering in the interactive workflows between TaintDroid and Android, covering the lifetime of four main Android components (i.e., Activity, Service, Content Provider, and Broadcast Receiver). Specifically, to obtain the optimal cost-benefits tradeoff, we have to perform an *in-depth* study on the runtime data collection workflow and thus abstracted five key functionality steps by extending the Android framework layer as shown in Fig. 3, which can interact with the intent service primitives to accurately carry out data and control flows collection.

The five logical functionality steps of the *Monitor* for overcoming the migration challenge are listed in Section 5.3.1. It is worth noting that each abstraction function in the five logical functionality steps above is not a concrete java function name. Instead, they denote a set of taint or intent handling methods for each step in our extension of the framework layer of the current Android operating system based on the Android Open Source Project. In this way, by using the intent service primitives to interact with TaintDroid, the *Monitor* inspects the relevant components on an ICC path and the data and control flows associated with the intent at runtime and further identifies the intent's sender, the intent-matched component, and the receiver at several critical points of the ICC process, respectively.

*5.3.2. Monitor implementation*

*Monitor* aims to track the thorough intent data/control workflows to identify the real senders and the receivers and their subsequent behaviors. The implementation places emphasis on the tracking of implicit control flows. We need to migrate TaintDroid to trace sensitive intent data.

Firstly, as highlighted in Fig. 3, we have defined a set of service primitives for intent communications. They encapsulate the sensitive data operation functions and work as middleware between the core methods of TaintDroid and the Android intent mechanism at the method-level and file-level framework layer. The new intent primitives encapsulate the functions of returning the source of current taint, obtaining the next tag with the original taint, setting/getting the tags, and so on. In this way, the main functions at the framework layer shown in Fig. 3 can cooperate with TaintDroid efficiently. For instance, when apps call APIs to get those privacy-sensitive data, based on the function *AddTaintToData(data)* in Fig. 4, we taint the data as the *T-data* by which we can trace and distinguish the data from others. Also, we can extract the tag from *T-data* and identify the *T-data* by comparing the number with the function *IdentifyTaintData(T-data)*. The bit vector of the tag is null if the data is not tainted.

Meanwhile, to catch the intended sensitive data accurately, by revising the main files of TaintDroid, our *Monitor* defines a group of new sensitive data and eighty taint tags for identifying them in intent communications. For example, the sensitive location data, `TAINT_LOCATION_Latitude=0x00010004`, `TAINT_LOCATION_Longitude=0x00010008`. Not only do we consider the privacy-sensitive data (e.g., locations, contacts, phone state) as sensitive, but we also regard the information that the user inputs or acquires from other files and other content providers (e.g., `SharedPreferences`). The 8-digit hexadecimal taint number is big enough to cover all sensitive data types we have defined.

Then, we leverage components-based inter-app communications by abstracting five key functionality steps embedded in the Android framework layer to enable the systematic tracking of implicit control flow with TaintDroid. To extract the intent usage pattern, we cover all critical methods involved in the lifetime of intent, i.e., the methods to start, send, find, and receive intent involved in main Android components, such as `startActivity()`, `bindService()`, and `resolveIntent()`, etc. In addition, to obtain the optimal cost-benefits tradeoff, we exploit an optimized probe-based codes instrumentation strategy to inspect the critical ICC checkpoints and generate the minimum set of codes extension, which incurs less code redundancy and logical structural disturbance.

Specifically, we implement the five functionality steps to track the intent through its whole lifetime as follows:

(1) Setting Taint. When the sender gets sensitive data from sensitive sources, using the function *AddTaintToData*, the *Monitor* taints the sensitive data and adds a variable tag (an 8-digit hexadecimal taint number) to it, which clearly labels its source. We name the sensitive data tainted as *T-data*. By examining thousands of apps, we identified thirty-eight types of sensitive data, such as location, phone number, history, network, SMS message, accelerometer, data from `SharedPreferences`[2] and so on.

(2) Checking Intent. When the sender sets the intent attributes (e.g. extra, action), the *Monitor* checks the *T-data* to see whether or not it is tainted or retained through the function *IdentifyTaintData(T-data)*.

(3) Sending Intent. If there is a sender that calls the system API, e.g., `startActivity()`, `startService()`, to send an intent, the *Monitor* identifies the identity of the sender and the details of the intent using functions *FindSenderComponent* and *GetDetailsofIntent*.

[2]SharedPreferences is a persistent storage method provided by Android.

(4) Receiving Intent. Upon obtaining the best matched component, the *Monitor* can find out all candidates and the real receiver via *FindAllCandidate* and *FindReceiver()*.
(5) Checking Taint. When the receiver extracts the *T-data* from an intent, as long as the sensitive APIs are called, the *Monitor* will check if any parameter in the APIs is *T-data* to identify the source of the data with *IdentifyTaintData*. Note that we exploit the multiple classes icon in Fig. 3 to denote that this functionality needs to perform more complex inspection operations at multiple key points than step (2).

In addition, To inspect the sensitive data in interested APIs, we take advantage of a mature machine-learning technique named 'SuSi' in [32] for achieving the most likely source and sink methods. The data is tagged as tainted if it comes from a privacy-sensitive source. If the tainted data is found in the sink, the privacy-sensitive data may inevitably be leaked. We describe the implementation of data and control flows tracking in *Monitor* by answering the four questions below:

**Is there any sensitive data in the intent?** When apps provide data for an intent, based on the *T-data*, we inspect the parameters of the data to see if it has been tainted with the function *IdentifyTaintData(T-data)*. If so, the tainted data will be retained with a new variable tag and a source code to show the intent's source clearly. If not, the data will be tainted with this specific intent. We have defined more than eighty types of tags to identify sensitive data, e.g., `TAINT_sharepreference=0x00010018`, `TAINT_network_state=0x00010012`. In this way, the *Analyzer* can easily figure out where the sensitive data comes from in the receiver at runtime.

**Who is the sender of the intent?** When an app sends an intent, we need to capture the sending event and the information of the original source component. The operation of calling API (e.g., `startService(intent)`) for sending an intent is generally inherited from another class for `Activity` or `Service` component. The `BroadcastReceiver` will execute the calling operation by acquiring the `Context` object and using the API in `Context`. As Fig. 4 shows, the implementation of these APIs is in the `ContextWrapper` class. Therefore, to reach the cost-effective goal, by integrating the codes for the functions of *FindSenderComponent* and *GetDetailsofIntent* into the `ContextWrapper` class, we can notice whenever an intent is sent. In this way, we can utilize the Java reflection mechanism to figure out which component calls the API to send the intent and which package the component belongs to, even though there are multiple transfers through the implicit control flow.

**Who is the receiver of the intent?** After capturing the sending event, we want to know which component becomes the candidate as its attributes match those of the intent and which component receives the intent at the end to disclose complex implicit control flow as we find in the evaluation. In our design, we integrate two functions *FindAllCandidate* and *FindReceiver* into the `PackageManagerService` (PMS) for querying the components that match with the intent by traversing the components of all apps as candidates. There are three types of components involved in intent matching: first, for the receiving component of `Activity`, if there is more than one matched component, the PMS selects one component from the list of candidates by comparing their priorities, such as the preferred order and so on. Alternatively, the PMS can also ask the user to choose one component; secondly, for the receiving component of `Service`, the *Monitor* will choose the first candidate; thirdly, for the receiving component of `BroadcastReceiver`, the PMS sends intent to all candidates. Therefore we can monitor all candidates and the actual receiver of intent in PMS to inspect the implicit control flow leading to data leak threats through the unintended receivers.

**How does the receiver use the sensitive data extracted from intent?** The *Monitor* inspects the data outputted to a file or sent to another device to determine whether it is tainted. If it is, the *Monitor* identifies the source of the data through the tag in the *T-data* with function *IdentifyTaintData(T-data)*.

Therefore, it can indicate how the receiver uses the data extracted from intent. Also, we consider more sensitive methods employed to store, send, or get sensitive data with taint tag, in the *Monitor*, such as the methods to store data in `SharedPreferences` and database (e.g., `Editor.putString()`). Since these sensitive methods don't need to apply for permissions, they could easily be overlooked by state-of-the-art technologies, leading to data leak threats such as the bypassing introduced in Section 6.1.3. Therefore, we design track methods showcased in Fig. 8 in Section 6.1.3 to address this implicit control flow based on Algorithm 2.

### *5.4. Analyzer*

*Analyzer* exploits the taint logs outputted by the *Monitor* to build the specific threat patterns reports for the users.

#### *5.4.1. Key technique: Pattern building*

As Table 1 depicts, a threat patterns to be built by *Analyzer* is composed of three objects, including the *Sender*, the *Intent*, and the *Receiver*. To ensure efficiency, we only adopt the most useful attributes, e.g., the *taint data*, which denotes the new sensitive data for intent. Based on Table 1, there are two key technologies below for building patterns:

*Intent data extraction.* To build accurate threat patterns, the *Analyzer* needs to extract the intent-related information from *APK* package and logs. Firstly, the *Analyzer* needs to extract the related nodes, child nodes, and their attributes by iterative traversal of the DOM tree in the `AndroidManifest` file, such as package names, permissions, intent-filter, action, category, data, etc. The package names enable the *Analyzer* to obtain the *process ID* of the app, which is a unique identifier assigned to each app by the Android system. Each record contains a *process ID* associated with the application that generated the log. Second, based on this process ID, the parser filters all unrelated logs from Android and other applications, where irrelevant logs refer to logs that do not contain the same process ID. Thirdly, the *Analyzer* reads every filtered log and extracts the intent relevant information that is useful for building the ICC patterns. For instance, as we can see in Listing 1, which shows the parts of logs containing the intent data we need to extract according to the predefined rules. The '1812' denotes the *process ID*, the tainted sensitive data is the device ID, and the tag has been changed from '1812' to '1056768' during the transfer in decimal. The logs also record the candidates (with identifier `ifilterMatch-`) and the real receiver package/component of intent.



Table 1
The most useful attributes of application and intent adopted in our ICC patterns analysis

| Sender | Intent | Receiver |
|---|---|---|
| process ID | action | process ID |
| package | categories | package |
| components | type | components |
| permissions required | scheme | permissions required |
| permissions lacked | taint data1 | permissions lacked |
| source methods[1] | ... | sink methods |
| candidates | taint data*N* | startCompt |

[1]The source methods indicate where the sensitive data most likely comes from based on the 'SuSi' [32].

```
TaintLog(1812): intentTaint :DeviceId
TaintLog(1812): intentTaint −content:000000000000000
TaintLog(1812): receiverLeak : tag−8192−newTag−1056768
  ......
TaintLog(276): ifilterMatch −component:
1812−lu.uni. serval . iac_startactivity1_sink . InFlowActivity
  ......
TaintLog(276): receiver −pakageName:
1825−lu.uni. serval . iac_startactivity1_sink
TaintLog(276): receiver −componentName:
1825−lu.uni. serval . iac_startactivity1_sink . InFlowActivity
```

Listing 1. The parts of the logs

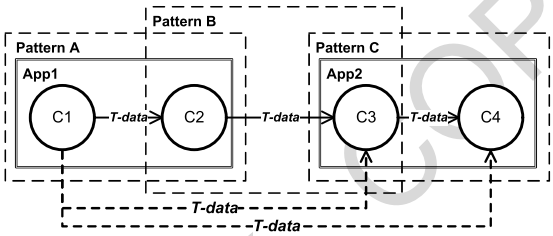



Fig. 5. The threat patterns of four components.

*Analysis of attributes related to permissions in patterns.* The *Analyzer* needs extra work to analyze the attributes related to the permissions in the patterns by identifying the attribute *permissions required* in the sender based on the permissions required to generate the tainted data in the intent. The attribute *permissions required* in the receiver is adopted to implement the *sink method*. Hence, for the sender, the attribute *permissions lacked* denotes the one that the sender does not have, but the receiver requires, and vice versa for the receiver. Firstly, there are redundant patterns generated by the multi-hop intent transfers between multiple components. Figure 5 depicts the generation of redundant patterns (*Pattern A, B and C*) built by the process mentioned above based on the four components in a streamlined way. It is incorrect that the source or destination of the sensitive data tainted as *T-data* is not the real component that sends or accepts the intent. In *Pattern B* and *Pattern C*, the *T-data*'s source is considered as *C*2 and *C*3 respectively, however, the real source and destination is *C*1 of *Pattern A* and *C*4 of *Pattern C* respectively. Secondly, there are extra patterns generated by the Android for launching the internal components that we are not concerned about. For instance, if the destination component is `activity` and there is more than one matching with the intent attributes, the Android will deliver the intent named *a* to the `ResolverActivity` firstly to let the user choose the desired component. After the user selects the destination component, instead of the sender, the `ResolverActivity` transfers a new intent named *b* to the destination.

To address the inflation issue, the *Analyzer* takes advantage of a deflation technique to eliminate the redundant patterns as follows. The deflation technique can build an ordered pattern list based on the components in the sender and the receiver of a single pattern. It then traverses each pattern to compare the

taint tag for identifying the real source in the sender and the destination/sink in the receiver, respectively. In this way, the three patterns in Fig. 5 will be condensed into one, and the *Analyzer* is able to figure out if the final receiver $C4$ starts a private component to leak out the sensitive data after receiving the intent. Likewise, the proposed deflation technique also can remove the unnecessary and interfering patterns that come from the Android system's internal components. For the example mentioned above of `ResolverActivity` in the second reason, using the deflation technique, the *Analyzer* is able to simplify the two patterns in the intent transfer process as one by executing the following steps: replacing the sender of intent $b$ with the sender of $a$ and then keeping the intent $b$ meanwhile discarding the $a$. Note that this deflation process does not negatively impact the detection owing to the systematic analysis of SIAT.

In addition, not only improving the pattern building efficiency, but the patterns deflation technique also helps to handle the multiple apps/components communications based on Algorithm 1 proposed below, e.g., detecting the intent collusion among three or more apps/components. Let the *deflation deep* be $n$, denoting the maximum number of components in an ICC path. The case in Fig. 5 can be extended to $n$ components, and the deflation ratio should be $\frac{1}{n-1}$.

*5.4.2. Analyzer implementation*

The *Analyzer* implements Algorithm 1 to identify the possible threat patterns in the ICC patterns. According to the attributes in Table 1, Algorithm 1 considers five different cases, which cover all data leak

**Algorithm 1** Threat patterns identification

**Input:** *patterns* ⇐ all patterns
**Output:** *PatternTypes* ⇐ a map of pattern and type

```
1: Let pattern be a pattern in patterns.
2: Let sender be the sender object in a pattern.
3: Let intent be the intent object in a pattern.
4: Let rcver be the receiver object in a pattern.
5: for each pattern ∈ patterns do
6:    for each component ∈ sender.components do
7:       if component ∈ Intent.candidates then
8:          add (pattern, "hijacking") to PatternTypes
9:          continue
10:      end if
11:   end for
12:   if (rcver.taintleak = true) ∧ (sender.lackpms = null) then
13:      add (pattern, "hijacking") to PatternTypes
14:   else if (sender.lackpms ≠ null) ∧ (rcver.lackpms = null) then
15:      add (pattern, "spoofing") to PatternTypes
16:   else if (rcver.startCompt = true) ∧ (rcver.lackpms = null) then
17:      add (pattern, "spoofing") to PatternTypes
18:   else if (sender.lackpms ≠ null) ∧ (rcver.lackpms ≠ null) then
19:      add (pattern, "collusion") to PatternTypes
20:   end if
21: end for
```

types we target in this paper. Different cases correspond to different identified rules. *Analyzer* iterates through each pattern to find the best matching case that has the same attributes.

**Case 1** (lines 6–11): Algorithm 1 traverses all components in the *sender* app to examine carefully if the list of candidates contains a component from the sender when the receiver component does not belong to the sender. If so, we add the pattern and the threat type "hijacking" to `PatternTypes`. In this case, we deem that the instance illustrated in Fig. 1 is happening and the candidate component from the sender should be the real destination instead of the receiver component. Therefore, the receiver component hijacks the intent, which is supposed to be sent to another component.

**Case 2** (lines 12–13): Suppose there is some data from the intent used by a specific sensitive method in the receiver. At the same time, the sender has the permissions related to the sensitive method in the receiver. In that case, we add the pattern and the threat type "*hijacking*" to `PatternTypes`. Afterward, the data extracted from the intent will be utilized by the sensitive method in the receiver, which means the receiver proactively acquires the private data from the sender through obtaining the intent. The situation in which the sender lacks permission related to a sensitive method in the receiver is considered illegal behavior and is classified as another type of threat.

**Case 3** (lines 14–15): When the sender lacks the permission that the receiver needs to call a sensitive method in the ICC process, but the receiver does have the permission for the data from the sender, we determine that the sender sends the intent to spoof the receiver and then assign the threat type in `PatternTypes` as intent "spoofing". In this case, the sender will let the receiver do something that it cannot do without the necessary permission for it. Therefore, the receiver's privileges will be misused unexpectedly.

**Case 4** (lines 16–17): When the receiver starts a private component with the permissions for the data from the sender, we think the receiver is spoofed and add the pattern with the threat type to `PatternTypes`. Hereby the sender calls a private component via the other exposed components and will not trigger any illegal behavior.

**Case 5** (lines 18–19): When the sender lacks the permissions that the receiver needs for calling a sensitive method, meanwhile the receiver also lacks permission to generate data from the sender, we consider that they are colluding. Thus we add the type intent "*collusion*" threat to `PatternTypes`, indicating that they are escalating the privilege from each other and complementing permissions for each other through the inter-component communication process, which is a clear case of intent collusion.

### *5.5. Data obfuscation resilience*

The data obfuscation resilience challenge for SIAT is to handle the malicious ICCs and try to obfuscate or encrypt sensitive data to circumvent the detection. Firstly, as AppFence [21] does, we extend TaintDroid to add tracking for all thirty-eight sensitive data types based on the interleaving taint tag allocation mechanism in the stack frame. Our interface library only provides the ability to add and not set or clear taint tags so that the untrusted functionality can not obfuscate or encrypt data to remove taint tags. Secondly, SIAT employs the retaint operation to track sensitive data, which might be obfuscated prior to being stored in an intent. Based on the implicit control flows analysis shown in Fig. 4 and Fig. 8 below, SIAT can capture a variety of SSL/encryption related operations (e.g., `SSLSocketFactory` and `SecretKeySpec`) and two bypassing operations in Section 6.1.3, and further retaints the data. Practically, the obfuscation techniques at the java codes level are exploited in the malware context to circumvent the detection of data leaks in the android system. To achieve this circumvention, the obfuscation methods transform the program codes (e.g., leaking sensitive data via throwing an exception) without

changing the behaviors. Nevertheless, SIAT can identify most cases of java code obfuscation given the intermediate methods/objects by the intent-related control flows analysis as depicted in Section 6.1.3. Also, SIAT can co-exist with conventional control dependencies, enabling formal protection techniques to discover more circumvention cases [18], which is beyond the scope of this paper.

### *5.6. The complexity*

The complexity of SIAT depends on the number of apps and components at runtime. Since the complexity of *Monitor* mainly relies on the actual lifetime of the app, here we focus on analyzing the complexity of *Analyzer*. Assume all feasible ICC patterns between $n$ apps per app contains $m$ components to be analyzed in the SIAT. There are $(nm - 1)$ components for each specific component that needs to be communicated. In practice, if only the vulnerable paths between multiple different apps could incur malicious behaviors, based on the Algorithm 1 and patterns deflation technique, there are $n(n-1)m^2$ patterns in this situation. Therefore, the computation complexity of building the ICC patterns and identifying the threat patterns is $O(n^2m^2)$ and $O(n^2m^3)$, respectively.

## 6. Evaluations

This section presents the experimental evaluation results of SIAT based on the four datasets below:

- DroidBench3.0 [2], which is an app collection for benchmarking ICC-based sensitive data leaks and consists of many types of ICC-related threats.
- Droidbench-iccta [1], which has three sets of apps for testing the inter-app collusion issues, and was released by EC SPRIDE Secure Software Engineering Group.
- Our Developments, similar to the DroidBench3.0, consist of more than forty self-developed apps that only have simple threat patterns and functions for comprehensive testing. Twenty-six ICC processes also cover at least three components with various sensitive APIs. Concerning efficiency and accuracy, the intent call entries are consistent with the app entries to simplify the call graph, and each app-pair ICC is independent of the other.
- Real-World, which contains about 2100 real-world apps[3] downloaded from the Google Play market [16]. We focus on collecting the most suspicious apps, e.g., cameras, notebooks, account books, social tools, malicious software (malware) injecting ads, and so on, according to the mobile malware revealed in the state-of-the-art approaches, Android security white paper, and data breach research reports in recent years.

Our evaluation addresses the following three questions:

- **RQ1:** What is the accuracy of SIAT compared to state-of-the-art approaches?
- **RQ2:** How well does SIAT perform in practice? What could SIAT find in real-world applications?
- **RQ3:** What about the individual performance of the *Monitor* and the *Analyzer*?

### *6.1. Results for RQ1 (accuracy comparisons)*

To evaluate the accuracy, we compare the SIAT with some state-of-the-art approaches achieving high accuracy in Section 2. They are the well-known runtime technique XManDroid, and two representative

[3]We have uploaded some typical apps in the link https://github.com/JinxKing/SIAT/tree/master/apk.

static approaches, DIALDroid and AmanDroid. These methods can easily be acquired and mainly focus on revealing data leak-related threats like SIAT.

*6.1.1. Accuracy comparisons overview*

As depicted in Fig. 6 and Table 2, we employ three performance indicators to evaluate the accuracy: **Precision**, denoted by $p$, which is the fraction of relevant instances among the retrieved instances; **Recall**, denoted by $r$, which is the fraction of the total amount of relevant instances that are actually retrieved; **F-measure**, which computes the comprehensive score $\frac{2\times p\times r}{p+r}$. Figure 6 depicts the value of three indicators in total. It is worth noting that the *ICC path* here denotes the malicious ICC path incurring data leaks.

Figure 6 provides an overview of the comparisons of accuracy. Tables 2 and 4 illustrate the details of the results. For simplicity, we merely present some typical detection results. Although we have identified more than twenty malicious ICC paths in our manual analysis, we chose seventy-five apps only covering eight ICC paths. Notably, we have tried to execute the IccTA's successor named RAICC [35], which boosts the detection by uncovering the atypical ICC methods within the app. Unfortunately, we could not execute RAICC + ApkCombiner [24] on most of the app pairs (over 70%). There are many crashed test cases, and RAICC + ApkCombiner can mainly detect the inter-app leaks in Droidbench-iccta (the same authors of RAICC), e.g., identifying a leak in the source app `SendSMS` with one extra false positive. The accuracy value might be misleading and does not reflect the actual performance. Thus we omit RAICC in comparisons here.

*Results of DIALDroid.* As Fig. 6 shows, the DIALDroid merely obtains 0.64 precision and 0.54 recall in total. The DIALDroid performs static taint analysis to identify attributes of the intent and the intent filter to trace the data flow associated with the intent. Then it uses SQL stored procedures and queries to calculate sensitive channels in the database according to the matching rules between the intent and the intent filter. However, the DIALDroid cannot accurately tell whether the data in the intent meets the requirements of the receiving component. When the data format doesn't meet the program's requirements,

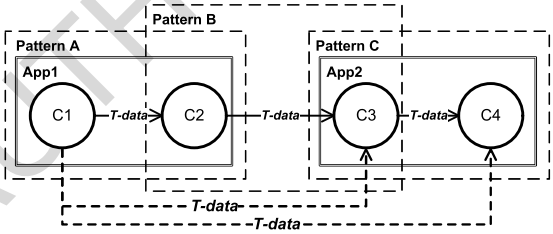



Fig. 6. Comparisons of three accuracy metrics.

Table 2

Overview of accuracy comparisons between DIALDroid, Amandroid, XManDroid, and SIAT

| DataSet | Number | | Malicious ICC Paths($\sum\checkmark/\sum\otimes$) | | | | Precision($p=\sum\checkmark/(\sum\checkmark+\sum\otimes)$)/Recall($r=\sum\checkmark$/number of ICC) | | | | F-Measure($2pr/(p+r)$) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Apps | ICC | DIALDroid | Amandroid | XManDroid | SIAT | DIALDroid | Amandroid | XManDroid | SIAT | DIALDroid | Amandroid | XManDroid | SIAT |
| **DroidBench3.0** | 11 | 9 | 7/2 | 0/0 | 9/0 | 9/0 | 0.78/0.78 | 0/0 | 1.00/1.00 | 1.00/1.00 | 0.75 | 0.00 | 0.94 | 1.00 |
| **Droidbench-iccta** | 6 | 3 | 3/0 | 3/0 | 3/0 | 3/0 | 1.00/1.00 | 1.00/1.00 | 1.00/1.00 | 1.00/1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| **Our Developments** | 40 | 26 | 10/2 | 5/0 | 19/2 | 26/0 | 0.83/0.38 | 1.00/0.19 | 0.90/0.73 | 1.00/1.00 | 0.52 | 0.32 | 0.81 | 1.00 |
| **Real-World** | 75 | 8 | 5/10 | 2/4 | 6/8 | 7/0 | 0.33/0.63 | 0.33/0.25 | 0.43/0.75 | 1.00/0.88 | 0.43 | 0.28 | 0.55 | 0.93 |
| **Total** | 132 | 46 | 25/14 | 10/4 | 37/10 | 45/0 | 0.64/0.54 | 0.71/0.21 | 0.78/0.80 | 1.00/0.98 | 0.59 | 0.32 | 0.79 | 0.99 |

The ICC path in Table 2 and 3 denotes the malicious ICC path incurring data leaks. $\checkmark$ = True Positive, $\otimes$ = False Positive, $\odot$ = False Negative.

Table 3

The partial results of ICC paths detection in DroidBench3.0, Droidbench-iccta, and our developments. The malicious behaviors of Real-World are listed in Table 4. The ICC paths here are true malicious ICCs, including the recognized ICCs in public datasets and self-developed ICCs

| Dataset | Source | Destination | Num. of ICC Paths | DIALDroid | Amandroid | XManDroid | SIAT(Ours) |
|---|---|---|---|---|---|---|---|
| DroidBench3.0 | SendSMS | Echoer | 1 | ✓ ⊗ | ⊙ | ✓ | ✓ |
| | StartActivityForResult1 | Echoer | 1 | ✓ ⊗ | ⊙ | ✓ | ✓ |
| | DeviceId_Broadcast1 | Collector | 1 | ✓ | ⊙ | ✓ | ✓ |
| | DeviceId_ContentProvider1 | Collector | 1 | ✓ | ⊙ | ✓ | ✓ |
| | DeviceId_OrderedIntent1 | Collector | 1 | ✓ | ⊙ | ✓ | ✓ |
| | DeviceId_Service1 | Collector | 1 | ⊙ | ⊙ | ✓ | ✓ |
| | Location1 | Collector | 1 | ✓ | ⊙ | ✓ | ✓ |
| | Location_Broadcast1 | Collector | 1 | ✓ | ⊙ | ✓ | ✓ |
| | Location_Service1 | Collector | 1 | ⊙ | ⊙ | ✓ | ✓ |
| Droidbench-iccta | startActivity1_source | startActivity1_sink | 1 | ✓ | ✓ | ✓ | ✓ |
| | startService1_source | startService1_sink | 1 | ✓ | ✓ | ✓ | ✓ |
| | startbroadcast1_source | startbroadcast1_sink | 1 | ✓ | ✓ | ✓ | ✓ |
| Our Developments | Sender0 | ReceiverTest0 | 1 | ✓ | ✓ | ✓ | ✓ |
| | Sender0 | ReceiverTest1 | 1 | ✓ | ✓ | ✓ | ✓ |
| | Sender0 | ReceiverTest2 | 1 | ✓ | ✓ | ✓ | ✓ |
| | Sender0 | Receiver-SharedPreferences | 1 | ⊙ | ⊙ | ⊙ | ✓ |
| | Sender0 | Receiver-application | 1 | ⊙ | ⊙ | ⊙ | ✓ |
| | Sender1 | ReceiverTest0 | 1 | ⊙ | ⊙ | ✓ | ✓ |
| | Sender1 | ReceiverTest1 | 1 | ⊙ | ⊙ | ✓ | ✓ |
| | Sender1 | ReceiverTest2 | 1 | ⊙ | ⊙ | ✓ | ✓ |
| | Sender1 | Receiver-SharedPreferences | 1 | ⊙ | ⊙ | ⊙ | ✓ |
| | Sender1 | Receiver-application | 1 | ⊙ | ⊙ | ⊙ | ✓ |

the sensitive method will not be executed. However, the DIALDroid does not consider it and assumes that the sensitive method must be executed. In addition, DIALDroid treats the case that sensitive data arrives in other applications via intent as a privacy breach, which improves the overall coverage while introducing false positives.

*Results of AmanDroid.* Similarly, the AmanDroid achieves 0.71 precision and 0.21 recall in that it cannot analyze the complex ICC-based data flow. The AmanDroid cannot analyze the data flows when facing the complex ICC paths, and thus it cannot detect any malicious ICC path in DroidBench 3.0, leading to a lower recall than others.

*Results of XManDroid.* As shown in Fig. 6 and Table 2, the XManDroid obtains 0.73 recall on Our Developments due to the seven ICC paths suffering from intent spoofing, which the XManDroid cannot identify. Consequently, the XManDroid only achieves a precision of 0.78 and a recall of 0.80 in total. The XManDroid enables users to predefine a list of ICC restriction policies and automatically block ICCs that match any policy. These policies are based on the permissions of the sender and the data in intent. Thanks to its permission identification mechanism, which will not intercept the delivery of intent only if the permissions in the receiver match the ones in the sender, the XManDroid performs well both on Droidbench-iccta and DroidBench3.0, as shown in Table 2. However, when the sender sends out the sensitive data with permission that the receiver doesn't have, the XManDroid prohibits this ICC directly without considering whether the receiver uses the data later. *This case is a common problem in many runtime protection approaches, which raises a high false alarm rate*. For example, the experimental results on Our Developments show that even if the receiver does not extract any sensitive data from intent, the XManDroid still thinks there is malicious behavior without identifying the receiver's behaviors. As a result, it makes the XManDroid detect two false positives ⊗. *In contrast, the SIAT traces both of the*

*data flows in the senders and receivers, then analyzes the whole transmission process that enables SIAT to generate fewer false negatives than the XManDroid.*

*Results of SIAT.* SIAT cannot track a few output methods due to the limitation of the built-in TaintDroid. However, compared to the existing approaches, as depicted in Fig. 6 and Table 2, the proposed SIAT can achieve an accuracy improvement of about 25%∼200% with a precision of 1.0 and a recall of 0.98. *There are two reasons why SIAT performs much better. Firstly, unlike the DIALDroid and the XManDroid, SIAT traces the data flow in the receiver at runtime by capturing and verifying the data in a sensitive method, which makes SIAT acquire more precise data flows. Secondly, DIALDroid, AmanDroid, and XManDroid do not detect intent spoofing, which is one of the major reasons their precision is lower than ours.*

### *6.1.2. Details of ICC path detection*

Table 3 shows details of malicious ICC paths of DroidBench 3.0 and Droidbench-iccta, and only ten malicious ICC paths of Our Developments due to the paper limits. The ICC paths in Real-World are given in Section 6.2. The original three ICC paths in Droidbench-iccta are innocent since the receivers can get the device ID from intent by themselves with the related permissions. To make the ICC paths illegal, we delete the permissions for device IDs in the three receivers.

The results in DroidBench 3.0 for DIALDroid are much better than AmanDroid; nevertheless, there still are two deficiencies: The first one is that the DIALDroid cannot identify the malicious ICC path when the type of the component is `Service`; obviously, there are two cases for the two source apps named `DeviceId_Service1` and `Location_Service1`; the second one is that the DIALDroid considers all possible branches to be executed when facing many branches in source codes, e.g., for destination app named `Echoer`, the two branches in codes make the DIALDroid detect two extra false malicious ICC paths denoted by ⊗.

Furthermore, as mentioned before, the DIALDroid cannot tell the receiver's real requirements of the intent data formats, leading to extra false positives (i.e., *aforementioned coincidental malicious ICC*). For instance, in a Real-World dataset, for app `vbox7handler`, it will exit immediately if the data in the intent does not have `vbox7 /.com /play`. However, DIALDroid still constructs the vulnerable ICC path between the sender and the `vbox7handler`. For another app named `UrlToPdf`, after receiving the intent, it will output the data to logs only if it identifies a key-value pair with the key `android.intent.extra.TEXT` in the vector `EXTRA` of the data in the intent. Nevertheless, the DIALDroid still considers that the log should be triggered since it doesn't care if the receiver validates the data, while SIAT can distinguish whether or not the log function should be triggered. On the other hand, all four approaches can achieve good detection results on Droidbench-iccta, regarding the simple ICC paths in 3-pair apps.

Indeed, from the software engineering perspective, the dynamic approaches only observe a limited part of the apps covered by the run-time inputs considered. However, a set of three attributes, i.e., action, data, and category, can readily cover most intended ICC execution paths in our evaluation. Therefore, the static techniques DIALDroid and Amandroid have lower accuracy than the other two dynamic approaches when facing complex ICC paths across multiple apps.

### *6.1.3. Two cases of bypassing*

The bypassing is similar to the malware collusion in a way, i.e., two components try to work cooperatively so that each component only performs part of the behavior to bypass the detection. Nevertheless, the main difference is that the two components come from the same application in the above cases of bypassing. Based on extensive experiments and in-depth analysis, as depicted in Fig. 7, in the receiver, we

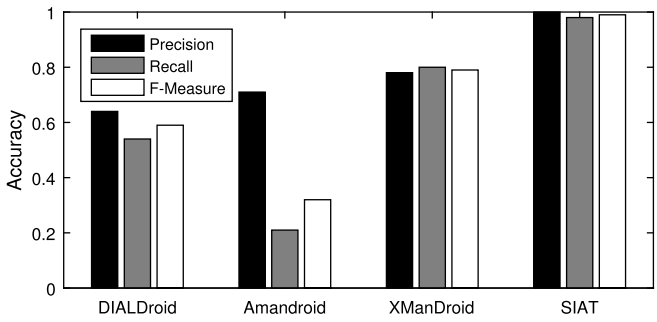



Fig. 7. Two cases of bypassing in the receiver.

```
public class Component_A extends AppCompatActivity { protected void onCreate(Bundle
    savedInstanceState ) { ...
    Intent receivedIntent = getIntent () ;
    receivedIntent . setClass ( this , MyService.class ) ;
    Editor editor = getSharedPreferences (" settings ", 0) . edit () ;
    String id= receivedIntent . getStringExtra (" android . intent . extra .TEXT");
    if (id != null ) {
      editor . putString (" deviceId ", id ) ;
      editor .commit();
    }
    startService ( receivedIntent ) ;
  }
}
```

Listing 2. The Component A puts the sensitive data into SharedPreferences

discover the following two undisclosed cases of malicious bypassing, which can invalidate the existing approaches by taking advantage of special intermediate methods/objects:

*SharedPreferences.* The first case is that, as shown in Fig. 7, in the receiver, if the *Component A* stores the sensitive data into the `SharedPreferences` in the form of a key-value pair after receiving the intent. Subsequently, the *Component B* can extract the data from the `SharedPreferences` object and output the data to the outside device. Listing 2 and 3 present example codes to showcase the bypassing in this case.

*Application.* Similarly, *component A* assigns the data extracted from the intent to the variable of the `Application` object after receiving the intent. There is a unique *Application* object per Android app at runtime, so each component can share the same one. Afterward, another *component B* can implicitly extract the data from the intent by searching the variable in the shared `Application` object and then calls the APIs for sending the SMS or writing in a file to output the data to the outside of the device as aforementioned.

*Our resolutions.* We have successfully realized the above two bypassing cases in Our Developments with destinations named `Receiver-SharedPreferences` and `Receiver-application`, as

```
public class Component_B extends Service {
  public int onStartCommand(Intent intent, int flags, int startId) {
    Log.v("leakData", getSharedPreferences("settings", 0).getString("deviceId", "default"))
        ;
    return super.onStartCommand(intent, flags, startId);
  }
}
```

Listing 3. The Component B gets the sensitive data from SharedPreferences and then outputs them

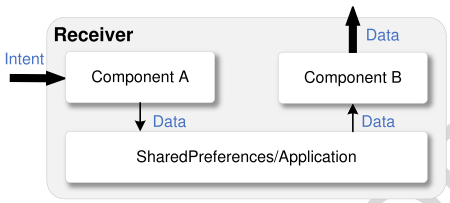



Fig. 8. The *Monitor*'s workflow on examples of Listings 2 and 3.

shown in Table 3. For the first case, both DIALDroid and AmanDroid only notice that the *Component A* has stored the sensitive data in the *SharedPreferences*. Still, they cannot detect that *Component B* has obtained the sensitive data from the `SharedPreferences`, and its following malicious behaviors. For the second case, we employ DIALDroid and AmanDroid to try to detect the app pair in several detection rounds. However, the results show no leak of sensitive data from receiver's intent. Consequently, malicious apps can easily bypass the detection of DIALDroid and AmanDroid in this case. Also, the XManDroid cannot detect the two cases due to its omitting of the actual behavior of the receiver, which incurs the false negatives $\odot$.

Based on the workflow for monitoring *SharedPreferences* in Fig. 8 and the identification algorithm of the original sources of tainted data for the bypassing in Algorithm 2, we showcase the solution of bypassing as follows:

Depending on a system-wide real-time tracing of the tainted data, as shown in Fig. 8, the *Monitor* firstly taints the original sensitive data with a tag. Then retains the data as *T-data*′ (*T-data*′ ← *T-data*) when storing it in an intent. Thanks to the TaintDroid, after being delivered to the receiver with intent, only if the *T-data*′ is utilized as the parameter of some sensitive function that is used to store data and then output them out of the device, our *Monitor* will identify the *T-data*′ and figure out its original source by comparing the taint tag and the predefined source code. Unfortunately, unlike the `Application` object, in `SharedPreferences`, the `putString` method will free the *T-data*′ in memory when writing it into the XML file as a key-value pair. That is, *the TaintDroid can be circumvented in this case, regarding the unknown original source.* In contrast, our systematic tracking method is able to deal with this case as follows: (1) logs the details of `putString` and `getString` actions which involve tainted sensitive data; (2) the *Analyzer* reads the logs to compare the tainted data related to `putString` and `getString` in sequence based on the unique key, to find out some equal data; (3) if there is a matching data, based on the improved patterns built by the Algorithm 2, the *Analyzer* can figure out the original source of the tainted data. In this way, when the corresponding component puts

**Algorithm 2** Identification of original source of tainted data for bypassing

**Input:** $m \Leftarrow$ current pattern needs to find original source of tainted data
**Output:** $m \Leftarrow$ improved patterns contains the original source of tainted data;

```
1: Let patterns be the patterns generated before
2: Let pattern be a pattern in patterns.
3: Let sender be the sender object in a pattern.
4: Let intent be the intent object in a pattern.
5: Let rcver be the receiver object in a pattern.
6: Let tdata be the tainted data in a pattern.
7: Let source[tdata] be the source method of tdata.
8: for each tdata ∈ m.rcver.tdata do
9:    if (tdata ∈ m.intent.tdata) then
10:      m.source[tdata] ← patterns.sender
11:   end if
12:   for each pattern ∈ patterns do
13:      if (pattern.rcver.component = m.sender.component) ∧ (tdata ∈ pattern.intent.tdata) then
14:         m.source[tdata] ← pattern.source[tdata]
15:      end if
16:   end for
17:   for each pattern ∈ patterns do
18:      if tdata ∈ pattern.intent.tdata then
19:         m.source[tdata] ← pattern.source[tdata]
20:      end if
21:   end for
22: end for
```

the *T-data*′ via the `putString`, and further sends the new tainted sensitive *T-data* out of the device after getting it out from the XML file, all functions that transfer the sensitive data will be accurately identified by SIAT. It is worth noting that, different from the *Application*, we implement Algorithm 2 for the *SharedPreferences* through a tainted value matching mechanism to trace the data flow based on the particular key-value pairs putting/getting entries in logs. To improve the original sources of the tainted data in the patterns, Algorithm 2 can match the tainted data in intent with the data in the receiver iteratively, even though the data are encrypted by an AES-based encryption tool (e.g., Secure-Preference [36] for *SharedPreferences*).

**RQ1 Answer.** SIAT can significantly increase the detection precision of ICC threats via systematic tracing, discovering the real intent usage pattern, and resolving the coincidental malicious ICCs and bypassing cases.

### *6.2. Results for RQ2 (Performance on Real-World Apps)*

SIAT can monitor and further record the apps' behaviors at runtime with taint logs. For evaluation, we have to run Real-World applications by adopting the automated testing script to trigger the applications' behaviors in the system. We need to design and input the corresponding scripts for each app to run the test with *monkeyrunner* [28], a popular Android tool for running test suites. However, it is time-consuming

Table 4
Analysis results on real-world apps

| Sender | | | | Receiver | | | | Type |
|---|---|---|---|---|---|---|---|---|
| Name | Version | Component | Sensitive Data | Name | Version | Component | Sensitive Method | |
| TopGoodNightImages | 10 | StartActivity | string-extra | TraductoresScout | 33 | MainActivity | Log | ***hijacking*** |
| PEC2012 | 24 | MoreTabActivity | string-extra | Prizmshare | 2 | MainActivity | write | ***spoofing*** |
| SimCardManager | 20400 | Main | deviceId | Notepad | 50 | NoteEditActivity | fileOutputStream | ***collusion*** |
| fotoalbumgpslite | 8 | ImageActivity | location | silentcamera | 13 | CameraActivity | — | — |
| lowlevel.sendapp | 11 | Application | — | urlripper | 72 | ProcessIntent | — | — |

The string-extra here denotes the string `extra` field in intent. '—' denotes the false positive case in existing approaches.

to handle many apps this way. To save the testing time and effort for the apps without malicious ICCs, we depend on the manual analysis engineering below to find the apps that hold the suspicious ICC paths and then write the corresponding scripts. Finally, we run the scripts and related apps simultaneously in our system to trigger the possible malicious behaviors for analysis.

*Manual analysis engineering.* We employ manual reverse engineering (dex2jar + jd-gui) to obtain the ICC paths-related source codes for each *APK* file. Firstly, we go through these codes by combining the sensitive methods defined in SIAT with the static code analysis tool DIALDroid, to investigate if each identified ICC path was indeed malicious. In this way, we have analyzed thousands of suspicious apps and quickly eliminated most of them (about 75%). Afterward, we run these script-driven suspicious apps with monkeyrunner under SIAT. Finally, we have verified that there are ICCs in about 163 application pairs without suspicious behaviors. For all pairs of applications that have ICCs, there is no sensitive data in the ICCs of the 121 applications. Meanwhile, there is intent hijacking in the ICCs of sixteen application pairs, intent spoofing in the ICCs of six application pairs, and malware collusions in the ICCs of four application pairs. Table 4 illustrates several typical unrevealed threats and false positives identified by SIAT.

Specifically, to ensure validity, we double-checked if sensitive methods related to malicious activities had been launched to exploit the vulnerable application successfully by carefully injecting some checkpoints into the Android system to identify the debug outputs. Statistically, we gain *an insight that most benign apps (about 84%) tend to use much more ICC paths than malicious apps, and the malicious apps intend to construct more complex ICC patterns to circumvent the detection.* It is worth noting that, to avoid subjectivity, two authors have carried out an inter-rate agreement protocol obtaining a Cohen's kappa of 0.9 or so, indicating an almost perfect agreement. Thus we focus on the ICCs that the two authors agreed on.



*Intent hijacking.* The `TopGoodNightImages` (10,000+ downloads) sends out an intent whose `extra` field stores the non-sensitive data. While the `TraductoresScout` (50,000+ downloads) receives the intent with required attributes and then lets the data leak out from intent into a log. Thus the data from the `TopGoodNightImages` can be sent out of the `TraductoresScout`, and other sensitive data will make it even worse.

*Intent spoofing.* After the `Prizmshare` receives the intent from the `PEC2012`, it writes the data extracted from the intent into a file. Even though the `PEC2012` can't write data to a file by itself due to the lack of the write permission, the receiver `Prizmshare` can escalate the privilege for it.

*Intent collusion.* The `SimCardManager` (10,000+ downloads) sends out an intent with device ID to the receiver `Notepad` (1,000,000+ downloads), then the `Notepad` receives the intent and stores

the extracted device ID into a file. Since the `SimCardManager` doesn't have permission to store a file and the `Notepad` neither has permission to get the device ID. It's utterly suspicious that they are complementing the permissions for each other.

*False positive cases.* Besides, there are two typical cases of false positives in existing approaches, which SIAT addresses in Table 4. The `Fotoalbumgpslite` sends out an intent whose data has the device's locations. While for the `silentcamera` (5,000,000+ downloads), the location information in intent is never traced in storage methods (e.g. `SharedPreferences.putString`) or other sensitive methods. Therefore, the sensitive data will not be leaked out to the receiver. The `Lowlevel` sends out a URL to the `urlripper`, and then the `urlripper` is able to access the Internet address. Hence, in SIAT, it is legal for the sender to utilize the function of other applications owing to its Internet permission.

**RQ2 Answer.** SIAT can uncover several unrevealed instances of data breach threats in real-world apps and identify several typical cases of false positives in existing approaches.

### 6.3. Results for RQ3 (run-time performance)

We now evaluate the runtime performance of SIAT with the above datasets. The *Monitor* and the *Analyzer* are two independent components of the workflow of SIAT. Notably, the actual runtime of *Monitor* is related to the lifetime of the app. Hence we evaluate their runtime overhead below separately before summing them.

#### 6.3.1. Monitor performance

Since our *Monitor* is a modified version of the Android system at the framework layer, we compute the app runtime cost on our *Monitor* and the native Android operating system, respectively. We randomly selected dozens of app pairs that can launch various malicious ICC paths from these datasets mentioned above. Also, to achieve the accurate runtime interval, we have inserted related time-stamped recording codes in a variety of crucial APIs in the *Monitor* and apps. It is worth noting that the runtime cost of each part on Android in Fig. 9 represents the recorded average app execution time at the same APIs after we run apps without the *Monitor*. We run apps under the *Monitor* and the Android native operating system,

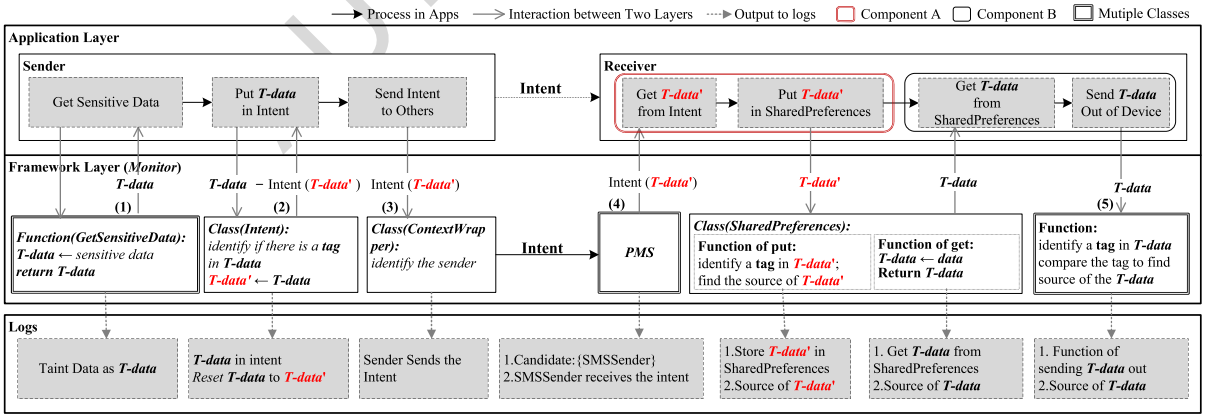
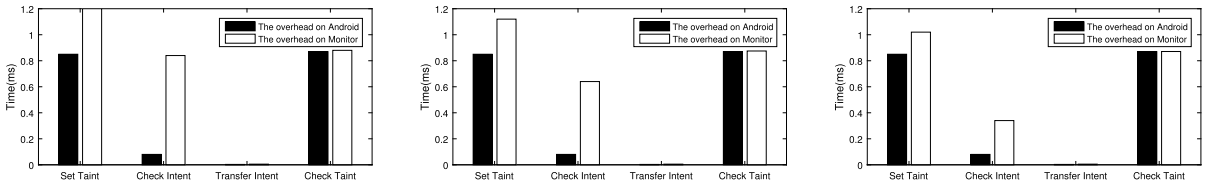



Fig. 9. The comparison of app runtime in various parts between *Monitor* and Android on different datasets. It is worth noting that we do not select the TaintDroid as the baseline in that the overhead of *Monitor* is almost the same as TaintDroid. DroidBench3.0 + IccTA denotes the short of DroidBench3.0 + Droidbench-iccta.




(a) Real-World. (b) Our Developments. (c) DroidBench3.0+IccTA.

Fig. 10. The time cost of *Analyzer* under various app-pairs.

respectively. We divide the *Monitor* module into four parts according to the five steps in Section 5.3 to calculate and compare the time cost separately.

Figure 9 shows the evaluation results for each part. In Fig. 9(a), the time overhead of Real-World is longer than the others in that the real-world apps maintain more complex and complete functionalities that lead to more ICC paths. Thanks to the lightweight extension scheme, which meets the requirements of SIAT architecture by inserting less than 800 lines of java codes in about forty essential files. Both in the transfer intent and check taint processes, the functions for *Monitor* almost do not incur any runtime overhead compared with the original Android. The set taint process leads to about 0.3 ms overhead due to the import of TaintDroid functions. The major runtime cost is incurred by the check intent process in *Monitor*, which exploits the reflective calls to figure out the component that sends the intent. Nevertheless, the overhead is less than 1 ms and is negligible for the end-users.

### 6.3.2. *Analyzer performance*

Figure 10(a) depicts the total time cost of analyzing multiple app-pairs with our *Analyzer* as the number of app-pairs increases. The entire time cost increases almost linearly, along with the number of app pairs. Thanks to the pattern deflation technique, the average time cost is less than 100 ms per app pair. The time cost of app-pairs in Real-World rises sharply when the number of app pairs reaches five, owing to some large-size apps that generate more complex patterns. Most of the app pairs from DroidBench3.0 + IccTA and Our Development only have simple structures and functions used for experimental purposes.

To investigate the influence of app size in-depth, we analyze a variety of app size-dependent factors that affect the time cost, such as the number of patterns, the size of log files, the number of codes, and so on. We find that the number of patterns is the most influential factor. In this regard, we carry out an experiment that takes about 13.7 s to analyze a log file containing about 200 patterns (including threat and normal patterns), indicating an average time cost of 68.5 ms per pattern.

Figure 10(b) further illustrates the average time cost of analyzing the logs of 140 different app pairs with the *Analyzer* in twenty runs. These 140 different app-pairs are randomly chosen from the datasets. The $x$ axis is the serial number identifying every app pair. Consistent with Fig. 10, the time cost of the DroidBench3.0 + IccTA and Our Developments concentrates on the lower time region of less than 60 ms. In contrast, the time cost of analyzing the logs of the app pairs from the Real-World dataset is much longer than others due to a large number of patterns incurred by their sophisticated functions.

**RQ3 Answer.** Thanks to the sound design strategy of SIAT, the main runtime cost of the Monitor is incurred by the check intent process ($\leqslant$1 ms), which exploits the reflective calls to figure out the component that sends the intent. The overall time overhead for processing a single app-pair is less than 200 ms and thus is negligible.

## 7. Limitations

*Usability limitations.* There are the same usability obstacles of SIAT as some existing approaches. From the software engineering point of view, the users may not accept installing a customized Android with built-in SIAT due to security and usability concerns. Therefore, we would employ the SIAT as a runtime safety guard background for ICCs generating ICC security reports.

*Data flow limitations.* Implicit control flow is a transfer of control between procedures using some mechanism, which adds flexibility to system design [4]. Unlike traditional tracing of implicit data flows [34], our approach does not deal with covert channels, or implicit data flows. Instead, we only deal with the explicit data flows and the intent-related implicit control flows of the sender and receiver sides in Inter-Component Communications scenarios where privacy-sensitive data leakage may exist.

*Subjectivity limitations.* The manual analysis for real-world apps is subject to human subjectivity, which has never been well-studied. In our evaluations, we pay more attention to the complex ICC path for disclosing sophisticated malicious purposes. Nevertheless, a Cohen's kappa of 0.9 indicates an almost perfect agreement between the two authors. Besides, the employed DIALDroid is subject to false positives or false negatives in the cases of Section 6.1.1 and 6.1.3, respectively, and our sensitive methods are identified based on 'SuSi' and taint tags. Thus we cannot cover all cases in our manual analysis.

## 8. Conclusion

In this paper, we present the design and implementation of the SIAT, which provides real-time systematic tracking of privacy-sensitive data and visibility into how the collaborative malicious behaviors take place via intent for data leaks, based on its two crucial modules: *Monitor* and *Analyzer*. With the well-defined built-in intent service primitives for the seminal TainDroid, enabling the cooperation of TainDroid and the extended framework layer of Android at multi-level of intent propagation, SIAT works as a runtime safety guard for ICCs by not only handling the explicit data flows but also the intent-related implicit control flows at both the sender and receiver sides in a systematic perspective, discovering the intent usage pattern and resolving the coincidental malicious ICCs and bypassing cases.



## Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61872130, 62122023, U20A20202, 62002167, and 61874042; the Science and Technology Project of the Department of Communications of Hunan Provincial under Grant No.201928; the Hunan Natural Science Foundation for Distinguished Young Scholars under Grant No. 2020JJ2010, the Hunan Science and Technology Innovation Leading Talents Project under Grant No. 2021RC4019, the Key R & D Projects of Changsha under Grant No.kq1907103, the Youth Program of National Natural Science Foundation of China under Grant No.61902121.

## References

[1] S. Arzt, Droidbench-iccta, https://github.com/secure-software-engineering/DroidBench/tree/iccta.

[2] S. Arzt, Droidbench3.0, https://github.com/secure-software-engineering/DroidBench.
[3] H. Bagheri, A. Sadeghi, J. Garcia and S.M. Covert, Compositional analysis of Android inter-app permission leakage, *IEEE Transactions on Software Engineering* **41**(9) (2015), 866–886. doi:10.1109/TSE.2015.2419611.
[4] 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: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 669–679. doi:10.1109/ASE.2015.69.
[5] S. Bhandari, W.B. Jaballah, V. Jain, V. Laxmi, A. Zemmari, M. Singh Gaur, M. Mosbah and M. Conti, Android inter-app communication threats and detection techniques, *Computers & Security* **70** (2017), 392–421. doi:10.1016/j.cose.2017.07.002.
[6] A. Bosu, F. Liu, D.D. Yao and G. Wang, Collusive data leak and more: Large-scale threat analysis of inter-app communications, in: *Proceedings of the 2017 ACM on AsiaCCS*, ACM, 2017, pp. 71–85.
[7] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer and A.-R. Sadeghi, Xmandroid: A new android evolution to mitigate privilege escalation attacks, Technische Universität Darmstadt, Technical Report TR-2011-04, 2011.
[8] S. Bugiel, S. Heuser and A.-R. Sadeghi, Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies, in: *Presented as Part of the 22nd USENIX Security Symposium*, 2013, pp. 131–146.
[9] H. Cai, N. Meng, B. Ryder and D.Y. Droidcat, Effective Android malware detection and categorization via app-level profiling, *IEEE Transactions on Information Forensics and Security* **14**(6) (2019), 1455–1470. doi:10.1109/TIFS.2018.2879302.
[10] 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: *NDSS*, 2017.
[11] L.P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu and S.C. Spandex, Secure password tracking for Android, in: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 481–494.
[12] A. Das, K. Das and M.S. Hossain, An integrated inspection and visualization tool for accurate Android collusive malware detection, in: *7th International Conference on Networking, Systems and Security*, 2020, pp. 107–114. doi:10.1145/3428363.3428376.
[13] K.O. Elish, H. Cai, D. Barton, D. Yao and B.G. Ryder, Identifying mobile inter-app communication risks, *IEEE Transactions on Mobile Computing* **19**(1) (2020), 90–102. doi:10.1109/TMC.2018.2889495.
[14] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel and A.N. Sheth, Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, *ACM Transactions on Computer Systems (TOCS)* **32**(2) (2014), 5. doi:10.1145/2619091.
[15] W. Enck, M. Ongtang and P. Mcdaniel, Understanding Android security, *IEEE Security & Privacy* **7**(1) (2009), 50–57. doi:10.1109/MSP.2009.26.
[16] Google play market, http://paly.google.com/store/apps/.
[17] M.I. Gordon, D. Kim, J.H. Perkins, L. Gilham, N. Nguyen and M.C. Rinard, *Information Flow Analysis of Android Applications in Droidsafe*, 2015.
[18] M. Graa, N. Cuppens Boulahia, F. Cuppens and A. Cavalliy, Protection against code obfuscation attacks based on control dependencies in Android systems, in: *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, IEEE, 2014, pp. 149–157. doi:10.1109/SERE-C.2014.33.
[19] S. Groß, A. Tiwari and C.H. Pianalyzer, A precise approach for pendingintent vulnerability analysis, in: *European Symposium on Research in Computer Security*, Springer, 2018, pp. 41–59.
[20] R. Hay, O. Tripp and M. Pistoia, Dynamic detection of inter-application communication vulnerabilities in Android, in: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ACM, 2015, pp. 118–128. doi:10.1145/2771783.2771800.
[21] P. Hornyack, S. Han, J. Jung, S. Schechter and D. Wetherall, These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications, in: *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 639–652. doi:10.1145/2046707.2046780.
[22] R. Johnson, M. Elsabagh, A. Stavrou and J. Offutt, Dazed droids: A longitudinal study of Android inter-app vulnerabilities, in: *Proceedings of the 2018 on AsiaCCS*, ACM, 2018, pp. 777–791.
[23] Y.K. Lee, J.Y. Bang, G. Safi, A. Shahbazian, Y. Zhao and N. Medvidovic, A sealant for inter-app security holes in Android, in: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 312–323.
[24] L. Li, A. Bartel, T.F. Bissyandé, J. Klein and Y. Le Traon, Apkcombiner: Combining multiple Android apps to support inter-app analysis, in: *IFIP International Information Security and Privacy Conference*, Springer, 2015, pp. 513–527.
[25] L. Li, A. Bartel, T.F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau and P.M. Iccta, Detecting inter-component privacy leaks in Android apps, in: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015, pp. 280–291.
[26] F. Liu, H. Cai, G. Wang, D. Yao, K.O. Elish and B.G. Ryder, Mr-droid: A scalable and prioritized analysis of inter-app communication risks, in: *2017 IEEE Security and Privacy Workshops (SPW)*, 2017, pp. 189–198. doi:10.1109/SPW.2017.12.

[27] L. Lu, Z. Li, Z. Wu, W. Lee and G. Jiang, Chex: Statically vetting Android apps for component hijacking vulnerabilities, in: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM, 2012, pp. 229–240.
[28] monkeyrunner, https://developer.android.com/studio/test/monkeyrunner.
[29] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein and Y. Le Traon, Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis, in: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 469–484.
[30] D. Octeau, D. Luchaup, M. Dering, S. Jha and P. McDaniel, Composite constant propagation: Application to Android inter-component communication analysis, in: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015, pp. 77–88.
[31] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein and Y. Le Traon, Effective inter-component communication mapping in Android: An essential step towards holistic security analysis, in: *Presented as Part of the 22nd USENIX Security Symposium*, 2013, pp. 543–558.
[32] S. Rasthofer, S. Arzt and E. Bodden, A machine-learning approach for classifying and categorizing Android sources and sinks, in: *NDSS*, Vol. 14, Citeseer, 2014, p. 1125.
[33] J. Ren, A. Rao, M. Lindorfer, A. Legout and D.C. Recon, Revealing and controlling pii leaks in mobile network traffic, in: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 361–374. doi:10.1145/2906388.2906392.
[34] A. Russo, A. Sabelfeld and K. Li, Implicit flows in malicious and nonmalicious code, in: *Logics and Languages for Reliability and Security*, IOS Press, 2010, pp. 301–322.
[35] J. Samhi, A. Bartel, T.F. Bissyandé and J. Klein, Raicc: Revealing atypical inter-component communication in Android apps, in: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1398–1409.
[36] Secure-preference, https://github.com/scottyab/secure-preferences.
[37] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel and A. Sabelfeld, An empirical study of information flows in real-world javascript, in: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2019, pp. 45–59. doi:10.1145/3338504.3357339.
[38] Y. Tsutano, S. Bachala, W. Srisa-An, G. Rothermel and J. Dinh, An efficient, robust, and scalable approach for analyzing interacting Android apps, in: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 324–334.
[39] F. Wei, S. Roy, X. Ou et al., Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps, in: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 1329–1341. doi:10.1145/2660267.2660357.
[40] D. Wu, Y. Cheng, D. Gao, Y. Li and R.H. Deng, Sclib: A practical and lightweight defense against component hijacking in Android applications, in: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ACM, 2018, pp. 299–306. doi:10.1145/3176258.3176336.
[41] K. Yang, J. Zhuge, Y. Wang, L. Zhou and H. Duan, Intentfuzzer: Detecting capability leaks of Android applications, in: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ACM, 2014, pp. 531–536.
[42] X. Yu, F. Wei, X. Ou, M. Becchi, T. Bicer and D. Yao, Gpu-based static data-flow analysis for fast and scalable Android app vetting, in: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 274–284. doi:10.1109/IPDPS47924.2020.00037.