AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection

Zhemin Yang
Fudan University
yangzhemin@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

Yuan Zhang
Fudan University
yuanxzhang@fudan.edu.cn

Guofei Gu
Texas A&M University
guofei@cse.tamu.edu

Peng Ning
NC State University
pning@ncsu.edu

X. Sean Wang
Fudan University
xywangcs@fudan.edu.cn

Abstract

Android phones often carry personal information, attracting malicious developers to embed code in Android applications to steal sensitive data. With known techniques in the literature, one may easily determine if sensitive data is being transmitted out of an Android phone. However, transmission of sensitive data in itself does not necessarily indicate privacy leakage; a better indicator may be whether the transmission is by user intention or not. When transmission is not intended by the user, it is more likely a privacy leakage. The problem is how to determine if transmission is *user intended.* As a first solution in this space, we present a new analysis framework called AppIntent. For each data transmission, AppIntent can efficiently provide a sequence of GUI manipulations corresponding to the sequence of events that lead to the data transmission, thus helping an analyst to determine if the data transmission is user intended or not. The basic idea is to use symbolic execution to generate the aforementioned event sequence, but straightforward symbolic execution proves to be too time-consuming to be practical. A major innovation in AppIntent is to leverage the unique Android execution model to reduce the search space without sacrificing code coverage. We also present an evaluation of AppIntent with a set of 750 malicious apps, as well as 1,000 top free apps from Google Play. The results show that AppIntent can effectively help separate the apps that truly leak user privacy from those that do not.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

Keywords

Android security; privacy leakage detection; symbolic execution

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516676 .

1. INTRODUCTION

With the growing popularity of Android, millions of applications (or *apps* for short) are available to users from a variety of Internet sites (called *app markets*). While users enjoy the rich features of the apps, their sensitive personal data, such as phone numbers, current locations, and contact information, may be stealthily collected and misused by the ill-intended developers of some apps. A recent study has showed that Android apps frequently transmit private data to unknown destinations without user consent [46]. To protect users, there is a great need for strong analysis tools that Android app markets can use to identify and remove malicious apps.

State-of-the-art approaches of privacy leakage detection on smartphones focus on detecting sensitive data transmission, i.e., whether personal data leaves the device [21, 22, 26, 30, 40, 29]. However, in this era of mobile apps with cloud computing, what constitutes a privacy leakage by mobile apps is a subject that needs reconsideration. Many benign apps provide services from the cloud to end users. These apps normally need to collect sensitive data such as location, contact, to send out to the cloud. Malicious apps that steal user data may also exhibit the same behavior, namely transmitting private information to the cloud (or via other means). Therefore, transmission of sensitive data by itself may not indicate true privacy leakage; a better indicator should be whether the transmission is *user intended or not.*

- *User-intended data transmission.* To use the function provided by an app, a user often tolerates his/her private data being sent out via some communication channels. For example, when using SMS management apps [3], a user can forward an SMS message to a third party, by several button clicking on the touchscreen. As another example, when using a location-based service [7], a user usually knows his/her location is sent out to get interesting contents tailored to the location. Since this kind of functional use of sensitive data is consistent with user intention, we should not treat this kind of transmission as a privacy leakage.
- *Unintended data transmission.* The irregular transmission of sensitive data performed by an app, which is unknown to users and irrelevant to the function user enjoys, is defined as unintended data transmission, or privacy leakage. In most cases, users are unaware of this kind of transmission because the malicious apps always do that in a stealthy manner.

The above shows that whether sensitive data transmission is a privacy leakage or not actually depends on whether the transmission is user intended or not. Unfortunately, due to the complex nature of user intention and different/unpredictable settings of different apps, it is almost impossible to have an automated method to determine user intentions. Alternatively, it is more practical to design an automated tool to provide a human analyst with the context information in which the data transmission occurs. Intuitively presented context information will make the task of the human analyst easier in determining if the transmission is user intended. This motivates our work on the *AppIntent* framework. Given sensitive data transmission, AppIntent derives the input data and user interaction inputs that lead to the transmission. The context information of the transmission shown to the analyst is in the form of a sequence of UI manipulations (i.e., GUI screens along with the highlighted GUI controls that indicate the supposed user operations) that is captured from a controlled execution of the app with the derived input data and user interaction. By looking at the displayed UI manipulations, a human analyst can then make a judgement.

Symbolic execution is an effective technique to extract feasible inputs that can trigger specific behaviors of a program such as particular transmission of sensitive data. The key idea of symbolic execution is to systematically explore feasible paths of the program under analysis by reducing the search space from an infinite number of possible data inputs to a finite number of data scopes (represented by symbolic inputs). However, existing symbolic execution techniques mainly focus on non-interactive programs [10, 16, 28, 39]. Dealing with events triggered by user actions in GUI apps is challenging because the possibly large number of combinations of input events can severely worsen the path explosion problem during symbolic execution. However, in AppIntent, user interactions cannot be abstracted away from apps for symbolic execution because user interaction is an essential part to judge whether the transmission is intended by the user or not.

To deal with the path explosion problem, we have developed a new symbolic execution technique called *event-space constraint guided symbolic execution* for Android apps. We first apply static analysis to the target app to identify the possible execution paths leading to the sensitive data transmission under analysis (such as sending SMS). We then use these paths as the basis to generate our event-space constraints, which represent all the possible event sequences for the given execution paths by considering the call graph and the Android execution model. Our guided symbolic execution then considers only the paths that satisfy the event-space constraints. Our experiments show that these constraints restrict the search space very effectively since the number of execution paths to be explored during the guided symbolic execution is usually small.

To evaluate the effectiveness of *AppIntent*, we perform an extensive experimental evaluation using real-world apps including 750 malicious apps reported in [46] and 1,000 top free apps from Google Play, to detect whether they transmit user's private data and to distinguish whether the transmission is user intended or not. In our experimental results, 252 apps have sensitive data transmission, among which 224 apps contain user *unintended* transmission while other 28 apps contain only *user-intended* data transmission.

The contribution of this paper is fourfold. First, we note that sensitive data transmission does not always indicate privacy leakage; rather, user-intended data transmission should be discriminated from user-unintended. Second, we develop an event-space constraint guided symbolic execution technique, which effectively reduces the event search space in symbolic execution for Android apps. As a result, event inputs as well as data inputs related to each propagation path of data transmission can be effectively extracted. Third, we develop a dynamic program analysis platform to execute the app driven by the discovered event and data inputs, so that we can display the sequence of UI manipulations, emulating the entire process leading to the data transmission. Finally, we evaluate our approach by using 750 reported malicious apps, as well as 1,000 top free apps from Google Play. Some interesting findings are also provided together with the evaluation results.

The rest of this paper is organized as follows. Section 2 introduces the challenge of symbolic execution for Android, and Section 3 gives an overview of the AppIntent framework. Section 4 presents the details of event-space constraint guided symbolic execution. The dynamic analysis platform of AppIntent is depicted in Section 5. Section 6 presents the evaluation of AppIntent using real-world Android apps. Section 7 discusses the related work, and Section 8 concludes this paper and points out some future research directions.

2. BACKGROUND: SYMBOLIC EXECUTION FOR ANDROID APPS

Symbolic execution is a program analysis technique that has been used in a wide range of applications such as test case generation [14, 17, 27, 28, 34, 39], fuzz testing [35], and security flaws detection [13, 15, 20, 26, 31, 42]. It is a traversal process, which explores a search space during the analysis process. The general idea of symbolic execution is to limit the search space because its execution time and practicability depend on this scope. For those non-interactive programs, symbolic execution can efficiently explore the search space of data inputs through a well-defined classification of these inputs. However, symbolic execution faces unresolved challenges when it is applied to GUI apps.

GUI apps, which are widely used in computers and handheld devices, are driven by not only data inputs, but also event inputs. Users can interact with apps by triggering runtime events such as clicking a certain button. Event inputs, which introduce highly variable program behaviors and hard to be classified into input scopes, greatly increase the search space of GUI apps. To the best of our knowledge, there are no efficient solutions to this problem, and most of the existing symbolic execution approaches for GUI apps sacrifice code coverage for performance by applying random scheduling strategy [38], exhaustively searching possibilities (to an upper bound of event sequences) [25], or assuming that event handlers will not cooperate with each other [24]. Recently, Contest [9] reduces the symbolic execution time of smartphone apps to 5%-36% of the original running time by utilizing profiling results, but the cost of this analysis is still too high.

When modeling the space of runtime event inputs, the most important characteristic of the space is the possible orders of events. In most cases, the behavior of a GUI app can be represented by the events triggered by the user along with the order of these events.

2.1 Android Basis

Similar to Java GUI apps, Android apps are usually driven by runtime events and callbacks. The non-determinism introduced by arbitrarily and distinctively triggered events increases the complexity when exploring the search space and severely challenges the symbolic execution of GUI apps. The search space of events is decided by Android programming and execution model, which needs a careful consideration in analysis.

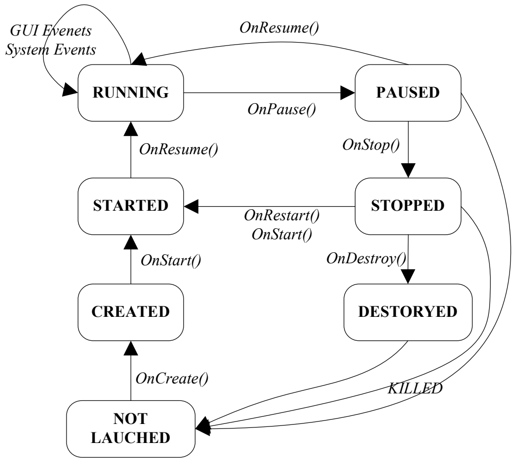


Figure 1: Android application model. This figure depicts the lifecycle of Android activities. The lifecycle of other components are similar.

There are two major kinds of events in Android: callbacks to manipulate the state transition of an app, and listeners to handle system events and user interactions with GUI components:

Android Events: Callbacks of Lifecycle States. Unlike in the common Java world, Android app does not have a unique program entry such as main(). Instead, it is composed of one or more components which work together to fulfill the functionality. The major type of components in Android is activity. An activity represents a single screen with a user interface. The other components, e.g., services, content providers, and BroadcastReceivers, are background tasks that perform long-running operations or respond to other threads. For each component, app developers override callback functions, which are commonly used to maintain its lifecycle, as depicted in Figure 1. These callbacks are expected to be automatically invoked by Android application manager. Therefore, symbolic execution faces a severe challenge because of the non-deterministic and unbounded triggering order of callbacks. For example, a possible execution could be *(OnStart ⇒ OnPause ⇒ OnResume ⇒ OnPause ⇒ OnResume ⇒...)*. It will further worsen the already notorious search space explosion problem of traditional symbolic execution. Actually, symbolic execution may never finish because the search space is infinite. We propose a guided symbolic execution mechanism which can effectively solve this problem with static analysis.

Android Events: GUI Events and System Events. An app running on Android is commonly GUI based, and its execution is typically driven by events from the specific GUI controls (represented as a *View* object) that the user interacts with. An app contains a collection of nested interfaces, called event listeners. These listeners capture user interactions with the app GUI. When respective interactions occurs on the GUI controls, for example, if a button is clicked by a user, the pre-defined event handlers are triggered correspondingly. System events are handled in the same way. Like callbacks, runtime events are also non-deterministic. They can be triggered in any order and at any time, thus exhaustively executing all possible sequences of events is a task that will never end. Fortunately, events in an Android app are commonly invoked when the state of the app is *RUNNING*. In this state, the main thread is hung to wait for incoming events. Thus, the event triggering behavior commonly depends on the order, not the exact triggering time.

3. GOAL AND OVERALL ARCHITECTURE

AppIntent is not an automated method to detect unintended data transmission, which is probably a mission impossible. Instead, as a first step in this space, AppIntent is designed to be an automated tool to present to a human analyst the sequence of UI manipulations that corresponds to the sequence of events that leads to the sensitive data transmission, thereby facilitating the discrimination of whether sensitive data transmission is user intended or not.

Our Goal. To achieve our vision, we have the following three goals:

- Produce the critical app inputs that lead to sensitive data transmission. Specific to Android GUI apps, inputs are always composed of: a) Data inputs which contain text inputs from outside; b) Event inputs from user interactions through GUI interface and from system through IPC. In addition, we need to track down the root-cause that gives rise to the transmission and filter out the massive set of irrelevant inputs.
- Guarantee a good code coverage. To find all feasible paths, we need to thoroughly traverse diverse program paths that may lead to a leakage, and at the same time, we want to ensure low false positive as well as low false negative rate during this analysis. In addition, to enable large-scale validation tasks, we do not want too much overhead.
- Provide an easy-to-understand tool for human analysts to ascertain under what circumstance the sensitive data transmission happens. Using the produced app inputs, we need to conduct the execution of an app according to each feasible path. We want to exercise the app's functionality automatically, which can emulate users' operations, and by observing the UI manipulation and prompting, we can then easily judge whether the data transmission is essential for a user-intended functionality.

Overall Architecture. Figure 2 depicts the overall architecture of AppIntent, which analyzes a target app in two steps:

- *Event-space Constraint Guided Symbolic Execution.* The first step is to generate critical inputs incurring sensitive data transmission. We adopt static taint analysis to preprocess and extract all possible data transmission paths as well as possible events related to each path, which helps to construct an event-space con-

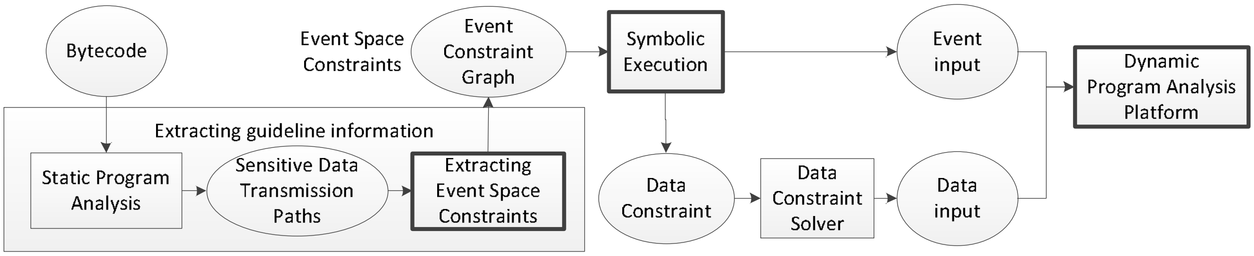


Figure 2: Overall Architecture of AppIntent

straint graph. Subsequently the graph is used in the guided symbolic execution to extract critical inputs. Meanwhile, code coverage is guaranteed due to the nature of symbolic technique. The detail is introduced in Section 4.

- *Dynamic Program Analysis Platform.* Inputs generated in the first step is not intuitive enough though they precisely tell under what conditions transmission would happen. Using these inputs, we adopt Android `InstrumentationTestRunner` [1] to automate the app execution step by step, which reflects users' interactions in UI manipulations, and the sensitive data propagation is also tailored to the related UI for a better understanding. We believe it can effectively visualize the root cause of the transmission so that we can intuitively judge whether the transmission is user intended or not.

4. EVENT-SPACE CONSTRAINT GUIDED SYMBOLIC EXECUTION

In this section, we present our event-space constraint guided symbolic execution technique for Android apps. We show how to reduce the search space considerably and finish the symbolic execution in an acceptable amount of time without sacrificing the code coverage.

We begin with an intuitive example, and then present an overview of this stage, followed by a detailed description of how to construct the event-space constraint graph using static analysis. Finally we describe how the graph facilitates guided symbolic execution.

4.1 A Concrete Example

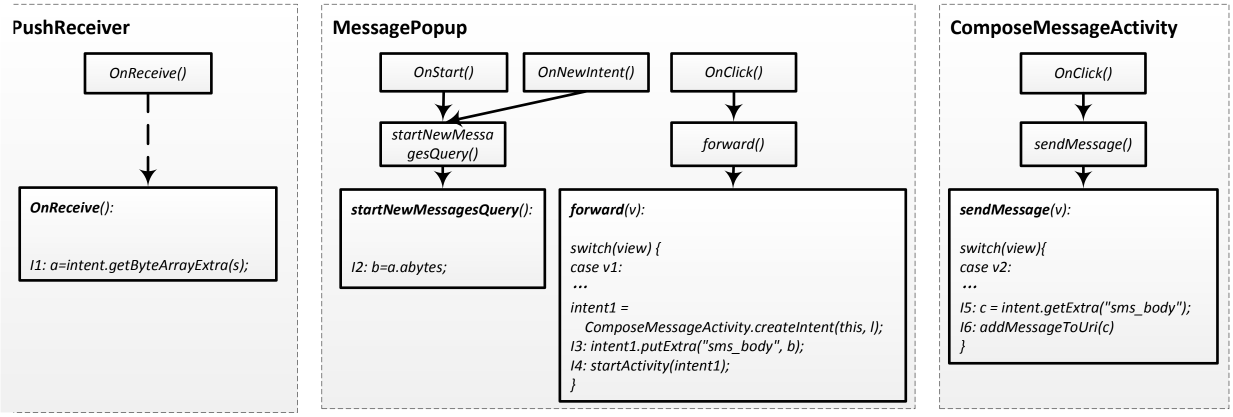
Here we use an app, *Anzhuoduanxin* [3], to demonstrate how our event-space constraint guided symbolic execution works. The app has a program path containing the transmission of an SMS message when a user forwards a new incoming message. For easy understanding, as depicted in Figure 3, we simplify the data propagation to a path involving only one BroadcastReceiver, *PushReceiver*, and two activities, *MessagePopup* and *ComposeMessageActivity*. The new message is handled in the *onReceive()* method of *PushReceiver* that starts up the activity *MessagePopup*, and the message is displayed in the foreground on which a user can click the FORWARD button to invoke the *forward()* method that starts up the activity *ComposeMessageActivity*. On the next user interface, the user can click the SEND button to invoke the *sendMessage()* method to have the message forwarded.

In our symbolic execution, we first use static taint analysis to identify all possible transmission paths, and then we extract instructions of sensitive data propagation with the context information along each path. In our example, we get the path: $\{OnReceive, \text{i1}\} \Rightarrow \{startNewMessagesQuery, \text{i2}\} \Rightarrow \{forward, \text{i3}\} \Rightarrow \{forward, \text{i4}\} \Rightarrow \{sendMessage, \text{i5}\} \Rightarrow \{sendMessage, \text{i6}\}$. Then we construct an event-space constraint graph according to the information gathered in static analysis. As Figure 4 shows, those massive irrelevant events to this path have been filtered out, and only 18 events related to this path, including lifecycle callbacks, GUI events, and system events, are kept. We connect these events with edges according to the lifecycle state transition and the call graph. This event-space constraint graph is used as a guideline for symbolic execution to find sequenced events that possibly incur the transmission. Since our goal is to find the root cause and disclose the context of the user actions, we only need to find the shortest paths that cover the sensitive data transmission instructions respectively. As Figure 5 shows, for the given transmission, we get only two chains of events in sequence, which will be verified during symbolic execution, with a very small overhead. On our dynamic program analysis platform, the feasible chain is used to emulate a user's operations step by step automatically, which demonstrates which functionality is executed when sensitive data transmission happens. In this case, we can easily determine that this is indeed user-intended data transmission.

4.2 Overview of Event-space Constraint Guided Symbolic Execution

As stated earlier, the major challenge symbolic execution faces is the problem of space explosion, which is dramatically worsened by the Android GUI interaction and execution model. A complete app-wide symbolic execution is not scalable due to the large number of possible events. Actually, to achieve sensitive data transmission, usually only a small portion of events will be triggered in sequence, along with sequenced instructions that propagate the data. This motivates us that if we are provided with a set of instructions that possibly incur the transmission, we only need to consider and extract the events that may trigger at least one instruction of the set, as well as the possible prerequisites of these events. In this way, the event search scope can be greatly limited to those related events instead of massive irrelevant events while code coverage is guaranteed. We construct an event-space constraint graph aided by static analysis, and it facilitates symbolic execution in finding possible sequences of events that are used to reproduce the transmission.

In the following, we first give a definition of this special graph, and then explain how to obtain this graph by static program analysis.



```
OnReceive():

I1: a=intent.getByteArrayExtra(s);
```

```
startNewMessagesQuery():

I2: b=a.abytes;
```

```
forward(v):

switch(view) {
case v1:
...
intent1 =
   ComposeMessageActivity.createIntent(this, l);
I3: intent1.putExtra("sms_body", b);
I4: startActivity(intent1);
}
```

```
sendMessage(v):

switch(view){
case v2:
...
I5: c = intent.getExtra("sms_body");
I6: addMessageToUri(c)
}
```

Figure 3: A simplified SMS forwarding case.

4.3 Construction of the Event-space Constraint Graph

As depicted in Figure 4, the event-space constraint graph is a directed graph, with each node in the graph representing a lifecycle callback, a GUI event, or a system event. There are two kinds of nodes:

- A thick-line node represents an event of which the event handler method contains at least one instruction of a given data propagation path. We call this kind of events *critical events*.
- A thin-line node represents an event which is a prerequisite for a critical event, and it does not contain any instructions of the given path. Such an event could be either a lifecycle callback of the activity that contains this critical event, or an event belonging to any prerequisite component that eventually starts up the activity that contains this critical event. We call this kind of events *essential events*.

A directed edge in the graph represents the order of precedence for two adjacent nodes. Edges can be calculated according to the lifecycle state transition and the call graph together.

Basically, for the graph, we ensure:

- All critical events should be included.
- All lifecycle callbacks of an activity that contains a critical event should be included.
- Any event belonging to a prerequisite component that eventually starts up an activity containing a critical event should be included, as well as its lifecycle callbacks.
- No edge violates the predefined order of the lifecycle state transition or the sequence of the call graph.

4.3.1 Extracting Critical Events

To build the the event-space constraint graph, first of all, we need to extract all critical events according to the given data transmission path. For each instruction in the path, we backward traverse the call graph to find all events that might trigger it. As shown in Figure 3, backward traversing the call graph from instruction 2 (i2), we can get two critical events, *OnStart()* and *OnNewIntent()*. We may introduce some false positives due to the limitation of static analysis techniques, but symbolic execution can eliminate these false positives later. In this phase, we finally obtain sequenced critical events, $<PushReceiver$, $onReceive>$, $<MessagePopup$, $OnStart|OnNewIntent>$, $<MessagePopup$, $OnClick>$, and $<ComposeMessageActivity$, $OnClick>$.

An activity may have different views to lay out various user controls (e.g. buttons), on which a user interacts with the app, and user interactions of various views are usually handled by the same handler method. The above critical events that we have extracted are from only the call graph and does not have the information about views except the handler methods. It poses a difficulty for the later guided symbolic execution. To solve this issue, we build a program dependency graph, extract branch conditions for view parameters from the graph, and annotate the critical events with these conditions as the context information. As depicted in Figure 3, the extracted branch condition for i3 and i4 is `view==v1`. After that, if we find that a critical event involves different views, we divide this event into several thick-line nodes, with respect to each view. Other GUI events are handled in a similar way.

4.3.2 Extracting Essential Events

So far, we get all the critical events that contain the instructions of the given transmission path, but they are just the critical interior nodes to symbolically execute the path. According to the Android runtime execution model, we also need to collect the essential events that are the prerequisites to the critical nodes, in order to behave well during symbolic execution. For example, an execution can not directly invoke *OnResume()* before the app is activated by invoking *OnCreate()* and *OnStart()* in sequence. Actually, an app strictly follows the state transition order of the app lifecycle, as illustrated in Figure 1. For each critical event of a component, we first supplement those missing lifecycle callbacks with directed edges according to the origin order. And then, aided by the call graph, we supplement all prerequisite components that eventually start up the activity which contains a critical event, as well as edges produced according the call graph. Meanwhile, the corresponding lifecycle callbacks of these prerequisite components are added in. In Android, inter-component communications are implemented through *Intents*. Thus, if a component receives an intent from another one, we treat the sender of the intent as the prerequisite of the receiver component, and add a directed edge to represent their order. Especially, if an intent is used

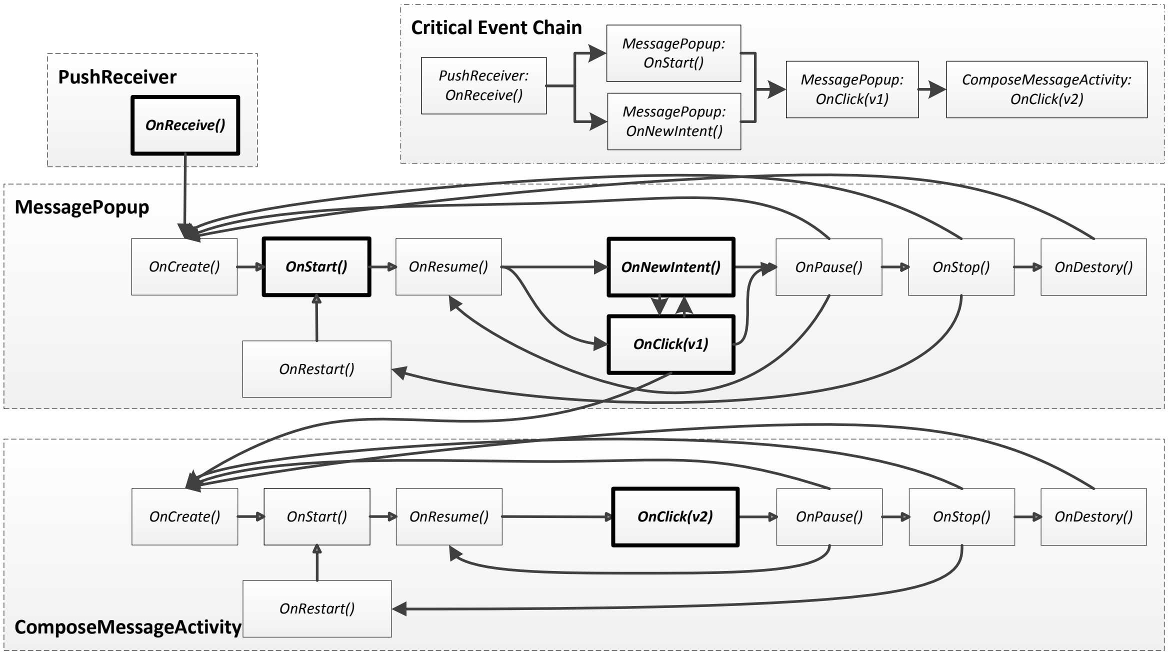


Figure 4: The extracted event-space constraint graph of the given example.

to start a new activity or service, the *onCreate()* callback of created component is marked as the receiver of the intent. In the current version of AppIntent, we only track intents that eventually start a new activity or service, as well as broadcast messages that are properly handled by a *BroadcastReceiver*, because they are officially documented by Google [2] and most of the apps use these two approaches to send intents. The intents with undocumented usage (which we envision could be handled in a similar way) are left to future work. The supplemental process will not end until there is no any prerequisite components found.

Upon finishing the above steps, we finish constructing the event-space constraint graph. As noted earlier in this section, for a given transmission path, all related events, including the critical and the essential events, have been extracted. The subsequent symbolic execution only needs to traverse this graph for sequenced events that possibly trigger the transmission and verifies whether it is a valid path. Since massive irrelevant events have been filtered out, the search scope is greatly reduced while the code coverage is guaranteed well.

4.4 Guided Symbolic Execution

We now explain how to traverse the above constructed graph to derive all possible sequenced events as the guideline to symbolic execution. The process of guided symbolic execution is depicted in Algorithm 1, in which P represents the events that are triggered before the last traversed critical event, and C represents the data constraints that should be fulfilled to reach the current execution point. If C is empty, then none of the data inputs can result in the target execution, i.e., the path can never be covered in any program execution.

Algorithm 1 works as follows. Our symbolic execution traverses the event-space constraint graph using the thick-line nodes as step stones. Each time when we proceed from a

```
$G \leftarrow$ Event-space Constraint Graph
$CEC \leftarrow$ Critical Events Chain of $G$
$C \leftarrow \emptyset$, $P \leftarrow \emptyset$
$StartPoint \leftarrow$ App Entrance of the Main Activity
Procedure TraverseGraph($ce$)
    forall the $ne : < ce, ne > \in CEC$ do
        $mp \leftarrow$ FindMinimalPath($ce$, $ne$, $G$)
        $C \leftarrow$ SymbolicExecute_Forward($C$, $mp$)
        $P \leftarrow P \oplus mp$
        if $\forall\ e : < ne, e > \notin CEC$ then
            Output: $C$ as Data Constraint, $P$ as Event Inputs
            exit()
        end
        else if $C$ != $\emptyset$ then
            TraverseGraph($ne$)
        end
        $C \leftarrow$ SymbolicExecute_Rollback($C$, $mp$)
        $P \leftarrow P - mp$
    end
end
TraverseGraph ($StartPoint$)
Output: No feasible inputs found
```

Algorithm 1: Event-space Constraint Guided Symbolic Execution

thick-line node, possible successors of this critical event are extracted from the event-space constraint graph. Since any of the successors can be the next critical event, we randomly pick an event first and calculate a feasible path from the current critical event to the chosen successor. Since only the essential prerequisites are needed, we extract the minimal path (using the Dijistra's algorithm) as a chain of events, which are sequentially triggered in the symbolic execution. If the event chain is revealed to be not available to any inputs ($C == \emptyset$), or all possible successors in critical event chains are already explored ($\forall\ e : < ne, e > \notin CEC$), we rollback the symbolic execution and try to trigger other feasible critical events.

Using Figure 4 as an example, guided by the event-space constraint graph, our symbolic execution explores a much smaller event space, as illustrated in Figure 5, and reports the following event chain as event inputs: { <*PushReceiver*, *OnReceive*>, <*MessagePopup*, *OnCreate*>, <*MessagePopup*, *OnStart*>, <*MessagePopup*, *OnResume*>, <*MessagePopup*, *OnClick*(v1)>, <*ComposeMessageActivity*, *OnCreate*>, < *ComposeMessageActivity*, *OnStart*>, < *ComposeMessageActivity*, *OnResume*>, < *ComposeMessageActivity*, *OnClick*(v2)> }. In addition, by using a modified version of `choco` data constraint solver [6], we generate corresponding data inputs according to the data constraints calculated in the symbolic execution.

5. DYNAMIC ANALYSIS PLATFORM

By using our event-space constraint guided symbolic execution, we can extract app inputs to trigger a given sensitive data transmission path. Although these inputs provide all the preconditions of target data transmission, they might not be intuitive enough for human to understand. To display these preconditions in an easy-to-understand manner, we set up a dynamic analysis platform to present which functionality is used when the transmission happens. With the help of Android `InstrumentationTestRunner` [1], a driven execution can be conducted for each sensitive data transmission path. AppIntent automatically generates a test case based on the inputs gathered before, and attaches it to the app by repackaging the original Android apk. Then, by running the test case though the Android activity manager, a controlled execution with the following features are presented:

- *Automatically trigger Event Inputs.* Events in the event chain are automatically triggered by performing corresponding operations. For example, to trigger a clicking event, a `performClick` operation is applied to the corresponding view, and we call the `setTestProviderLocation` method for a location change event. AppIntent currently does not support runtime events like phone call events because Android `InstrumentationTestRunner` does not support them. Since in most cases, view context of each event is already attached to the event chain, we can use the attached context directly for GUI events. On the other hand, if there is no view constraint, we randomly pick a view from the manifest file as the context of event. In addition, between each two GUI events, we generate a short delay so that analysts have time to observe the GUI display of each step.
- *Automatically provide Data Inputs.* Most of the data inputs generated by symbolic execution are text inputs to GUI elements, and we directly set the corresponding text field with the expected value. Some app inputs are messages from system or other apps, so we can attach these inputs to corresponding event messages. Besides, some apps trigger specific behavior based on the wall time of the Android system. For example, some malicious behavior happens only if a certain amount of time passed in the current execution. We explicitly generate sleep operations if data constraint relies on the current system time. In the current version of AppIntent, we do not support network inputs because we generate test cases through Android `InstrumentationTestRunner` [1], which cannot intercept and modify network inputs. This could be improved by hooking the network interfaces in the Android framework, which is our future work.
- *Highlight activated views of GUI events.* Activated view of each GUI event provides essential context, which represents a GUI element on the screen, for each user interaction. For example, if a clicking event is triggered, we need to know what element on the user interface is clicked by user. Thus, AppIntent highlights GUI element by setting its background color to red, as depicted in Figure 7(a) and Figure 7(b). For GUI elements whose view cannot be obtained by Android `InstrumentationTestRunner`, e.g., the list items, we highlight these elements by triggering some dialog box to display the view information.
- *Highlight sensitive data read and transmission.* To figure out whether the functionality of the app requires sensitive data transmission, our controlled execution needs to reveal when the data loading and transmission happen during the presented event chain. We highlight these two execution points by raising a notification dialog box, as depicted in Figure 7(c) and Figure 7(d).

6. EVALUATION

In the implementation of AppIntent, we first leverage DED [23] to decompile Android DEX files into Java bytecode. We implement our event-space constraint graph extraction on top of soot [8] and the guided symbolic execution engine on top of JavaPathfinder [10]. We implement the controlled execution and dynamic analysis platform on top of `InstrumentationTestRunner` [1]. In this section, we present our evaluation results on the effectiveness and accuracy of AppIntent. In our evaluation, the event-space constraint guided symbolic execution uses an Intel Xeon machine with 2 eight-core 2.0Ghz CPUs and 32 GB physical memory, which runs Debian Linux with kernel version 2.6.32. The controlled execution of AppIntent is run on Android 2.3.

6.1 Evaluation Methodology

In order to evaluate the effectiveness of AppIntent and its key techniques, we need to answer the following two questions: (i) When producing app inputs leading to some sensitive data transmission, to what extent does event-space constraint guided symbolic execution reduce the search space while guaranteeing the code coverage? (ii) Using the controlled execution based on app inputs, how effective is AppIntent to distinguish unintended data transmission with user-intended one?

In the following, we evaluate the execution time of symbolic execution with or without our technique to answer the first question and use two sets of real-world Android apps

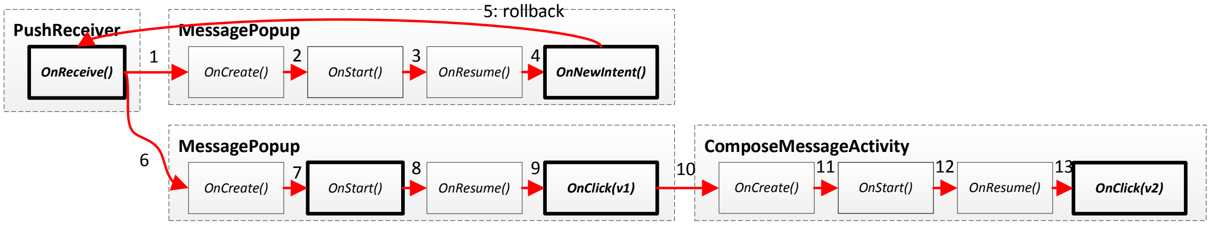


Figure 5: Event chains explored in symbolic execution by traversing the graph in Figure 4

to answer the second one. Besides, we provide some findings about sensitive data transmission patterns which are revealed by the result of AppIntent.

6.2 Effectiveness of Event-space Constraint Guided Symbolic Execution

To illustrate the effect of event-space constraint guided symbolic execution, we choose 3 famous apps from Google Play as samples. Among them, Maps is the Google map, Youlu is an SMS management app, and WeChat (a.k.a. Weixin) is a popular chatting tool. As presented in Table 1, without the help of event-space constraint graph, symbolic execution cannot finish in 5 days when we explore only 20 triggered events for the target app. This clearly demonstrates that exhaustively exploring event space is not scalable and practical. On the other hand, symbolic execution cannot cover the critical events in *WeChat* in this configuration, and failed to cover critical events in two cases(*Youlu* and *WeChat*) if we lower the threshold to 10 events. This means that naively limiting the search space can damage the effectiveness of symbolic execution. However, guided by event-space constraint graph, normally less than two hours are needed to explore the limited exploration space, and extract app inputs corresponding to the sensitive data transmission, without sacrificing the code coverage. Thus, it is clear that compared to existing approaches, guided symbolic execution proposed in this paper can greatly increase the exploration effectiveness.

Case	Origin (10 events) (hours)	Origin (20 events) (hours)	AppIntent (hours)
Maps	5.43	>120	0.40
Youlu	0.97	>120	0.13
WeChat	21.56	>120	1.33

Table 1: Running time of symbolic execution. Column 2 and 3 represent the running time when symbolic execution explores ten or twenty triggered events without the help of our event-space constraint guided symbolic execution. Column 4 shows the execution time of AppIntent.

6.3 Effectiveness on Analyzing Sensitive Data Transmission

In this experiment, two sets of real-world Android apps are selected to evaluate the effectiveness of AppIntent. The first set contains 750 malware apps from [46], which are known to perform malicious activities such as information leakage, money stealing, and privilege escalation. The second set contains 1,000 top free Android apps downloaded from Google Play. To compare with state-of-the-art privacy leakage approaches, we evaluate the same test datasets with TaintDroid [22] driven by MonkeyRunner. The results are depicted in Table 2. To verify the result of AppIntent, we perform manual analysis, in which we not only check whether apps reported by AppIntent transmit sensitive data, but also verify whether apps eliminated by AppIntent do not contain sensitive transmission.

As from the table, static taint analysis (the first step of AppIntent) detects 582 (442+140) cases of possible sensitive data transmission from two datasets. We find that 164 cases are false positives, which are eliminated by the next step of AppIntent, guided symbolic execution. With a manual analysis of the code of these programs, we find that most false positives in static analysis are caused by the insufficient context information and dead code, such as debugging code wrapped by *if(debug)* branches. There are another 44 cases from static analysis that failed to pass our symbolic execution. A further investigation shows that DED is unable to transform 42 cases from dex format to Java classfile, and the other two cases contain native code that currently cannot be handled by AppIntent. To check whether the app inputs generated by symbolic execution trigger sensitive data transmission, we applied manual analysis on the result of symbolic execution, and found that all cases transmit sensitive data defined in this paper.

With the app inputs extracted in symbolic execution, AppIntent successfully generates controlled executions for 358 (288+70) apps, among which, 245 (219+26) have been identified as unintended data transmission. We notice that the top free apps still have user unintended leakages, among which most apps are SNS (Social Networking Service) apps or apps that have embedded advertising modules. The leakage of SMS and contacts are all found in SNS apps. On the other side, malware may also contain both user intended and non-intended transmission, because malicious data leakage can hide behind some normal data transmission to bypass the state-of-the-art security validations. For example, we found that an application acts like an SNS app in disguise, but in the background, it stealthily transmits user contacts without user consent. It is worth noting that the current version of AppIntent failed to execute test cases of 43 apps because they are driven by network input, which is not supported by `InstrumentationTestRunner`. This could be further supported by instrumenting control code in Android framework.

As a comparison, TaintDroid can only detect 165 (125+40) cases as possible privacy leakage, most of which are leakage of device IDs. This is much less than AppIntent. Furthermore, the result of TaintDroid is hard to verify because it does not contain corresponding app inputs. Through our manual investigation, among these cases, 151 cases are also

	Malicious Apps				Google Play Apps			
Source	AppIntent (Static/ Symbolic/ Controlled Execution)	Unintended/ Intended Data Transmission	Local Logging	TaintDroid	AppIntent (Static/ Symbolic/ Controlled Execution)	Unintended/ Intended Data Transmission	Local Logging	TaintDroid
Device ID	389/256/ 246	198/0	73	101	98/43/43	24/0	19	37
Phone Info	53/50/50	50/0	1	0	0/0/0	0/0	0	19
Location	76/68/67	46/4	18	11	36/15/15	0/13	2	5
Contacts	13/13/13	1/10	2	0	10/10/10	1/9	1	3
SMS	27/27/17	16/3	0	0	9/8/8	1/7	0	0
Total	442/304/288	219/17	74	125	140/70/70	26/29	22	40

Table 2: Sensitive data transmission apps detected. The first part depicts the results of the chosen malware, and the second part are results of apps from Google Play. For each dataset, the first column represents the type of sensitive data transmitted, while Column 2 depicts the reported data transmission cases after each phase of AppIntent. Column 3 presents the number of data transmission of each kind. Column 4 depicts the number of sensitive data written to the local logging system. Column 5 lists the number of possible leakage cases detected by TaintDroid.

covered by AppIntent while 14 cases not. We manually checked the code of ten apps reported by TaintDroid but not reported in AppIntent, and found that nine of them do not actually leak privacy information. The remaining four cases either failed in DED, or contain native code that is not covered by AppIntent. In the 151 cases that are reported by both AppIntent and TaintDroid, 20 cases are actually classified as user-intended data transmission by AppIntent, which means they are not true privacy leakage. Since TaintDoird does not provide corresponding app input to trigger the sensitive data transmission, it cannot distinguish user-intended data transmission from unintended one.

In addition, we also have several interesting findings:

Finding 1: Data transmission of device IDs and phone numbers are very common but typically not noticed by most smartphone users. Among the detected unintended data transmission in the two selected datasets, most cases are transmission of device IDs or phone numbers. We also notice that almost all data transmission cases of device IDs and phone numbers do not inform users the operation. We believe that it occurs because Android apps use such information as the unique user identifier when connected to their own server.

Finding 2: Lots of apps write sensitive data into local logging system. Among the tested datasets, 96 (74+22) apps log sensitive data into local logging system, which is bad practice and may lead to indirect privacy leakage. Additionally, we find that not only device IDs and phone numbers are written to Android logs, but also locations and user contacts are temporally stored in several cases. These logged data can be leveraged by malicious apps that steal Android log instead of transmitting sensitive data directly. Since privacy leakage detection approaches do not cover leakage of local logging, such apps could bypass existing detection tools.

6.4 Analysis Time

Our static analysis phase costs 96 hours to analyze all 1,750 apps, among which 70 hours are used in static taint analysis. The analysis time can be further reduced by distributing the analysis workload to multiple machines. Since each application costs about 3.3 minutes on average, the analysis time is almost negligible to the Android market operators.

Our symbolic execution costs 5 to 134 minutes to verify a certain path reported by static analysis, depending on the search space and the complexity of the app. Verifying different paths can also be processed in parallel because exploring the possible search space of each sensitive data transmission path does not depend on information of other paths. As an offline analysis tool, such a validation time is also acceptable to the marketplaces that have enough computing power.

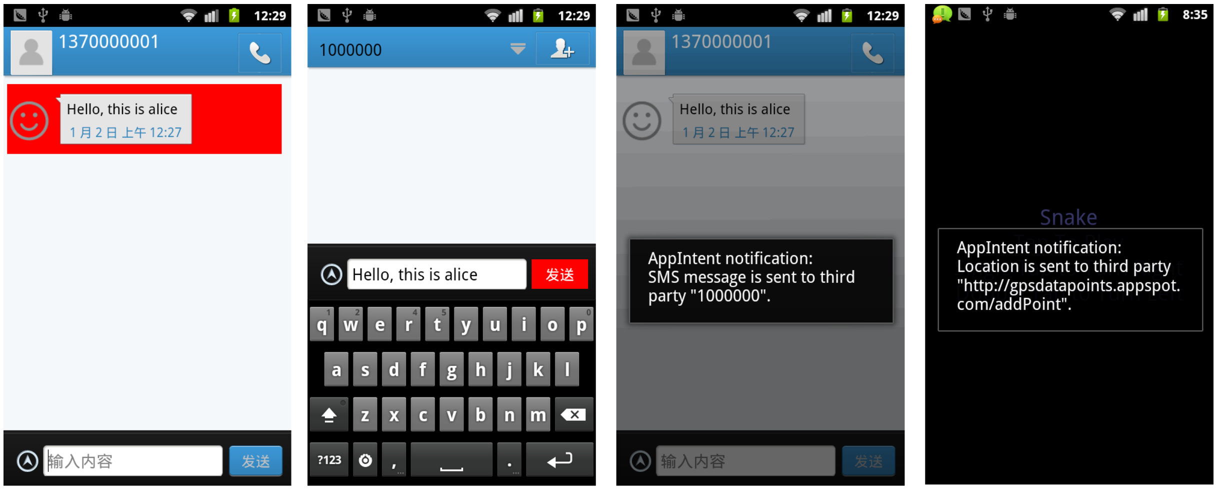
6.5 Case Studies

We now present two case studies from our evaluation: one represents user-intended data transmission (*Anzhuoduanxin*) and the other represents unintended transmission (*Tapsnake*). Video demonstrations of AppIntent for both cases are available at [4, 5].

Anzhuoduanxin [3] is an SMS management app that provides a set of SMS-related functions such as creating new messages or forwarding a cached message to another user. With the help of our event-space constraint guided symbolic execution, AppIntent generates two feasible app inputs that trigger sensitive SMS data transmission: Figure 6(a) depicts one of the feasible inputs, and a simplified version of the other is depicted in Figure 5. Without loss of generality, we choose the first one to illustrate here.

Our dynamic analysis platform accepts this input and creates an execution as demonstrated in the video [4]. In this case, the controlled execution first selects a record among the list which represents all conversation records stored in this phone. Then, by choosing a message and clicking a button that presents ”forward” in Chinese (Figure 7(a)), the app user can forward this message to someone else. This message can be sent to anyone by typing a named receiver and clicking a button titled ”send”(Figure 7(b) and Figure 7(c)). This execution is commonly used for forwarding a stored message to a friend of the app user, thus it should not be classified as malicious/unintended behavior.

Tapsnake is a malicious app that stealthily transmits user locations to a predefined third party receiver. Depicted in Figure 6(b), the app input generated by AppIntent shows that two components are activated when the location information is transmitted to a third-party user in Tapsnake: the


12:29
1370000001
Hello, this is alice
1月2日 上午12:27
输入内容
发送

(a)

12:29
1000000
Hello, this is alice
发送

(b)

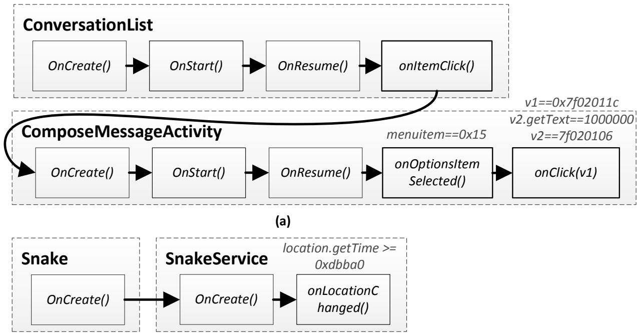
12:29
1370000001
Hello, this is alice
1月2日 上午12:27
AppIntent notification:
SMS message is sent to third party "1000000".
输入内容
发送

(c)

8:35
Snake
AppIntent notification:
Location is sent to third party "http://gpsdatapoints.appspot.com/addPoint".

(d)

Figure 7: Screen shots of case studies.



(b)

Figure 6: Feasible app inputs for sensitive data transmission in case studies.

main activity of this app, and an embedded Service which registers an event listener for location change event. Based on this input, the corresponding execution, demonstrated in the video [5], waits until the current time is greater than 0xdbba0 (which represents 15 minutes from the beginning of the wall time) Then, the location information is sent after a location change event is performed (Figure 7(d)). Since the original application is a simple ”snake” video game, and its functionality does not depend on the location information, thus this behavior is unintended.

6.6 Usability of AppIntent

To evaluate how useful the information provided by AppIntent is, we randomly selected 100 cases reported, and used them to evaluate the user experience. Since AppIntent mainly focuses on providing enough information for discriminating user-intended data transmission from unintended one at app markets, we invited three Android experts in our usability study. During the evaluation, we first introduced AppIntent to them with less than 15 minutes, and let them get familiar with the given cases. Then, we ran the driven executions generated by AppIntent in our Samsung Nexus S mobile phone and showed them to all three participants. After that, participants were asked to fill a sheet in which each case should be classified as ”user-intended” or ”unintended”. We find that they can make their decision in less than one minute after the driven execution finishes, which shows that AppIntent greatly speeds up the process in validating Android apps.

The results from these three users are unambiguously the same as our judgement in 98 cases. However, there are some different opinions in two cases, which are both data transmission of IMEI. Two out of three expects classified them into user-intended because they think these apps need the IMEI number to fulfill their functionality, while we classified them as unintended data transmission because there is no direct relation between the data transmission and the user experience. This evaluation shows that AppIntent is still a great assistance tool with high usability in practice. And it certainly also has some room to be improved in the future.

7. RELATED WORK

AppIntent seems to be the first to systematically study a method to separate user-intended Android data transmission from unintended ones. All other existing Android privacy leakage detection approaches only detect sensitive data transmission. Static Taint Analysis [21, 40] focuses on identifying the possible privacy leakage path with the help of reachability analysis and program slicing. However, these approaches commonly introduce a lot of false positives and cannot separate user-intended operations from unintended ones because of lacking user intention and context information. On the other side, Dynamic Taint Tracking techniques [22, 41] track the sensitive data at runtime by instrumenting profiling code to the original app code. They cannot be applied to automatically detect privacy leakages in marketplaces because they report leakage only if such dangerous propagation happens to occur in the execution. While not implemented, Vision [26] argues that user granting of sensitive data usage can be represented by End-user license agreements(EULA) and explicit notification during the execution. Similarly, BLADE [32] detects web drive-by download malware by recognizing whether it has user consent or not. However, mobile apps commonly do not provide EULA or notification even if the data transmission is user-intended (e.g. SMS forwarding). Pegasus [18] detects malicious behaviors that can be characterized by the temporal order in which an application uses APIs and permissions, and similar to this paper, it focuses on detecting malicious app behaviors that are inconsistent with the GUI events. Nevertheless, privacy leakages cannot be modeled as app usage of permissions or APIs, thus many privacy leakages cannot be detected by such approach. Besides, Pegasus verifies program behaviors based on application-specific properties, which are difficult to specify without the knowledge of application code. Recently, VetDroid [44] enhances Dynamic Taint Tracking by generating specifications for sensitive operations. However, the specification mainly focuses on the application logic but does not pay attention to the trigger condition of each operation.

AppIntent needs to extract app inputs to distinguish user-intended data transmission from unintended one. SmartDroid [45] proposes a hybrid static and dynamic analysis method to reveal UI-based event trigger conditions based on sensitive Android APIs. However, in order to generate a reproducible driven execution, we need both event inputs and data inputs. In this sense, AppIntent provides a more complete and systematic approach. Besides, instead of Android APIs, we need finer-grained analysis of app behaviors to detect privacy leakages. AppIntent proposes a symbolic execution approach for Android GUI apps to extract inputs. The search space explosion of symbolic execution is a well-known issue. Earlier guided symbolic executions direct the exploration with static analysis result [11, 33, 36, 37] or profiled program behavior[14, 19, 43]. All these approaches focus on the explosion caused by the data input space, and cannot reduce the search space of runtime events in Android. On the other hand, there is little work on limiting the event space. In order to limit the search space, Ganov, et.al [25] set an upper bound to the number of event sequences generated, Kudzu [38] used a random generated event order, and Ganov, et.al [24] proposed to generate test cases by symbolically executing each event handler separately. Although these features can limit the exploration of event space, they greatly sacrifice the code coverage. Contest [9] seeks to prune redundant event sequences by checking *subsumption conditions*, and can reduce the running time of symbolic execution to 5%-36% of the original execution time. However, the event space is still large after the pruning and the path explosion problem still exists. To the best of our knowledge, all existing approaches either trade accuracy for performance or suffering poor scalability. In this paper, by using the result of static analysis as the guideline, event-space constraint guided symbolic execution explores event space efficiently without sacrificing the accuracy.

8. CONCLUSION AND FUTURE WORK

This paper addresses one of the major challenges faced by smartphone markets - how to detect privacy leakage in Android apps. Unlike previous approaches that simply consider the transmission of private data as privacy leakage, we argue that such transmission may not indicate a true privacy leakage, instead, a better indicator should be whether the transmission is user intended or not. We present AppIntent, a new app validation framework to help human analysts determine if data transmission is intended by the user. With the help of event-space constraint guided symbolic execution technique proposed in this paper, the search space of symbolic execution is effectively bounded so that AppIntent can extract app inputs that represent user interactions in an acceptable amount of time. With the help of the dynamic analysis platform, AppIntent can also intuitively display the context information of the sensitive data transmission.

Our current techniques have the following limitations, which are also our future work. First, native code is currently not supported by AppIntent. Thus, privacy leakages in native code cannot be captured. Second, since the Android `InstrumentationTestRunner` [1] does not support instrumentation of network input, our dynamic analysis platform cannot simulate network inputs generated by symbolic execution. This could be solved by instrumenting code in Android network interface. Finally, AppIntent fails to analyze some apps because the DEX decompilation tool, DED [23], failed to decompile these apps. We plan to use Dexpler [12], which can directly parse DEX files, in soot, so that the decompilation from DEX to Java bytecode is not needed.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, and ChenHao Qu for his assistance in experiments. This work is funded by China National Natural Science Foundation under grants numbered 61103078 and 61300027, grants from the Science and Technology Commission of Shanghai Municipality numbered 11DZ2281500, 11511504404, 135-11504402 and 13JC1400800, a research grant from a joint program between China Ministry of Education and Intel numbered MOE-INTEL201202, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project numbered B114. This work is partially supported by the National Science Foundation under Grant no. CNS-0954096.

9. REFERENCES

[1] Android instrumentationtestrunner. http://developer.android.com/reference/android/test/InstrumentationTestRunner.html.

[2] Android intent. http://developer.android.com-/reference/android/content/Intent.html.

[3] anzhuoduanxin. http://dx.91.com/.

[4] Appintent demo: Anzhuoduanxin. http://www.youtube.com/watch?v=RRqWQk4ztmI.

[5] Appintent demo: Tapsnake. http://www.youtube.com/watch?v=L4IvXzpYqzw.

[6] Choco data constraint solver. http://www.emn.fr/z-info/choco-solver/.

[7] Google map. http://www.google.com/mobile/maps/.

[8] soot analysis framework. http://www.sable.mcgill.ca/soot/.

[9] S. Anand, M. Naik, H. Yang, and M. J. Harrold. Automated concolic testing of smartphone apps. In *Proc. FSE*, 2012.

[10] S. Anand, C. S. Pasareanu, and W. Visser. Jpf-se: A symbolic execution extension to java pathfinder. In *TACAS 2007*, pages 134–138, 2007.

[11] D. Babic, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proc. ISSTA*, pages 12–22, 2011.

[12] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. SOAP*, 2012.

[13] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *CCS*, pages 575–586, 2011.

[14] P. Boonstoppel, C. Cadar, and D. R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *TACAS*, 2008.

[15] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability signatures. In *IEEE Symposium on Security and Privacy*, 2006.

[16] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[17] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS*, pages 322–335, 2006.

[18] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song. Contextual policy enforcement in android applications with permission event graphs. In *Proc. NDSS*, 2013.

[19] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, D. Song, and E. X. Wu. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security*, 2011.

[20] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, pages 117–130, 2007.

[21] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[22] W. Enck, P. Gilbert, 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. In *OSDI*, pages 1–6, 2010.

[23] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security*, 2011.

[24] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Test generation for graphical user interfaces based on symbolic execution. In *AST*, pages 33–40, 2008.

[25] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Event listener analysis and symbolic execution for testing gui applications. In *ICFEM*, 2009.

[26] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proc. MCS)*, 2011.

[27] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[28] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[29] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, 2012.

[30] 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 *CCS*, pages 639–652, 2011.

[31] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *ICSE*, pages 199–209, 2009.

[32] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proc. CCS*, pages 440–450, 2010.

[33] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proc. SAS*, 2011.

[34] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security*, pages 67–82, 2009.

[35] G. Patrice, Y. L. Michael, and A. M. David. Automated whitebox fuzz testing. In *NDSS*, 2008.

[36] N. Rungta, E. G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proc. SPIN*, 2009.

[37] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proc. ISSTA*, 2010.

[38] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. *Security and Privacy, IEEE Symposium on*, 0:513–528, 2010.

[39] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[40] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[41] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO*, pages 243–254, 2004.

[42] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.

[43] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, 2009.

[44] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *CCS*, 2013.

[45] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, and W. Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proc. SPSM*, October 2012.

[46] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.