WeChecker: Efficient and Precise Detection of Privilege Escalation Vulnerabilities in Android apps

Xingmin Cui
The University of Hong Kong
xmcui@cs.hku.hk

Jingxuan Wang
The University of Hong Kong
jxwang@cs.hku.hk

Lucas C.K.Hui
The University of Hong Kong
hui@cs.hku.hk

Zhongwei Xie
The University of Hong Kong
jasonxie@connect.hku.hk

Tian Zeng
The University of Hong Kong
zengtian@connect.hku.hk

S.M.Yiu
The University of Hong Kong
smyiu@cs.hku.hk

ABSTRACT

Due to the rapid increase of Android apps and their wide usage to handle personal data, a precise and large-scaling checker is in need to validate the apps' permission flow before they are listed on the market. Several tools have been proposed to detect sensitive data leaks in Android apps. But these tools are not applicable to large-scale analysis since they fail to deal with the arbitrary execution orders of different event handlers smartly. Event handlers are invoked by the framework based on the system state, therefore we cannot pre-determine their order of execution. Besides, since all exported components can be invoked by an external app, the execution orders of these components are also arbitrary. A naive way to simulate these two types of arbitrary execution orders yields a permutation of all event handlers in an app. The time complexity is $O(n!)$ where n is the number of event handlers in an app. This leads to a high analysis overhead when n is big. To give an illustration, CHEX [10] found 50.73 entry points of 44 unique class types in an app on average. In this paper we propose an improved static taint analysis to deal with the challenge brought by the arbitrary execution orders without sacrificing the high precision. Our analysis does not need to make permutations and achieves a *polynomial* time complexity. We also propose to unify the array and map access with object reference by propagating access paths to reduce the number of false positives due to field-insensitivity and over approximation of array access and map access.

We implement a tool, WeChecker, to detect privilege escalation vulnerabilities [7] in Android apps. WeChecker achieves 96% precision and 96% recall in the state-of-the-art test suite DriodBench (for compairson, the precision and recall of FlowDroid [1] are 86% and 93%, respectively). The evaluation of WeChecker on real apps shows that it is efficient (average analysis time of each app: 29.985s) and fits for large-scale checking.

WiSec'15, Jun 24-26 2015, New York City, NY, USA
 ISBN 978-1-4503-3623-9/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2766498.2766509.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Verification

General Terms

Algorithms, Security, Verification

Keywords

Android, Privilege Escalation Attack, Taint Analysis, Control Flow and Data Flow Checking

1. INTRODUCTION

Android has won a dominate market share in the smartphone market [19]. With Android devices being prevalent and their wide usage to handle private data, they have become an attractive target for malware developers. [7] discovered the privilege escalation attack in Android apps. The idea is that an application with less permissions can gain access to the components of a more privileged application. By making use of the privilege escalation vulnerabilities, the malicious app cannot acquire extra permission. However it can perform privileged functions or get sensitive data without asking for the required permission. To prevent this attack, an application must enforce additional checks to protect the permissions it has been granted. However, since most Android application developers are not security experts, there is a need to validate the permission flow of Android apps before they are listed on the market.

In view of the rapid increase of Android apps on the market, a precise and scalable checker is in need for large-scale analysis. Taint analysis can be used to check the leak of sensitive data by tracing sensitive data to see whether it will flow into interested sinks. It has been widely used in the validation of Android apps by various works [4, 13, 1]. In this paper we propose an improved static taint analysis to detect privilege escalation vulnerabilities in Android apps precisely and efficiently. Privilege escalation attacks can be classified into two classes according to [3]: confused deputy attacks and attacks by colluding applications. In this work we focus on confused deputy attacks, which concerns malicious apps leveraging unprotected interfaces of a benign application. We classify the leak paths that would lead to confused deputy attacks into two types: capability leak paths and sensitive data leak paths.

Capability leak paths start from an entry point of the vulnerable app and end at an action call which is protected by

permissions. By utilizing this path, an unauthorised app can also perform protected actions. Sensitive data leak paths start from retrieving sensitive data by invoking a permission protected data call and end at a sink through which the retrieved data can be leaked to an unauthorised app. This type of leak paths are further divided into three subtypes based on their sink methods. The first subtype leaks sensitive data by returning it to the unauthorised caller application. The second subtype attaches the sensitive data to an implicit intent which can be intercepted by the attacker. The third subtype writes the data to a given url or a log file which can be accessed by any application.

One issue that effects the precision of taint analysis is Android's event-driven programming paradigm, which enables the arbitrary execution orders of different user-defined event handlers. Besides, since all exported components are accessible by an external app, the execution orders of these components are also arbitrary. The naive way to simulate the two types of arbitrary execution orders yields the permutation of all event handlers in an app. Suppose there are n event handlers in an app, the time complexity would be $O(n!)$. The statistics of CHEX [10] discovers 50.73 entry points of 44 unique class types in an app on average. The permutation solution would lead to a large overhead. In fact, in the time evaluation of CHEX, 22% of the tested apps needed more than 5 minutes to be analyzed, and lead to timeout since they set a maximum processing time to be 5 minutes for a optimized throughput. Its performance decomposition indicates that the permutation process constitutes the majority of the time overhead. In this paper we propose a *summary based taint analysis* to reduce this part of overhead, and achieve a precise yet efficient analysis.

During the analysis, we record the set of current tainted variables as the context information. We consider three types of variables based on their scope: *Local variables*, which are only valid within the current method, are recorded in set L; *private class variables*, which are valid within the current class, are recorded in set C; *public class variables*, which are valid through the current app, are recorded in set G. For clarity, we denote the private class variables as *class variables* and public class variables as *global variables*.

We calculate two types of summaries: *method summary* and *class summary*. Method summary is used to deal with the arbitrary orders of event handlers in a component and class summary is used to simulate the arbitrary execution orders of different components. Since local variables are effective only within the current method, execution of other methods only effects the taint status of class variables and global variables. In a method invocation, the taint status of the formal parameters cannot be determined without giving that of the actual parameters. Therefore a method summary contains information related to class variables, global variables and method parameters. Once the method summary is calculated, later invocation of this method only needs to refer to the method summary and do not need to step into the method. A class summary only includes the information related to global variables.

We use a call graph to represent a component. User-defined event handlers that can execute in arbitrary orders are put between a pair of *EventBegin* and *EventEnd* nodes. The checker first extracts the method list in each component and calculates the summary of each method. After that traversal of the call graph is converted to traversal of each method summary. In the first round, the context passed to the method summary of each event handler is the context at the EventBegin node. In the second round, the effect of other event handlers that can execute before the current one are taken into account. We will define how to get the context passed to each event handler in the system design part. In this way we considered the interference of other event handlers without permutation. Class summaries are used to deal with the arbitrary execution orders of different components using the same two round analysis strategy. Our strategy reduces the analysis overhead from $O(n!)$ to $O(n^2)$. The time complexity analysis will be given in the system design part. The evaluation on DroidBench test cases and randomly downloaded apps shows that our strategy is both efficient and precise.

Another challenge that needs to be dealt with in static analysis is the resolution of array and collection access. Previous work based on static analysis [1, 13] usually conservatively mark the whole array or collection as tainted when one element gets tainted. This results in false positives when tainted data is stored in one position but data access occurs in another position. In this paper we propose to unify the analysis of array and map access with object reference using access paths. The access path is identified by fields (for an object), indexes (for an array) or keys (for a map). Through propagating access paths, we can distinguish different elements of an array or map, and different fields of an object, thus achieving field sensitivity and reducing the number of false positives.

We implement a tool, WeChecker, to automatically identify capability and sensitive data leaks in Android apps. The analysis of WeChecker is highly precise by being flow-sensitive, context-sensitive, object-sensitive and field-sensitive [18]. It performs context-sensitive alias analysis to get an accurate approximation of the tainted variables. Besides, it constructs invocation chains to connect different components in the analyzed app through analyzing intents for inter-component communication. Each invocation chain starts from one entry point of the app and includes all other components that are reachable from this entry point. Here we denote the *entry points of an app* as exported components which require no permission from other components to interact with them. In this way, WeChecker gets a list of components that can be accessed and exploited by external apps and at the same time, identifies leak paths whose source and sink methods reside in different components.

The contributions of this paper are as follow:

- First, we did a systematic study on classifying leak paths that can lead to privilege escalation and classify them into two types: capability leak path and sensitive data leak path.
- Second, we propose a summary based static taint analysis to deal with the arbitrary execution orders of different event handlers and different components. It saves the overhead to make permutations of the event handlers and components without effecting the analysis precision.
- Third, we propose to unify the array and map access with object reference using access paths. This effectively reduces the false positives result from field-insensitivity and over approximation.

- Fourth, we implement a checker, WeChecker, to detect leak paths in Android apps. It achieves a 96% precision and 96% recall on the state-of-the-art test suite DroidBench. Its evaluation on real apps randomly downloaded from GooglePlay shows it is efficient and fits for large scale analysis.

The rest of the paper is organized as follows: We will introduce related background knowledge and the motivating example in section 2; In section 3, we present the two types of leak paths that would lead to privilege escalation attacks; System design is introduced in section 4; System implementation and evaluation are presented in section 5. Related work is listed in section 6. We conclude the paper in section 7.

2. BACKGROUND AND MOTIVATING EXAMPLE

Android is an open-source mobile operating system based on the Linux kernel[14]. All Android applications consist of four kinds of components: activities, services, broadcast receivers and content providers. These components can provide different functionalities and work together to implement the designed function. The manifest file declares the components contained in the application and presents other essential information to the Android system.

The lifecycle of Android components is managed by the Android framework in an event-driven manner. Besides, Android provides Event Listener interfaces. Developers can implement these interfaces and override the callback methods in react to users' UI interaction or system state change (eg. locationUpdate). Therefore unlike traditional Java programs which use a single main method as the entry point, Android components can have many entry points. These entry points include Android lifecycle methods and user-defined event handlers. These methods are invoked by the Android framework at runtime and their order of execution cannot be determined in advance.

Taint analysis checks whether tainted variables can reach sink methods. If any tainted variables reach a sink method, an alarm will be raised. The example in figure 1 implements an Activity. In the method onCreate, three buttons (*button1, button2 and button3*) register their event handlers. *sData* is a class variable shared between different methods. The statements in each event handler will be executed when the user clicks the corresponding button. The user can click these buttons in arbitrary orders. To find the leak by *sink1* on line 8, the checker needs to consider the situation that the user may click button2 before clicking button1.

The event handler of button2 invokes *taintIt* on line 16 using the return value of a source method as the first parameter. To find the leak by *sink6* on line 40, the checker needs to perform alias analysis to find out that *out.f* also gets tainted on line 38. When the method invocation returns on line 16, the return value is assigned to *rs*. Since *taintIt* returns the first parameter which is tainted, *rs* is also tainted. Therefore *sink2* on line 21 should raise an alarm. Lines 17-19 create a String array and initializes its first two elements. To avoid a false positive on line 22, the analysis needs to differentiate *arrayS[0]* and *arrayS[1]*. After the invocation of *taintIt*, *d* is sanitized on line 20, hence *sink4* on line 23 will not lead to a leak. To avoid a false alarm, the checker needs to sanitize tainted values when needed.

```
1  public class ExampleAct extends Activity {
2  Data sData;
3  protected void onCreate(Bundle savedInstanceState
       ) {
4          sData = new Data();
5          Button button1 = (Button) findViewById(R.
               id.button1);
6          button1.setOnClickListener(new
               OnClickListener() {
7           public void onClick(View arg0)  {
8                  sink1(sData.f);
9           }
10         });
11
12         Button button2= (Button) findViewById(R.
               id.button2);
13         button2.setOnClickListener(new
               OnClickListener() {
14         public void onClick(View arg0)  {
15                 Data d=new Data();
16                 String rs=taintIt(source(), d);
17                 String[] arrayS=new String[10];
18                 arrayS[0]=rs;
19                 arrayS[1]="no taint";
20                 d=new Data();
21                 sink2(rs);
22                 sink3(arrayS[1]);
23                 sink4(d.f);
24                 }
25         });
26
27         Button button3=(Button) findViewById(R.id
               .button3);
28         button3.setOnClickListener(new
               OnClickListener() {
29         public void onClick(View arg0)  {
30                 Data d1=new Data();
31                 taintIt("no taint", d1);
32                 sink5(d1.f);
33                  }
34          });
35 }
36 String taintIt(String in, Data out) {
37         Data x = out;
38         x.f = in;
39         sData = x;
40         sink6(out.f);
41         return in;
42      }
43 }
```

Figure 1: Example of Android Component Entry Points

The event handler of button3 also invokes *taintIt* on line 31 but with the first parameter untainted. Therefore *sink6* on line 40 and *sink5* on line 32 will not yield any leaks. To differentiate the two invocations of *taintIt* on line 16 and line 31, the analysis needs to be context-sensitive.

From the example in figure 1, we conclude that a highly precise analysis needs to: (1) Take into account the arbitrary execution orders of user-defined event handlers in one Activity and arbitrary execution orders of different components in one app. (2) Perform context-sensitive alias analysis to get an accurate approximation of the tainted variables. (3) Precisely resolve array access to avoid false positives brought by over-approximation. Meanwhile, it needs to perform inter-component communication analysis to get the list of accessible components by external apps. According to [1], the analysis also needs to be be flow-sensitive, context-sensitive, object-sensitive and field-sensitive[18].

In the system design section we will explain how our analysis meets these requirements.

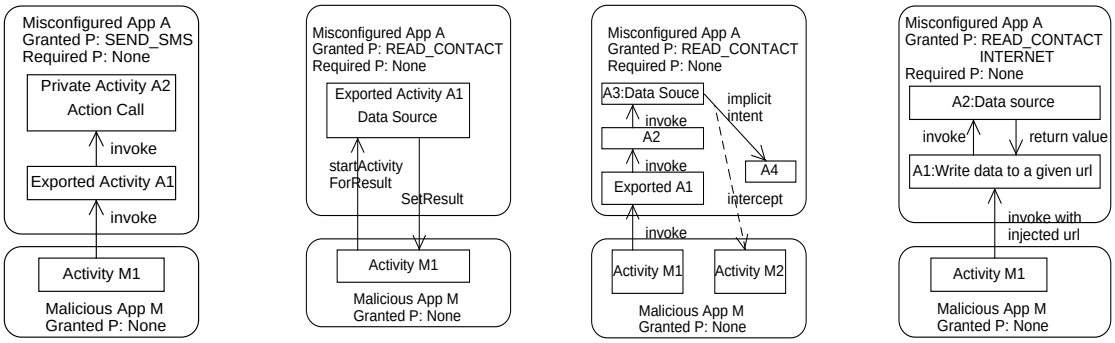

Figure 2: Leak path: type 1(leftmost), type 2.1(left), type 2.2(right), type 2.3(rightmost)

3. LEAK PATHS

Privilege escalation attacks can be enabled by making use of a chain of components (denoted as a leak path) through which the unauthorised application can perform permission protected actions or retrieve sensitive data using an authorised application as a deputy. Thus we check whether an application is vulnerable by checking whether leak paths exist in it. We use static taint analysis to catch the vulnerable application before it is delivered to users.

In previous studies, there does not exist a systematic study on classifying leak paths. In this paper, we classify leak paths into two types based on the nature of Android API calls. Android API calls can be classified into two types: action calls and data calls. The former has side effects on the system while the latter returns data without causing side effects. By utilizing the vulnerable app's privilege to invoke permission protected action calls, an unauthorized app gains the capability to invoke these calls. By utilizing the vulnerable app's privilege to invoke permission protected data calls, an unauthorized app can get access to sensitive data. Therefore we classify leak paths into capability leak paths (type 1) and sensitive data leak paths (type 2). To be more precise, we define **capabilities** as the ability to invoke permission protected action calls and **sensitive data** as data returned by permission protected data calls. Examples are the action call "sendTextMessage" which asks for the permission SEND_SMS and the data call "getLastKnownLocation" which asks for the permission ACCESS_FINE_LOCATION.

The first type of leak paths lead to capability leaks. Capability leak often occurs when a function which is intended to be used internally within an application can be invoked by external applications because of misconfiguration. This type of leak path starts from one entry point of the vulnerable app and ends at a permission protected action call. Figure 2 (leftmost) illustrates such a path. Misconfigured application A is granted the permission SEND_SMS but requires no permission from other applications to interact with it. The exported Activity A_1 serves as an entry point to app A for the malicious app M. As a result, app M acquires the capability to send SMS using A as a deputy. If the action call also takes an argument which is passed from the malicious app, the risk would be higher. The malicious app can manipulate the vulnerable app to alter the system settings to a designated state or browse a specified website.

The second type of leak paths lead to sensitive data leaks. It starts from one entry point of the vulnerable app and retrieves sensitive data by invoking a permission protected data call. However, this is not enough because the malicious app needs to gain access to this sensitive data to make the attack practical and meaningful. We classify the leakage of data to the attackers into three types. First (type 2.1), return the data to the invoking (malicious) application using the method *SetResult*. Second (type 2.2), send out the data using an implicit intent that can be intercepted by the malicious app. Third (type 2.3), write the sensitive data to external entities such as an url or log files which the malicious app or the attacker can get access to. Figure 2 shows these three subtypes of leak paths. Data sources are methods that can retrieve sensitive data. For type 2.1, the sink method is *SetResult*. Another requirement is that the Component A_1 in which the *SetResult* method resides must be exported and can act as an entry point to the misconfigured app. This is because SetResult can only return data to the direct invoking component which should be a component in the malicious app. For type 2.2, the sink methods are inter-component communication methods (eg.*startActivity*) which are attached with implicit intents. If the intent contains sensitive data, it can be hijacked by a malicious app. For type 2.3, the sink methods are methods that write sensitive data to a given url or to log files. For type 2.2 and type 2.3, both the source methods and sink methods can reside in private components. Since only exported components are accessible by external apps, these paths cannot be found without performing inter-component communication analysis.

4. SYSTEM DESIGN

We provide a tool, WeChecker, to check whether an app is vulnerable to privilege escalation attacks. Figure 3 gives an overview of WeChecker. It takes an apk file as input and decides whether this apk contains capability or sensitive data leak paths. If yes, it prints out each statement along this path from source to sink. Analysis based on some intermediate representations is easier than directly on bytecode. Our analysis is based on Jimple, a typed and compact 3-address code representation of bytecode [30]. We use Soot[1] to parse the apk files and get the Jimple representation.

WeChecker will perform the following procedures: (1)Parse

[1]http://sable.github.io/soot/

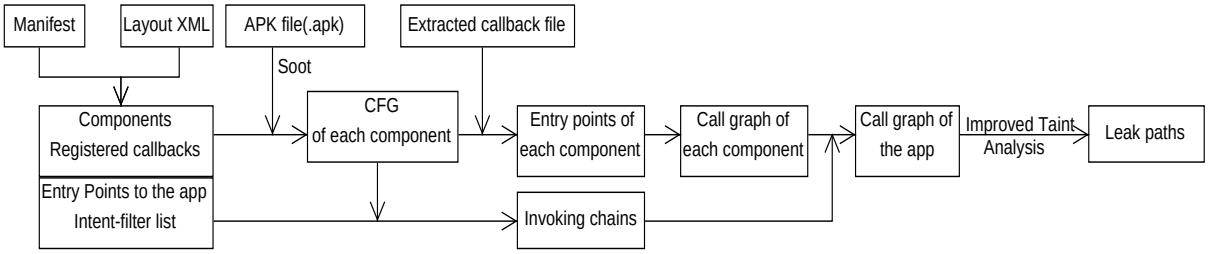

Figure 3: System Overview

the Manifest file and layout XML file to extract the list of components contained in the app, intent-filter list and callbacks registered in the layout file. If a component is exported and is not protected by any permissions, we regard it as an *entry point to the app*. If no entry point exists in the app, the checking will terminate and conclude it is secure since external applications are restricted to access the components in this app. If this app is protected by the same permissions (declared in the manifest file using the <*permission*> tag) as it is granted (declared in the manifest file using the <*uses-permission*> tag), the checking also terminates and concludes this app cannot be used as a deputy for privilege escalation. (2)Use SOOT to convert the apk to Jimple files, and build the control flow graph (CFG) of the methods in each component based on the Jimple files. (3)By traversing the CFG of each method, WeChecker discovers the intents sent by them and builds up the invocation chain. The definition and construction of invocation chains will be introduced in section 4.1. (4)Identify the entry points of each component, including the lifecycle methods and user-defined event handlers. (5)Construct the call graph of each component and the whole app. (6)Traverse the call graph of the app to detect leak paths.

In the remaining part of this section we will briefly introduce how to identify the entry points of each component and how to construct the invocation chains and call graphs. We lay our emphasis on how to traverse the call graph to find leak paths at a high precision.

4.1 Call Graph and Invocation Chain

Unlike traditional Java programs which use a single main method as the entry point, Android components can have many entry points, including Android lifecycle methods and user-defined event handlers. An entry point is invoked by the Android framework in an event-driven manner based on the state of the app or that of the smartphone. For example, event handlers registered for location change will be invoked when the location of the phone changes. Since there is no main method to organize these callbacks together, discontinuity will occur if we construct the call graph of a component in a traditional way [21, 8, 32, 23, 20] . Besides, since these entry points are invoked in an event-driven manner, we cannot pre-determine their order of execution.

Two issues need to be tackled in order to construct the call graph of a component: identification of entry points and reasonably represent their order of execution. To identify a complete list of entry points, we need to spot the lifecycle methods and user-defined event handlers. We extracted a list of lifecycle methods and align them according to the order defined in the Android documentation [15]. For example, for an Activity, we consider the following four sequences: (1) {onCreate, onStart, onResume, *RunningStateMethods*, onPause, onStop, onDestroy}, (2) {onCreate, onStart, onResume, *RunningStateMethods*, onPause, onStop, onCreate, onStart, onResume, *RunningStateMethods*, onPause, onStop, onDestroy}, (3) {onCreate, onStart, onResume, *RunningStateMethods*, onPause, onResume, *RunningStateMethods*, onPause, onStop, onDestroy}, (4) {onCreate, onStart, onResume, *RunningStateMethods*, onPause, onStop, onRestart, onStart, onResume, *RunningStateMethods*, onPause, onStop, onDestroy}. Here *RunningStateMethods* represent methods that are invoked when the Activity is in the running state, i.e. user-defined event handlers. These sequences are illustrated in the example of figure 4.

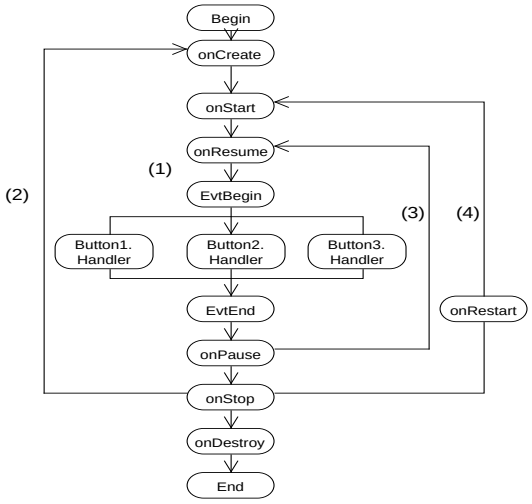

Figure 4: Call Graph of an Activity

There are two ways to define an event-handler: explicitly state them in the layout file or implicitly implement event listener interfaces and override event listener methods. For the first way, we parse the layout file to find the registered callback methods and add them to the call graph of the hosting activity. Several variants exist for the second way. Developers can choose to implement the predefined event-handler interface in a separate class or an inner class. Sometimes the Activity class itself implements the event-handler interface. We extract a list of event-handler interfaces (eg. OnClickListener) and retrieve the list of classes that imple-

ment these interfaces and record their overridden methods. When the analysis encounters a button that registers one of these classes, it will correlate the corresponding overridden method as the event handler of this button.

User-defined event handlers can execute in arbitrary orders during the running state of the hosting activity. This looks like different threads that can execute in parallel. Therefore we represent them in parallel between a pair of $EvtBegin$ and $EvtEnd$ nodes. An example of a call graph is given in figure 4.

Analyzing inter-component communication is prerequisite to find all potential leak paths. On one hand, the source and sink methods of a leak path can reside in different components. On the other hand, without parsing the sender and receiver components of an intent, we cannot get the set of components that can be accessed and exploited by external malicious apps. Consider the type 2.2 leak path in figure 2, a malicious app cannot invoke A_3 directly if A_3 is not exported. By resolving the intents sent by A_1 and A_2, an invocation chain can be constructed from A_1 to A_3. Since this chain starts from an app entry point A_1, all components in this chain become accessible for external applications.

WeChecker performs inter-component analysis and constructs invocation chains for the analyzed app. An *invocation chain* starts from an entry point of the app and is extended to include all components that are reachable from this entry point through inter-component communication. At first WeChecker initializes an invocation chain for each entry point. When it encounters an intent, it will parse the parameters to find the target receiver components according to the intent resolution rules in [16]. The receiver components are added to the corresponding invocation chain.

By constructing invocation chains, we can get a list of reachable components for external apps. WeChecker only raises an alarm when source and sink methods reside in these components to make sure the leak paths are exploitable. Inter-component analysis also enables WeChecker to identify leak paths whose source and sink methods are in different components. Since an invocation chain can be triggered by its initial component, the arbitrary execution of different exported components is converted to the arbitrary execution of different invocation chains.

4.2 Call Graph Traversal

WeChecker traverses the call graph of each component to check whether there exists a path from source methods to sink methods. We adopt the result of SUSI [25] as our source and sink methods. We further divide these methods into data calls and action calls for detection of the two types of leak paths.

During the traversal, WeChecker maintains three sets of tainted variables. One set, L, is for *local variables*, i.e. variables that are only valid within the current method. These include variables declared in the current method and formal parameters. This set is set empty at the entry of each method. The second set, C, is for variables that are valid within the scope of the current class (or component). We call these variables *class variables* in this paper. The taint status of these variables can be different given different execution orders of event handlers. The third set, G, is to store variables that are valid through the app. We denote these variables as *global variables*. The taint status of these variables may be effected by the execution of other invocation chains. Since an external app can invoke any exported component in the analyzed app and the invocation chain starting from this component will be executed subsequently, we assume that each invocation chain can be executed in arbitrary order.

Local variables in Jimple are named and fully typed, therefore the checker can step into the right invoked method based on the actual type of the instance. This is especially important for the analysis of class inheritance when there are overridden methods. Besides, our analysis is flow-sensitive, context-sensitive, object-sensitive and field-sensitive.

- Flow-sensitive: a flow-sensitive analysis takes into account the order of statements in a program[18]. The construction of the control flow graph of each method has considered the order of each statement and the control flow, therefore our analysis is flow-sensitive.

- Field sensitive: To achieve field-sensitivity, we adopt the idea of [29] and [1] and propagate access paths during taint analysis. An access path is of the form $x.f.g$ where x is an object and f and g are fields[1]. In this way different fields of the same object are separated.

- Context-sensitive: that is, procedure calling context is taken into account, and separate information is computed for different calls of the same procedure [18]. Our checker keeps a record of currently tainted variables (using access paths) during the analysis. This record along with the parameters serve as the context information to distinguish different call sites of a method invocation.

- Object-sensitive: that is, different host objects for the same field are treated differently [18]. Besides, different receiver object of a method call should be analyzed separately. Since we use access paths to identify local variables and heap variables, different initiator of the access path indicates different host object of the same field. Our analysis passes the status of the caller object to the callee (identified by *@this* in Jimple) at a method call and passes back the callee status to the caller when the method returns to distinguish different receivers of a method invocation.

To avoid the problem of infinite loop due to recursion, WeChecker restricts the depth of method invocation to be less than 50.

4.3 Alias Analysis

During the analysis, WeChecker computes two types of sets for each method: (1) Type1: the set of heap variables that refer to the same memory location, i.e, alias set. (2) Type2: the set of parameters, class variables, global variables and return variables that have the same value even if they are not heap variables. These sets are used to decide the taint status of certain variables in a method summary.

Let $\mathbf{A}$ denote the current type1 and type2 sets, i.e. $\mathbf{A} = \{A_1, A_2, \cdots, A_n\}$ where A_i is a type1 or type2 set. At the entry of the method $\mathbf{A}$ is $\emptyset$. WeChecker performs context-sensitive alias analysis[29, 28] to generate the type1 sets. Type2 sets are generated from assignment statements. We define the function $p(x)$ to return the set that contains x, i.e, $p(x) = A_i$ if $x \in A_i$. A new set is added to $\mathbf{A}$ when there

are assignments in the form $x.f = y.g$ and $p(y.g)$ returns $NULL$. The semantic rules to update **A** are as follow:

Case 1: When the analysis encounters the statement $x = new \cdots$, $x.f^n$ should be removed from the current set since x will point to a different memory location. The effect of this statement is:

$$\forall A_i \in p(x.f^n),\ A_i = A_i \backslash x.f^n,$$

Case 2: When the analysis encounters the assignment $x.f = y.g$, $x.f.f^n$ will be removed from the current set and added to the same set with $y.g.f^n$. The effect of this statement is:

$$\forall A_i \in p(x.f.f^n),\ A_i = A_i \backslash x.f.f^n$$

$$\forall A'_i \in p(y.g.f^n),\ A'_i = A'_i \cup \{x.f.f^n\}$$

Case 3: When the analysis encounters the method invocation $b = c.m(a_0, \cdots, a_n)$, four types of information needs to be taken into account when the method returns: (1) The receiver object c, i.e., if $this.f^n$ belongs to a set in the callee's context, $c.f^n$ should be added to the corresponding set in the caller's context. (2) Parameters of the method call, i.e., if the formal parameter $param_i.f^m$ is included in a set in the callee's context, the actual parameter $a_i.f^m$ should also be added to the corresponding set in the caller's context. (3) Class variables and global variables. If the callee adds a class or global variable $x.f^q$ to a set, it should be added to the correlated set in the caller's context. (4) Return value of the method call, i.e, the corresponding return variable should be added to the caller's set if it is enclosed in the callee's set.

$$\begin{aligned} A_{i\_caller} &= \{c.f^n : this.f^n \in A_{i\_callee}\} \\ &\cup \{a_i.f^m : param_i.f^m \in A_{i\_callee}\} \\ &\cup \{x.f^q : x.f^q \in A_{i\_callee} \wedge x.f^q \ not\ local\} \\ &\cup \{b.f^v : r.f^v \in A_{i\_callee}\} \end{aligned}$$

After the update on each statement, sets that contain only one element will be removed from **A**. Transitive closure of the existing sets is computed to derive a more complete set.

Let's take the *taintIt* method in figure 1 as an example. At the entry of the method, $\mathbf{A} = \emptyset$. Two sets $\{x, par_2\}$ and $\{x.f, par_1\}$ are added to **A** after the analysis of lines 37-38. Line 39 adds $sData$ to the existing set $\{x, par2\}$ and gives $\{x, par_2, sData\}$. The return statement on line 41 gives $\{r, par_1\}$ (r represents the return value). Transitive closure of existing sets gives $\{x.f, par_2.f, sData.f, par_1, r\}$. At the exit of this method,

$$\mathbf{A} = \{\{par_2, sData\},\ \{par_2.f, sData.f, par_1, r\}\}$$

Note that local variables such as x are eliminated since they are invalid outside the method. Mapping **A** back to the invocation of $taintIt$ on line 16 gives $\{d, sData\}$ and $\{d.f, sData.f, temp_1, rs\}$, where $temp_1 = source()$ is tainted. Therefore *sink2* on line 21 would raise an alarm for a leak. However, the statement on line 20 removes d and $d.f$ from the set, hence *sink4* on line 23 would not give any warning. In comparison, method invocation on line 31 would render sets $\{d1, sData\}$ and $\{d1.f, sData.f, temp_2\}$ where $temp_2$ = "*notaint*" which is not tainted. Therefore no leak occurs on line 32.

4.4 Method Summary and Class Summary

For each method, the *method summary* includes the following information: (1) The type1 and type2 sets computed according to the rules in section 4.3. Taint status of the variables in these sets is also recorded if can be determined. (2) *Conditional sinks*. We call a sink a conditional sink if the taint status of its parameters cannot be determined, i.e, its parameters contain parameters of the hosting method or class/global variables of the hosting class or their alias. If the parameter of the sink method is a local variable, we replace the local variable by its alias in the type1 or type2 set. (3) Paths that would definitely lead to a leak. This information is used to give warnings of leak paths. (4) The sets of tainted class variables and global variables (C and G).

In the example of figure1, the summary of *taintIt* includes the following information: (1) Two types of sets: $\{par_2, sData\}$, $\{par_1, par_2.f, sData.f, r\}$. Here r represents the return value. Since we cannot determine their taint status, no information about their taint status will be recored at this moment. (2) Conditional sink: $sink5(par_2.f)$. (3) Definite leak paths. Since there is no path that will definitely lead to a leak, no such information is recorded either. (4) Tainted variable lists: since no class or global variables are tainted, C and G are $\emptyset$.

WeChecker extracts the method list of each component and computes their method summaries. Our analysis is path-sensitive when it calculates the summary of each method. In other words, WeChecker considers every possibility of data flow. For example, if there are three if-else branches in a method, WeChecker will traverse all 8 (i.e. 2^3) possible paths. After extracting the method summaries, the traversal of the call graph converts to the traversal of each method summary according to their order in the call graph.

The traversal starts from the method summary of the first method in the call graph and invokes the next method it points to. During the traversal, at the entry of each method, the checker passes the set of current tainted variables as context to find answers to these questions: (1) Will the conditional sinks in the method summary actually lead to a leak with the given context? (2) What is the taint status of the class/global variables after the execution of this method? In the case of a method invocation, it also needs to answer the third question: (3) What is the taint status of the return value? When the method returns, the checker will determine which variable would get tainted after mapping the tainted variables in the callee back to the caller. For the example in figure 1, the invocation of $taintIt$ on line 17 passes a tainted value as the first parameter. From the method summary of *taintIt* we know that $par_2.f$ is also tainted, therefore the conditional sink $sink5(par_2.f)$ would lead to a leak. When $taintIt$ returns, $d.f$, $sData.f$ and rs would get tainted since par_1 is tainted. Therefore *sink2* will lead to a leak.

At the entry of the first method, the three sets of tainted variables (L, C and G) are all initialised to $\emptyset$. The context passed to the next method is the context at the return statement of the previous method. Methods between the *EventBegin* and *EventEnd* nodes are treated exceptionally. We will introduce how to deal with these methods in the next section.

The summary of a class is constructed after the analysis of this class. A *class summary* contains the following information: (1) Tainted global variable list (G). (2) Conditional sinks whose parameters are public class variables or alias of public class variables. The class summary can be extracted

from method summaries.

4.5 Strategy for Arbitrary Execution Orders

Within an activity, user-defined event handlers may execute in arbitrary orders depending on user input or system status. To get an accurate approximation of the tainted variables, the analysis need to consider the interference of other event-handlers when analyzing one event handler. In order to simulate the arbitrary execution orders of different event handler methods, a naive way is to get the list of all permutations of these methods and execute each permutation in this list. Suppose there are n event handlers, the number of permutation would be $n!$. This would lead to a high analysis overhead when n is big. We propose to check each event handler two times in two rounds to achieve a precise yet efficient analysis.

Note that the context passed to each method summary in the second round is different from that in the first round. In the first round, the context passed to each event handler is the context at the *EvtBegin* node. In the second round, the analysis takes into account the interference of other event handlers that can execute before the current method. As introduced in section 4.2, during the traversal of the call graph, WeChecker maintains three sets of tainted variables: L for local variables, C for class variables and G for global variables. The execution of other event handlers only effect C and G. Suppose there are n event handlers and they can execute in arbitrary orders. L_{i_1}, C_{i_1} and G_{i_1} denote the context after the analysis of event handler i in the first round. In the second round, the context passed to an even handler is the merge of the other $(n-1)$ events' context. The rule to merge the context of two events, e_i and e_j is defined in algorithm 1. Recall that we defined $p(x)$ to return the type1 or type2 set that contains x. For clarity, we further define $p_i(x)$ to return the set that contain x in the method summary of e_i. We denote the type1 and type2 sets of event i as $\mathbf{A_i}$.

Algorithm 1 Merge the Context of e_i and e_j

Require: $C_{i_1}, G_{i_1}, C_{j_1}, G_{j_1}, \mathbf{A_i}, \mathbf{A_j}$
Ensure: C_{m_1}, G_{m_1}
1: **function** ContextMerge($C_{i_1}, G_{i_1}, C_{j_1}, G_{j_1}, \mathbf{A_i}, \mathbf{A_j}$)
2: **for** $var:\ var \in C_{i_1} \cup G_{i_1}$ **do**
3: $A_j = p_j(var)$
4: $C_{j_1} \leftarrow C_{j_1} \cup c$: $c \in A_j \wedge c$ is classVar
5: $G_{j_1} \leftarrow G_{j_1} \cup g$: $g \in A_j \wedge g$ is globalVar
6: **end for**
7:
8: **for** $var:\ var \in C_{j_1} \cup G_{j_1}$ **do**
9: $A_i = p_i(var)$
10: $C_{i_1} \leftarrow C_{i_1} \cup c$: $c \in A_i \wedge c$ is classVar
11: $G_{i_1} \leftarrow G_{i_1} \cup g$: $g \in A_i \wedge g$ is globalVar
12: **end for**
13:
14: $C_{m_1} \leftarrow C_{i_1} \cup C_{j_1}$
15: $G_{m_1} \leftarrow G_{i_1} \cup G_{j_1}$
16: **end function**

Algorithm 1 takes as input the tainted variable sets and alias sets of e_i and e_j and outputs the merged context of these two events. Lines 2-6 traverse the tainted variable sets of event i and check if any variables in event j would get tainted if event i happens before event j. If so, add these variables to the corresponding tainted variable set of event j, i.e. add the variable to C_{j_1} if it is a class variable and add it to G_{j_1} if it is a global variable. Lines 8-12 consider the situation that event j happens before event i and check whether C_{j_1} and G_{j_1} would effect the taint status of the variables in event i. The merged tainted variable list is the union of the tainted variable sets of the two events. The process continues until the context of all other $n-1$ events except the current analyzed event handler has been merged together.

In the second round, we denote the context information at the entry of event i as $\{L_{i_2}, C_{i_2}, G_{i_2}\}$. L_{i_2} is initialized as $\emptyset$, C_{i_2} and G_{i_2} are initialized as the final C_{m_1} and G_{m_1} after merging all other $n-1$ event handler's context respectively. The time complexity of the merging process is $O(n)$. Since we need to analyze the n event handlers one by one, therefore the total time complexity is $O(n^2)$.

For the example in figure1, our checker identifies the event handlers for the three buttons and give them temporary method names *eventH1*, *evntH2*, *eventH3*. Their method summaries are listed here:
eventH1: {conditionalSink: $sink1(sData.f); C_{1_1} = \emptyset; G_{1_1} = \emptyset$}
eventH2: {leakPath: line21 $sink2(rs), rs$ tainted on line16 $C_{2_1} = sData.f, G_{2_1} = \emptyset$}
eventH3: $\{C_{3_1} = \emptyset, G_{3_1} = \emptyset\}$

When we analyze *eventH1* in the second round, the context at the entry is: $L_{1_2} = \emptyset, C_{1_2} = \{sData.f\}, G_{1_2} = \emptyset$, here C_{1_2} and G_{1_2} are merging results of the context of $eventH2$ and $eventH3$. Therefore the conditional sink $sink1$ would lead to a leak.

The same technique is used to deal with the arbitrary execution orders of different invocation chains in one app. Each invocation chain is checked two times in two rounds. In the first round, the analysis goes into each component and traverses each method summary according to its call graph. At the end of the first round, the class summary of each class is dirived. In the second round, the effect of other invocation chains are taken into account to figure out whether the conditional sinks in the class summaries will actually lead to a leak. Take the test case ActiviyComm1[27] in DroidBench as an example. The variable $data1$ is valid through the app. Since both *Activity*1 and *Activity*2 are exported, two invocation chains {{Activity1}, {Activity2}} exist for this app. $data1$ gets tainted in Activity2, therefore in the second round the conditional sink in Activity1 (line 26 of [27]):

$$sms.sendTextMessage(" + 49", null, data1, null, null);$$

will lead to a leak.

4.6 Array and Collection Access

Previous work based on static analysis [1, 13] usually conservatively mark the whole array or collection as tainted when one element gets tainted. This results in false positives when tainted data is stored in one position but data access occurs in another position. Array is one of the prime data structures in Java to store a set of values. Other classes for this purpose include all types of List, Set and Map implementations (such as LinkedList, ArrayList, HashMap, etc.). Arrays use indexes to identify different elements. Maps store <key, value> pairs and use keys to identify different values. In our checker we partially reduce the false positives resulted

from array and collection access by resolving and differentiating elements in arrays and maps.

```
1  $r4 = newarray (java.lang.String)[10];
2  ...
3  $r3 = virtualinvoke $r6.<android.telephony.
       TelephonyManager: java.lang.String
       getDeviceId()>();
4  $r4[5] = $r3;
5  $r4[4] = "no taint";
6  $r2 = staticinvoke <android.telephony.SmsManager:
        android.telephony.SmsManager getDefault()
       >();
7  $i0 = specialinvoke $r0.<de.ecspride.ArrayAccess2
       : int calculateIndex()>();
8  $r7 = $r4[$i0];
9  virtualinvoke $r2.<android.telephony.SmsManager:
       void sendTextMessage(java.lang.String,java.
       lang.String,java.lang.String,android.app.
       PendingIntent,android.app.PendingIntent)>("
       +49 1234", null, $r7, null, null);
```

Figure 5: Jimple code of ArrayAccess2

Like the representation of instance fields, we use access path to represent array and map elements. The access path of an array element initiates from the array variable and is followed by the index of this element. Take the Jimple code in figure 5 as an example (excerpt from the DroidBench test case ArrayAccess2), $r4[5] is represented using the access path $r4.5. If the index cannot be computed statically, the checker will list all the possibilities and raise an alarm unless it can figure out all these possible values are not tainted. For the test case ArrayAccess2, method calculateIndex() returns the constant value 4, therefore $r7 equals to $r4.4 which is not tainted.

```
1  $r4 = new java.util.HashMap;
2  specialinvoke $r4.<java.util.HashMap: void <init
       >()>();
3  ...
4  $r6 = virtualinvoke $r7.<android.telephony.
       TelephonyManager: java.lang.String
       getDeviceId()>();
5  interfaceinvoke $r4.<java.util.Map: java.lang.
       Object put(java.lang.Object,java.lang.Object
       )>("tainted", $r6);
6  interfaceinvoke $r4.<java.util.Map: java.lang.
       Object put(java.lang.Object,java.lang.Object
       )>("untainted", "Hello World");
7  $r2 = staticinvoke <android.telephony.SmsManager:
        android.telephony.SmsManager getDefault()
       >();
8  $r5 = interfaceinvoke $r4.<java.util.Map: java.
       lang.Object get(java.lang.Object)>("
       untainted");
9  $r3 = (java.lang.String) $r5;
10 virtualinvoke $r2.<android.telephony.SmsManager:
       void sendTextMessage(java.lang.String,java.
       lang.String,java.lang.String,android.app.
       PendingIntent,android.app.PendingIntent)>("
       +49 1234", null, $r3, null, null);
```

Figure 6: Jimple code of HashMapAccess1

The access path of a map element initiates from the map variable and is followed by the key of this element. The key is determined when this element is put into the map. In the example of figure 6 (excerpt from the DroidBench test case HashMapAccess1), $r4.tainted=$r6, $r4.untainted="Hello World". Therefore $r5 equals to $r4.untainted which is not tainted.

⊛=correct warning, ⋆ = false warning, ○ =missed warning
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	FlowDroid	WeChecker
Arrays and Lists		
ArrayAccess1	⋆	
ArrayAccess2	⋆	
ListAccess1	⋆	⋆
Callbacks		
AnonymousClass1	⊛	⊛
Button1	⊛	⊛
Button2	⊛ ⊛ ⊛ ⋆	⊛ ⊛ ⊛
LocationLeak1	⊛ ⊛	⊛ ⊛
LocationLeak2	⊛ ⊛	⊛ ⊛
MethodOverride1	⊛	⊛
Field and Object Sensitivity		
FieldSensitivity1		
FieldSensitivity2		
FieldSensitivity3	⊛	⊛
FieldSensitivity4		
InheritedObjects1	⊛	⊛
ObjectSensitivity1		
ObjectSensitivity2		
Inter-App Communication		
IntentSink1	○	⊛
IntentSink2	⊛	⊛
ActivityComm1	⊛	⊛
Lifecycle		
BroadcastRecvLifecycle1	⊛	⊛
ActivityLifecycle1	⊛	⊛
ActivityLifecycle1	⊛	⊛
ActivityLifecycle1	⊛	⊛
ActivityLifecycle1	⊛	⊛
ServiceLifecycle1	⊛	○
General Java		
Loop1	⊛	⊛
Loop2	⊛	⊛
SourceCodeSpecific1	⊛	⊛
StaticInitialization1	○	⊛
UnreachableCode1		
Miscellaneous Anrdriod-Specific		
PrivateDataLeak1	⊛	⊛
PrivateDataLeak2	⊛	⊛
DirectLeak1	⊛	⊛
InactiveActivity		
LogNoLeak		
Sum, Precision, and Recall		
⊛, higher is better	26	27
⋆, lower is better	4	1
○, lower is better	2	1
Precision $p = ⊛/(⊛ + ⋆)$	86%	96%
Recall $r = ⊛/(⊛ + ○)$	93%	96%
F-measure $2pr/(p + r)$	0.89	0.96

Table 1: DroidBench Test Results

It is not easy to unify the analysis of Lists using access paths since elements are added or removed without referencing to the index but are accessed by referring to the index. The precise resolution of List access is left for future work.

5. IMPLEMENTATION AND EVALUATION

5.1 Implementation

The entire system of WeChecker consists of around 9,000 lines of JAVA code. The system runs under Linux Ubuntu 13.04 on a computer with Intel Core I5 3.4GHz CPU and 4GB RAM.

5.2 Evaluation

5.2.1 Evaluation on DroidBench

DroidBench [26] is an open test suite for evaluating the effectiveness of taint analysis tools especially for Android apps. It contains test cases for general static analysis problems (field sensitivity, object sensitivity, etc.) and Android-specific challenges like correctly modeling an app's lifecycle, adequately handling asynchronous callbacks and interacting with the UI. For easy compairson with previous work[1], we evaluated our system on DroidBench V1.1 which contains 39 hand-crafted Android apps. Table 1 gives the analysis results of FlowDroid [1] and WeChecker when applied to DroidBench.

As table 1 shows, WeChecker achieves 96% recall and 96% precision. Thanks to the precise resolution of array access, WeChecker did not raise any false alarms for the test case ArrayAccess1 and ArrayAccess2. However, as mentioned in section 4.6, we conservatively take the whole LinkedList as tainted if one element gets tainted. This leads to the false positive for the case ListAccess1. In comparison, FlowDroid gives false warning for all the three test cases. ServiceLifecycle1 is not detected because there is no exploitable path in it. It only contains one service component which is not exported, therefore malicious applications cannot get access to this component. The result shows that WeChecker effectively tackles challenges brought by Android callbacks, inter-app communication, field and object sensitivity, etc.

5.2.2 *Evaluation on downloaded apps*

Apart from the evaluation on DroidBench, we also applied WeChecker to 1137 Android apps randomly downloaded from Google Play. WeChecker finished checking all the applications in less than 9.5hrs. The average analysis time of each app is less than 30 seconds (29.985s). For compairson, Bati's [2] average analysis time per app is about 26min. FlowDroid [1] finished checking most of its examined apps in under a minute. However FlowDroid only focuses on a single component and did not take inter-component analysis into account. CHEX [10] makes permutations to emulate the arbitrary execution orders of different splits. They define a split as a subset of app code that is reachable from a particular entry point method. CHEX limited the processing time of each app within 5 minutes to optimize the throughput. Their results on real applications show that 22% apps needed more than 5 minutes to be analyzed which leads to timeout. Their statistic of the median processing time for an app (37.02s) has excluded these apps. The median processing time of WeChecker is 21s even after taking the longest processing time (around 9mins) into account. [10] further found that split permutation causes the majority of the time overhead. WeChecker uses summary based two round taint analysis to reduce this part of overhead. Therefore it achieves a higher efficiency than other checkers.

WeChecker raised alarms for 79 leak paths among which 2 are capability leak paths and 77 are sensitive data leak paths. Among the sensitive data leak paths, 2 of them leak sensitive data to the caller app (type 2.1), 21 attach the data to implicit intents (type 2.2) and 56 of them write data to log files or external storage (type 2.3). FlowDroid over-approximates explicit inter-component communication by taking methods which send intents as sinks and callbacks which receive intents as sources [1]. Although WeChecker uses the same source and sink set as FlowDroid, it aims at the detection of privilege escalation vulnerabilities. Therefore either the source or sink method of a leak path requires permission. Hence WeChecker will not regard a path that sends out the received data from another app as a leak path since neither putExtra nor getExtra method requires permission.

6. RELATED WORK

Davi et al.[7] first proposed the privilege escalation attack in Android applications and demonstrated how to exploit a vulnerability at runtime. After that, there have been many approaches proposed to deal with this attack.

Some researchers proposed dynamic solutions to remedy the attack at runtime. TaintDroid[9] uses dynamic taint tracking to trace the flow of privacy sensitive data in third-party apps. It helps to detect when sensitive data leaves the system via untrusted apps. But evaluation shows that it is not efficient to use practically. [11] proposed IPC Inspector to mitigate permission re-delegation attacks by checking IPC call chains. IPC Inspector ensures that unauthorized applications cannot invoke privileged operations by reducing the permission set of an app after it receives communication from a less privileged application. However their mechanism is restrictive because of its limited usage scenario and large overhead. [3] uses a run-time monitor to regulate communications between applications. They designed and implemented a practical security framework to protect against confused deputy and collusion attack.

In contrast to dynamic analysis which brings in large performance overhead at runtime, static analysis checks the vulnerabilities in Android apps before installation. SCanDroid[12] is the first static analysis tool for Android. It checker whether inter-component(ICC) and inter-app(IPC) data flows violate the specification extracted from the manifest file. DroidChecker[4] and CoChecker[6] use static taint analysis to detect whether data from source methods can reach sink methods. But they both work on the source code. This makes their precision dependent of that of the decompiler tools. Besides, they fail to perform alias analysis, which further detriments their precision.

[17] proposed Woodpecker to deal with the problem of capability leak which essentially reflects the privilege escalation attack. They handled two categories of capability leaks: explicit and implicit. But they only considered 13 permissions that are defined by the framework itself without paying attention to user-defined permissions in third-party applications. Besides, they conclude a capability leak exists as long as there is a path that enables data to flow from one entry point to a permission-protected API without any security checkings in-between. This is true when this API has side-effect on the system settings. However, if this API aims to retrieve sensitive data and there is no way for the malicious app to get the retrieved data, the attacker cannot get any benefit.

AndroidLeaks [13] verifies whether sensitive data will be propagated and finally sent out of the phone. It computes forward slices from tainted data, and analyze the slice to check if any parameters to sink methods are tainted. But it performs pointer analysis in a context-insensitive way. Besides, it taints the entire collection and the whole object if any tainted data is stored in them. This leads to over-tainting and results in false positives.

CHEX [10] relies on static data-flow analysis to deal with component hijacking vulnerabilities. They designed a reliable way to discover all types of entry points in an Android component and modelled the asynchronous invocation of every entry point by splitting the app code. In this way, CHEX tackled the problem brought by the event-driven property. However, their solution requires a permutation of all splits which is time and resource consuming. FlowDroid[1] provides a novel and highly precise taint analysis for Android applications. It modelled the Android lifecycle and carried out context, flow, field and object-sensitive analysis to achieve a high degree of precision. However, it fails to perform inter-component analysis.

AppIntent [31] aims to detect whether sensitive data transmission is user intended or not. It uses a guided symbolic

execution approach to generate a sequence of GUI manipulations that would lead to the identified sensitive data transmission. Bati [2] claims to provide a vetting framework for Android apps. Their analysis considers a complete Android lifecycle that includes the asynchronous communication of multi-threading. They also introduce a value analysis algorithm to determine primitive values and string parameters. Despite of its claimed high precision, Bati's average analysis time per app is about 26min which is too long for large-scale app checking.

[5] and [22] studied potential vulnerabilities brought by the intent mechanism in Android apps. ComDroid focuses on the analysis of one single application and gives warnings when it encounters misconfigurations such as exported components or implicit intents. However, ComDroid produces too many false positives. [22] aims to connect components within one application and between different applications. Therefore it works on a large scale of applications and matches the exit/entry points of the currently analyzed application with a set of entry/exit points stored in the database. In our analysis, we only need to get the connection between components within the analyzed application to construct the invoking chain. For intents to communicate with components in another application, we only check whether sensitive data is attached to indicate potential data leakage.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we aim at checking whether an Android application is vulnerable to privilege escalation attacks by detecting two types of leak paths: capability leak paths and sensitive data leak paths. We use a summary (*method summary* and *class summary*) based two round static taint analysis to reduce the analysis overhead brought by the arbitrary execution orders of different event handlers and different invocation chains. During the analysis we use access paths to represent object fields, array elements and map elements. This reduces the false positives due to field-insensitivity and over-approximation of array access and collection access. We designed and implemented a tool, WeChecker to automatically detect whether privilege escalation vulnerabilities exist in Android apps. We demonstrated the high precision of WeChecker on the state-of-the-art test suite DroidBench. The evaluation of WeChecker on real apps shows that it is efficient and fits for large-scale analysis.

For future research, we aim to improve the analysis precision by taking implicit flow and reflection calls into consideration. Besides, we aim to resolve the inter-app communication at a higher precision to reduce the false alarms due to mis-resolution of the receiver components of an intent.

8. ACKNOWLEDGEMENT

The work described in this paper was partially supported by a collaboration research fund by Huawei, HKU Seed Fundings for Applied Research 201409160030, and HKU Seed Fundings for Basic Research 201311159149 and 201411159122. A patent about this technology is filed.

9. REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[2] M. Backes, S. Bugiel, E. Derr, and C. Hammer. Taking android app vetting to the next level with path-sensitive value analysis. *Report (Bericht)*, 2014.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[4] P. P. Chan, L. C. Hui, and S.-M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[6] X. Cui, D. Yu, P. Chan, L. C. Hui, S. Yiu, and S. Qing. Cochecker: Detecting capability and sensitive data leaks from component chains in android. In *Information Security and Privacy*, pages 446–453. Springer, 2014.

[7] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] S. K. Debray and T. A. Proebsting. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Trans. Program. Lang. Syst.*, 19(4):568–585, July 1997.

[9] 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12,NY, USA*, pages 229–240. ACM, 2012.

[11] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[12] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[13] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In S. Katzenbeisser, E. Weippl, L. Camp,

M. Volkamer, M. Reiter, and X. Zhang, editors, *Trust and Trustworthy Computing*, volume 7344 of *Lecture Notes in Computer Science*, pages 291–307. Springer Berlin Heidelberg, 2012.

[14] Google. Android. `http://source.android.com/`.

[15] Google. Android Developers. `http://developer.android.com/index.html`.

[16] Google. Intent. `http://developer.android.com/reference/android/content/Intent.html`.

[17] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[18] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[19] IDC. Smartphone OS Market Share, Q3 2014. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

[20] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 89–103. IEEE, 1999.

[21] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 287–298, New York, NY, USA, 2009. ACM.

[22] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[23] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP 2005-Object-Oriented Programming*, pages 362–386. Springer, 2005.

[24] G. Play. Ulysses Gizmons. `https://play.google.com/store/apps/details?id=com.binarytoys.ulysse`.

[25] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[26] E. SPRIDE. DroidBench. `https://github.com/secure-software-engineering/DroidBench`.

[27] E. SPRIDE. DroidBench test case for inter-component communication. `https://github.com/secure-software-engineering/DroidBench/tree/master/eclipse-project/InterComponentCommunication/ActivityCommunication1`.

[28] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

[29] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.

[30] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[31] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[32] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM.