CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities

Long Lu†   Zhichun Li‡   Zhenyu Wu‡   Wenke Lee†   Guofei Jiang‡

†College of Computing, Georgia Institute of Technology   ‡NEC Labs America, Inc.
{long, wenke}@cc.gatech.edu   {zhichun, adamwu, gfj}@nec-labs.com

ABSTRACT

An enormous number of apps have been developed for Android in recent years, making it one of the most popular mobile operating systems. However, the quality of the booming apps can be a concern [4]. Poorly engineered apps may contain security vulnerabilities that can severally undermine users' security and privacy. In this paper, we study a general category of vulnerabilities found in Android apps, namely the component hijacking vulnerabilities. Several types of previously reported app vulnerabilities, such as permission leakage, unauthorized data access, intent spoofing, and *etc.*, belong to this category.

We propose CHEX, a static analysis method to automatically vet Android apps for component hijacking vulnerabilities. Modeling these vulnerabilities from a data-flow analysis perspective, CHEX analyzes Android apps and detects possible *hijack-enabling flows* by conducting low-overhead reachability tests on customized system dependence graphs. To tackle analysis challenges imposed by Android's special programming paradigm, we employ a novel technique to discover component entry points in their completeness and introduce *app splitting* to model the asynchronous executions of multiple entry points in an app.

We prototyped CHEX based on Dalysis, a generic static analysis framework that we built to support many types of analysis on Android app bytecode. We evaluated CHEX with 5,486 real Android apps and found 254 potential component hijacking vulnerabilities. The median execution time of CHEX on an app is 37.02 seconds, which is fast enough to be used in very high volume app vetting and testing scenarios.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*

General Terms

Security

Keywords

Static analysis, app splitting, component hijacking vulnerability

1. INTRODUCTION

Android has gained tremendous popularity in recently years, with over 100 million activations globally [22]. Part of the success should be attributed to Android's easy-to-join application development community. More than 400k apps are available in the official Android Market, yielding 10 billion accumulative installations by the end of 2011 [3]. Alternative markets also play a big role in hosting and distributing a large number of apps. Most apps were developed and released in the last three years.

As large numbers of new apps, including newly updated versions, are constantly submitted to app markets and become instantly available for users, we believe it is critical to provide a scalable vulnerability filtering system for app market operators. Before apps are released, the system quickly vets the apps for potential security vulnerabilities and provides warning messages to help the developers generate fixes. Apparently, such systems should scale well in face of a high input volume and have reasonably low false positive rates to be useful. Therefore, we advocate a static analysis based approach, as opposed to dynamic ones, for its complete code coverage and scalability.

In this paper, we propose CHEX[1], a static app vetting tool for component hijacking vulnerabilities. Such vulnerabilities are found in apps that implement access control improperly on external requests or accidentally leak private data or privileges. In general, these vulnerabilities are exploited to carry out unauthorized read or write operations on sensitive resources. Therefore, we transform the detection problem into an equivalent data-flow problem that seeks to identify the existence of hijack-enabling flows in apps. Component hijacking vulnerabilities include, but are not limited to, several previously reported vulnerabilities, such as permission re-delegation and leakage [20, 21], intent spoofing [10], and private data leakage (*e.g.* login credentials).

Our work makes three folds of contributions. First, we designed a sound method that automatically discovers all types of app entry points at a low false rate, whereas previous works primarily use simple domain knowledge and only finds the common entry point types. Second, in order to efficiently model interleaved executions of multiple entry points and track data-flows crossing them, we propose the concept of *app splitting*, which generates and permutes data-flow summaries of each *split*[2]. The existence of hijack-

---

[1]CHEX stands for Component Hijacking Examiner.
[2]A term we defined to describe a fragment of code reachable from a single entry point
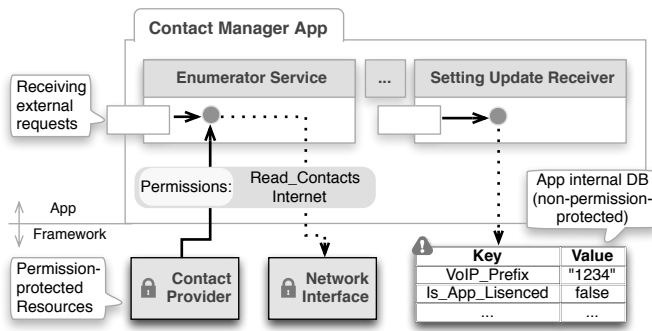
Figure 1: *An app vulnerable to component hijacking*

enabling data-flows is checked by means of a reachability analysis on customized system dependence graphs [25] that capture the variable dependencies globally. Finally, we built CHEX, an in-depth and scalable static app vetting tool for component hijacking vulnerabilities. We exercised CHEX with 5,486 popular free Android apps collected from both the Official Android Market and alternative Android markets. The median execution time of CHEX on an app is 37.02 seconds, which is fast enough to be used in very high volume app vetting and testing scenarios. Among all tested apps, 254 were found to have hijack-enabling flows. Our manual validation on all these suspicious apps shows a true positive rate of 81%. We also conducted detailed case studies on the vulnerable apps we uncovered, providing practical insight into the vulnerability and possible exploits. CHEX is based on Dalysis, our static analysis framework that directly consumes off-the-shelf Android apps in bytecode form and supports various types of program analysis tasks.

The rest of the paper is organized as follows. We study the component hijacking problem and present our analysis method in Section 2 and 3, followed by the implementation and evaluation of CHEX and Dalysis in Section 4 and 5. In Section 6, we discuss limitations of our work and possible workarounds. We survey related work in Section 7 and conclude the paper in Section 8.

2. COMPONENT HIJACKING PROBLEM

Android framework dictates a component-based approach to app design, for flexible interoperability among apps and efficient app lifecycle management. In this approach, app developers organize their code into individual *application components* [6] (*i.e.* `Activities`, `Services`, and etc.). Each component fulfills a logically independent task and can serve requests from other components in the same app, the framework, or another app if the component is publicly available (or is *exported*, in Android terminology). For example, an instant messaging app may need a contact enumerator (*i.e.* collecting all contacts on the device) to suggest friends for the user. Instead of implementing its own, the app can leverage an existing contact enumerator component exported by a contact manager app.

However, the capability of reusing a component under its containing app's identity can lead to serious security threats, when the component is security-critical but not well protected. To generalize threats of this kind, we introduce the concept of *component hijacking*, describing a class of attacks that seek to gain unauthorized access to protected or private resources through exported components in vulnerable apps. As shown in Figure 1, if the contact manager app fails to deny requests from unauthorized apps, a malicious

app can easily take advantage of (or hijack) the `Enumerator Service` and consequently gain access to user's contacts without the required permission. Recent works [11, 20, 21, 27] reported attacks similar to this particular example, all of which derive from the classic confused deputy attack [23] and aim at escalating privileges of attacking apps in the context of Android's permission system. Note that, although permission-protected resources (*e.g.* contacts, geo-location, and *etc.*) are obvious targets, component hijacking is by no means limited to these confused deputy attacks that bypass the permission checks. In fact, if carelessly exposed, data or invokable interfaces that are only intended for app's internal use (thus not permission-protected) can also become targets of component hijacking attacks. In this case, where no explicit permission is involved, the attacking app seeks to tamper or steal private data of a vulnerable app that does not enforce access control or input validation properly. For instance, in Figure 1, the security-critical information stored in the app internal database can be tampered through the `Setting Update Receiver` in an SQL-injection fashion. Complicated cases exist, where an attacker can leverage a chain of vulnerable components to steal private data, modify critical settings, or perform privileged actions, by simply issuing crafted requests as a regular app.

Several topics related to component hijacking were studied by recent works. ComDroid [10] checks app metadata and API usages for publicly exported components. Such components, if granted direct or indirect access to sensitive resources, may become launching points for hijacks. Grace *et al.* [21] analyzed factory stock apps to identify permission leakage, a threat that also spurred studies on its runtime mitigations [8, 12, 20]. While these works are effective in archiving their own goals, they target at the vulnerabilities that only represent a subset of component hijacking (*i.e.* hijacks seeking to access non-permission-protected sensitive resources are not covered). Plus, these works do not intend to provide any in-depth detection method suited for scalable app vetting. Our work aims to bridge this gap.

It is noteworthy that component hijacking vulnerability is not caused by any insecurity intrinsic to Android framework. In fact, Android does provide a set of mechanisms to secure app components and their interactions. Instead, similar to other security vulnerabilities in software, component hijacking stems from issues that are hard to avoid in reality, such as undertrained developers, lacks of proper app quality assurance, and usability issues of existing security mechanisms. We expect component hijacking vulnerability to emerge rapidly in terms of popularity and severity. As the user population of Android constantly grows, more and more developers are migrating to this platform, often with inadequate experience or knowledge on its security mechanisms. In addition, the current app distribution model offers a convenient way for amateur developers to release their apps to a wide range of users. With these factors adding up, the odds becomes high for a regular Android user to install apps that insecurely handle external requests and thus are subject to component hijacking. Attackers who are now struggling with crafting new exploiting techniques on Android would not easily let this new attacking vector pass by.

Apps with component hijacking vulnerabilities are generally not malicious on their own, but can be coerced by attacking apps to conduct malicious activities. Defensive efforts may focus on either finding the vulnerabilities in benign apps, or detecting corresponding exploits from suspicious apps. Our work follows the first approach for the more distinguishable and less volatile nature of the

subject being detected, than that of the second one. Without loss of generality, we define component hijacking attacks as follows:

DEFINITION 1. *An unauthorized app, issuing requests to one or more public components in a vulnerable app, seeks to:*

    G1 : READ *sensitive data out of the app; or*
    G2 : WRITE *to critical data region inside the app; or*
    G3 : perform a combination of G1 and G2.

Based on this definition, to determine if a given component (or set) is vulnerable to hijacking is equivalent to finding feasible data flows that can enable any of the three goals above without going through any security checkpoint. We refer to these flows as *hijack-enabling flows* hereafter. In a simple example, the `Emulator Service` in Figure 1 is vulnerable if a hijack-enabling flow exists that fulfills *G1*: the flow propagates the contact list into an object to be returned to the requestor, serving as a data sink from which the requestor (or attacking app) can <u>read</u> data directly. In a more complex scenario, the data sink may not seem immediately accessible to the requester (*e.g.* sending contact to an URL, as shown in Figure 1). However, if the component contains a hijack-enabling flow that <u>writes</u> requester-supplied input into certain output-controlling data (*e.g.* the destination URL), requestors can still indirectly <u>read</u> the contact information by redirecting the output and achieve *G3*. Component hijacking is also possible on a chain of components, when the hijack-enabling flows span across component boundaries.

Defining component hijacking from a data flow perspective allows us to transform the vulnerability detection problem into an equivalent data flow analysis problem. A different but related topic is data leakage detection [15, 24], which looks for individual data flows that indicate sensitive data being propagated out of certain containment scope. Note that apps sending out sensitive data are not necessarily exploitable nor harmful (*e.g.* an app sends users' GPS information to remote servers for location-based services). Therefore, data leakage detection only reports outbound sensitive data flows without clarifying their security implications. In contrast, component hijacking vulnerability is always exploitable and undermines user's privacy. On the other hand, techniques for identifying component hijacking vulnerability can be applied to finding data leaks, but not vice versa. Because finding data leaks are essentially identifying special hijack-enabling flow that enable *G1* with all data sinks supposed to be accessible by attackers.

Component hijacking gives attackers the freedom to surreptitiously perform privileged actions and access private data. In our *threat model*, successful hijacks require users to willingly install the attacking app on their devices. To create a user population of decent size, attackers can resort to many illicit techniques that promote their apps in the market and lure users. Given component hijacking apps often requesting little to no permissions, users, even vigilant ones, tend to trust them easily. Although attackers cannot control, but only hope for, the availability of vulnerable apps on users' devices, the reality has been working towards attacker's favor due to the large number of under-trained Android developers and an overall lack of app quality assurance. Therefore, as a defensive effort, we designed CHEX to assist apps developers, testers, and market operators in filtering out apps vulnerable to component hijacking attacks before they reach end user devices. We chose to target CHEX on non-malicious apps, which constitute the majority of exploitation targets, so that we can safely assume a non-adversarial application scenario (*e.g.* heavy obfuscations and anti-analysis techniques are out of our concern) and solely focus on designing the detection and analysis method.

3. DETECTION AND ANALYSIS METHOD

CHEX follows a static program analysis approach, featuring a novel data-flow analyzer specially designed to accommodate Android's special app programming paradigms. Static analysis makes sense for vetting benign apps in that, the anti-analysis techniques that are commonly used in adversarial scenarios are out of scope, and the advantages of static analysis, such as its completeness and bounded time complexity, are well suited to addressing the vulnerability discovery problem.

Existing data-flow analysis and modeling methods are not immediately applicable to Andriod apps due to Android's special event-driven programming paradigm. Our flow- and context-sensitive analyzer, incorporated with a number of analysis techniques and models that we devised for Android apps, can efficiently discover data flows of interest within the entire app. Its underlying flow extraction mechanism is separated from the high level policies that define interesting flows. As a result, our data-flow analysis method can be applied to other applications than vulnerability discovery. Our method also offers the flexibility to choose if the Android framework code[3] needs to be included or simply modeled during the analysis, depending on specific usage scenarios. In this paper, we model the framework code for reasons discussed in Section 3.2.

Next, we present a concrete example to illustrate component hijacking vulnerabilities, as well as typical challenges associated with performing data-flow analysis on Android apps.

3.1 A component hijacking example

Our example is a hypothetical Android app that aggregates the popular location-based services and provides a one-stop solution for users. Figure 2 shows a critical `Service` component of the app. Upon requested by particular `Intents`, this component obtains user's location information and synchronizes it with a remote server. Despite that the component is intended for the app's internal use only, its developer carelessly left it open to other apps. This mistake is not uncommon partly because Android by default publicly exports components that register to accept particular `Intents`. Here, we demonstrate two possible component hijacking attacks on this example app and highlight the challenges associated with analyzing the code. The vulnerabilities in this example app are similar to those that we found in the real apps and reported in Section 5.2.

In Figure 2, Method `onBind` (Ln. 5) is invoked by the framework whenever a requester component connects to the `Service`. Android programming paradigm dictates that apps organize their logic into components of different kinds, whose life-cycles are managed by the framework in an event-driven manner. Each component implicitly or explicitly registers event handlers (*e.g.* Ln. 5, 10, and 32). These handlers serve as the entry points through which the framework starts or activates the component when handled events happen. Apps, even average-sized ones, can have a large amount of entry points of diverse object types and appearances, which posed the first challenge to our analysis:

    C1 : Reliably discovering all types of entry points (or event handlers) in their completeness.

Method `onBind` returns to the requester component an object that implements the `IBinder` interface (Ln. 6) — a common pattern to achieve inter-component communications in Android apps. The requester component can then send messages for the

---

[3]Android framework consists of the Dalvik runtime and Android system libraries. We refer to it as the framework hereafter. Note that apps (including system apps) are not part of the framework.

```
1  public class SyncLocSrv extends Service{
2    Location currLoc;
3    final Messenger mMessenger = new Messenger(new
         ReqHandler());
4
5    public IBinder onBind(Intent intent) {
6      return mMessenger.getBinder();
7    }
8
9    private class ReqHandler extends Handler {
10     public void handleMessage(Message msg) {
11       ...
12       switch (msg.what) {
13       case MSG_UPDATE_LOCATION:
14         // get GPS location
15         currLoc = lm.getLastKnownLocation(PROVIDER);
16         break;
17       case MSG_SYNC_LOCATION:
18         // sync GPS with specified URL
19         String url = msg.getData().getString("url");
20         String[] sendParams = new String[] {url};
21         new SendToNetwork().execute(sendParams);
22         break;
23       ...
24       default:
25         ...
26       }
27     }
28   }
29
30   private class SendToNetwork extends AsyncTask<String
         , String, String> {
31     // run in a separate thread
32     protected String doInBackground(String[] params) {
33       HttpClient hc = new DefaultHttpClient();
34       HttpPost pst = new HttpPost(params[0]); // URL
35       pst.setEntity(new StringEntity("gps:"+currLoc));
36       HttpResponse resp = hc.execute(pst);
37       return resp.toString();
38     }
39   }
40   ...
41 }
```

Figure 2: *Vulnerable component example*

Service to handle via the object. It is the framework that delivers the message and invokes handleMessage as an entry point (Ln. 10) when an incoming message arrives. Since the invocations of different entry points in an app can be asynchronous, we faced the second challenge:

C2 : Soundly modeling the asynchronous invocations of entry points for analysis.

Once connected to the example Service, an attacking app can exploit at least two separate component hijacking vulnerabilities to obtain the device location and perform network communications respectively, neither incurring any permission violations or user interactions. Specifically, the attacking app can send a MSG_UPDATE_LOCATION message, followed by a MSG_SYNC_LOCATION message, to coerce the message handler to first retrieve the device location (Ln. 15) and then send the data to a URL of the attacker's choice (Ln. 21). Alternatively, using a single MSG_SYNC_LOCATION message, the attacking app is able to make connections to arbitrary URL he supplies in the message. Based on Definition 1, these two particular cases of component hijacking are enabled by data-flows that respectively allow the attacker to (i) read the location data (*i.e.* realizing *G1*), and (ii) write

to the variable that controls the URL to be contacted (*i.e.* realizing *G2*).

Sometimes it takes multiple individual data-flows, loosely connected or partially overlapped, to enable one of the three goals described in Definition 1. In our example, two individual data-flows together allow the attacking app to read the location information (*i.e.* by forcing the vulnerable component to retrieve and send the location information to a specified URL): one flow carrying location data obtained on Ln. 15 to the HTTP Post on Ln. 36 and the other carrying requester-supplied URL on Ln. 19 to the same HTTP Post operation. To detect such hijack-enabling flows , a data-flow analyzer needs to tackle the challenge of:

C3 : Assessing the collective side-effects of individual data-flows and identifying converged flows of interest.

For optimized responsiveness, Android apps always perform blocking operations within the doInBackGround method in AsyncTask[4], such as network-send (Ln. 30). The message handler prepares the network-send parameter with the requester-supplied URL (Ln. 20). Once execute on the next line is called, the framework starts doInBackGround (Ln. 32) in a new thread, introducing another entry point to the component. Code that is reachable from each entry point is a segment of the entire component code. These segments can be statically determined via reachability analysis. We refer to them as *splits* (defined shortly). Although executing in separate contexts, splits are by no means isolated and in fact can relate to each other through inter-split data-flows. Heap and global variables used in different split can form these flows. Note that there exist two hijack-enabling flows that originate from the split started by handleMessage and reach to the split started by doInBackground: (i) the heap variable currLoc assigned with the location data (Ln. 15) and used as the HTTP Post content (Ln. 34), and (ii) the local array sendParams containing the URL (Ln. 20), implicitly passed to params on Ln. 32 by the framework as an entry point parameter, and used for the HTTP Post (Ln. 34). Therefore, our analyzer needs to be capable of:

C4 : Tracking data flows across splits and components.

In summary, this example demonstrates that a component is vulnerable to hijacks when it is exported to the public without limiting its interfaces to intended users. It also shows that using hijack-enabling data-flows to model the vulnerability is general and straightforward. A program analyzer aiming at detecting these flows faces four major challenges imposed by the unique Android programming paradigms (*C1, C2*) or by the complications of the data-flows (*C3, C4*). Next, we introduce our approach to conducting data-flow analysis on Android apps, with vulnerability detection as an application. We propose analysis methods and models that overcome the challenges discussed above. They are expected to be useful to other types of app analysis as well.

3.2 Analysis methods and models

The reason why we chose to model the framework, instead of including all its code into the analysis scope, is because of the complexity of analyzing the framework code and the simplicity of modeling its external data-flow behavior. Due to the framework's extensive use of reflections, mixed use of programming languages, and overwhelming code size, including the framework code into the analysis scope incurs a significant amount of overhead and introduces certain extent of inaccuracy to the analysis. Therefore,

---

[4]A convenient threading construct provided by the framework.

Algorithm 1 Entry points discovery

$M_f \leftarrow$ {Uncalled framework methods overridden by app}
$M_a \leftarrow$ {App methods overriding framework}
$E \leftarrow$ {Listeners in Manifest; Basic component handlers}
repeat
   $G \leftarrow$ BuildCallGraph(E)
   for all $m_a \in M_a \wedge m_a$ overrides $m_f \in M_f$ **do**
    if m_a's constructor $\in G$ **then**
      $E \leftarrow E \cup \{m_a\}$
    end if
   end for
until E reaches a fixed point
output E as entry point set

---

analysis that only require a partial knowledge on the framework's external behavior, such as data-flow analysis, should model rather than diving into the framework, to avoid unnecessary performance overhead and inaccuracy. In addition to modeling the framework in terms of its data-flow behavior, our analysis requires type information of framework-defined classes (the app-level classes are derived from these types). We will show in Section 4 that, such information can be easily extracted from the framework, which the analyzer uses to build the complete class hierarchy.

Entry point discovery: As the first step to deal with the multi-entry-point nature of apps and tackle *C1*, we designed an algorithm that discovers entry points in app code at a very low false rate, without requiring analyzing the framework code. To avoid ambiguity, we use the following definition of entry points in this paper:

DEFINITION 2. *App entry points are the methods that are defined by the app and intended to be called only by the framework.*

Entry points in an app can be large in amount, often with a great variety in their object types. For instance, each UI elements in an app can define multiple event listeners to be called at different moments as particular events happen. Similarly, each component can implement handlers to get notified about its life-cycle changes. Therefore, we avoided any manual efforts that use expert knowledge to generate sets of possible entry points, due to its error pruning nature and no guarantee for completeness.

Since the entry point methods are supposed to be called by the framework, the latter then requires the prior knowledge about these methods. In fact, there are only two ways for an app to define entry points that can be recognized by the framework: either via explicitly stating them in the manifest file, or implicitly overriding methods or implementing interfaces that are originally declared by the framework as app entry points. Those defined using the first option can be determined by parsing the manifest. To find the rest, our algorithm first generates the set of uncalled methods in the app that override their counterparts declared in the framework, and then excludes methods that are unreachable even by the framework (*i.e.* dead methods). Telling apart entry points from dead methods that override the framework, despite neither is called by the app, is based on two facts unique to entry points: (i) the containing class of any entry point always have at least one instantiated object (since app entry points are non-static methods), and (ii) there should be no app-level invocation on the original method that the entry point overrides or on any decedents of the original method in the class hierarchy. In contrast, dead methods that override the framework mostly cannot satisfy both conditions.

Our entry point discovery method, as formulated in Algorithm 1,

follows an iterative procedure until a fixed point is reached for the entry point set E. Method set M_f and M_a are generated by a simple scan of the class hierarchy and all call sites in the app code. E is initialized to include entry points declared in manifest files and basic component-life-cycle handlers defined in the code. Compared with other entry points, the component-life-cycle handlers have very few types and are the only entry points whose containing class is created by the framework (*i.e.* calls to their constructors are invisible at the app level). During each iteration, a new call graph G is built based on the already discovered entry points in E. Due to the new entry points added in the last iteration, the new G may contain previously unreachable methods and classes instantiations. A method $m_a \in M_a$ is added to E as a new entry point when m_a overrides a framework method or interface $m_f \in M_f$ and m_a's containing class is instantiated in G. We build the call graph using the entire E, rather than just using the newly discovered entry points in the previous iteration, so that the point-to analysis supporting the call graph builder can be as complete and accurate as possible. The algorithm terminates when E stops growing and contains all possible entry points. Very rare false positives can happen only when framework-declared methods are never called in the app while they are already overwritten by instantiated classes and made for app use.

App code splitting: Once all entry points are discovered, we model their asynchronous invocations and addresses *C2* with a novel technique named *app code splitting*. We define the concept of splits as follows:

DEFINITION 3. *A split is a subset of the app code that is reachable from a particular entry point method.*

From a static analysis perspective, app executions can be viewed as a collection of splits executing in all feasible orders, possibly interleaved. The idea of modeling app execution in terms of splits may seem challenging at the first glance. However, constrains imposed by the framework and our focus on data-flow analysis significantly simplify the realization of the idea. In fact, most splits in an app can only be executed in a sequential order (*i.e.* not interleaving each other), because the framework invokes the majority of app entry points in the main thread of an app. The mere exceptions are entry points of concurrency constructs, such as threads. Since our goal is to perform security vulnerability detection, concurrency-incurred data-flows are usually not a concern in this context due to their extreme unreliability to be reproduced or exploited. Therefore, we can safely approximate the app execution as sequential permutations of splits that are feasible under framework constraints.

Under this app splitting model, our data-flow analysis first computes the *split data-flow summary* (SDS) for each split in the app. It then starts the permutation process and, for each possible sequence of splits, generates *permutation data-flow summary* (PDS) by linking the SDS of each split in the sequence. As the permutation proceeds, each PDS is checked for *interesting data-flows* specified by pre-defined *policies*. Eventually, all possible data-flows can happen in the app are enumerated.

Figure 3 shows two SDS marked by dashed boxes. They are generated based on the two entry points, `handleMessage` and `doInBackground`, in our example discussed in Section 3.1. An SDS consists of intra-split data-flows whose end nodes represent: (i) heap variables[5] entering or exiting the split scope (depicted by

---

[5]Variables with a global scope, as opposed to local variables.

| (**getLastKnownLocation**, -1, Tag_SensSrc, Ln.15) | (**handleMessage**, 1, Tag_InputSrc, Ln.N) |
| (**currLoc**, Ln.X, <init>) | (SendToNetwork.**execute**, 1, Tag_TransSink, Ln.21) |

SDS: handleMessage

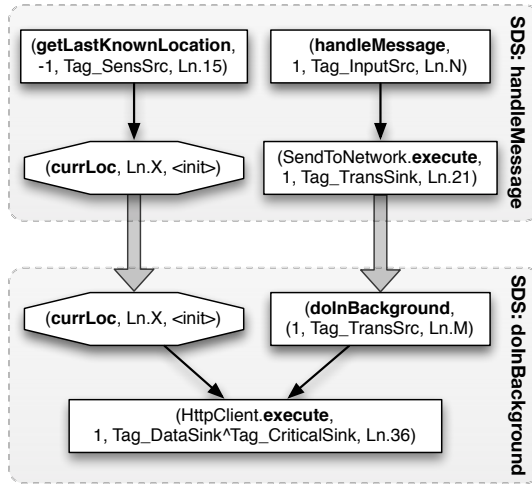| (**currLoc**, Ln.X, <init>) | (**doInBackground**, (1, Tag_TransSrc, Ln.M) |
| (HttpClient.**execute**, 1, Tag_DataSink^Tag_CriticalSink, Ln.36) | |

SDS: doInBackground

Figure 3: *Linked-SDS for the running example*

octagons); or (ii) pre-defined sources or sinks (depicted by rectangles). We omitted intermediate nodes in the SDS in Figure 3 to ease illustration. In essence, an SDS only contains data-flows within a split that may contribute to connecting a source to a sink (may resides in another split). We refer to these data-flows as *interesting-flows* hereafter. The upper SDS in Figure 3 has two isolated data-flows: the one on the left propagates the location data (a sensitive source, tagged as Tag_SensSrc) to a heap variable currLoc, and the one on the right carries the requester's input (tagged as Tag_InputSrc) to a transit sink; The lower SDS captures the convergence of a heap variable and a transit source at a sink associated with two tags (Tag_DataSink and Tag_CriticalSink, explained shortly). We compute SDS via a context- and field-sensitive data dependence analysis, identifying interesting-flows in the current split. As Figure 3 shows, heap variables are represented by their heap location key, which is a three-tuple in the form of $(field, allocSite, method)$, indicating the $field$ whose containing object was allocated at $allocSite$ in $method$ ($field$ of any array object is null). Pre-defined sources and sinks (data entry or exit points of the analysis's interest) are represented by a four-tuple, $(method, paramIndex, tag, callSite)$, indicating that a parameter of a method called at $callSite$ is a source or sink depending on the tag. Note that in Figure 3 the line numbers of $allocSites$ and $callSites$ that are not shown in Figure 2 are substituted by capital letters (*e.g.* Ln.X).

Our analysis method allows for a fairly flexible way of defining and extending tags associated with sources and sinks. Tags are used to differentiate sources and sinks with different semantic meanings given by the analyzer users based on their specific usage scenarios. Policies that specify interesting-flows can be defined based on the tags associated with their end nodes:

$P := F_{int} \bowtie [F_{int} \mid \emptyset]^n$, $F_{int} := [Tag] \rightsquigarrow [Tag]$, where $\bowtie$ defines a *join* relationship exists between two interesting-flows (*i.e.* two flows, or their extensions, intersect or converge with each other), and $\rightsquigarrow$ defines an interesting flow with two end nodes of specified tags. By supporting customizable tags and the *join* relationship in defining interesting-flows, our analyzer provides a means of expressing the side-effects of converged flows on a semantic level, which solves *C3*.

For component hijacking vulnerability detection, we define two general source tags, Tag_SensSrc and Tag_InputSrc, to mark the start points of interesting-flows that propagate sensitive

information or requester's input. We also define three general sinks to mark end points of interesting-flows that are to make data publicly accessible (Tag_PublicSink), make data accessible to specified entities (Tag_SpecifiedSink), or write data into critical data regions (Tag_CriticalSink). With these tags defined, we can easily convert Definition 1 into three simple policies to capture hijack-enabling flows:

P1 : {Tag_SensSrc $\rightsquigarrow$ Tag_PublicSink}

P2 : {Tag_InputSrc $\rightsquigarrow$ Tag_CriticalSink $\bowtie$ Tag_SensSrc $\rightsquigarrow$ Tag_SpecifiedSink}

P3 : {Tag_InputSrc $\rightsquigarrow$ Tag_CriticalSink}

These policies are checked on every newly generated PDS as the split permutation continues. As for the example discussed in Section 3.1, our analyzer can detect the hijack-enabling flows, satisfying *P2* and *P3*, from a PDS that links the SDS of *handleMessage* with that of *doInBackGround*, as shown in Figure 3.

The PDS generation is carried out by two basic operations – link and unlink an SDS. The link operation adds a new SDS into a PDS if inter-split data-flows exist from the latter to the former. It draws data-flow edges (*e.g.* the two thick edges in Figure 3) from leaf nodes in the PDS to those reachable root nodes in the new SDS. For Android apps, the only two channels through which data can flow across splits are: heap variables sharing the same location key tuple, and framework API pairs that transit data among splits. We introduce a pair of special tags, Tag_TransSink and Tag_TransSrc, to model these API pairs. The link operation can reject the SDS if no edge can be drawn and the SDS does not contain flows starting with any pre-defined source. A rejection suggests that the new SDS has no effect on any potential propagation of interesting-flows in the current PDS, and thus, there is no need to add it. Unlink operation simply reverts the last link operation.

Intuitively iterating through all split permutations can be a prohibitively expensive operation for apps with a large number of entry points. We leverage on the continuity of data-flows across splits to carry out a simple but effective search pruning. The depth-first search only appends an SDS to the current permutation if it is accepted by the link operation and then continues iterating along that path. As shown in Section 5, this pruning greatly reduces the search space and time overhead of the permutation process. The permutation also considers a few constraints on the launch order of splits that handle life-cycle events of basic components (*e.g.* entry points relating to component initialization and termination are called in fixed orders).

Finally, *C4* is addressed, because all interesting-flows in an app, both intra-split and inter-split ones, are constructed during the split permutation process. Our app splitting technique enables a data-flow analysis that is more efficient and better accommodates the event-driven nature of Android apps, than the conventional methods, which synthesize a main function explicitly invoking event handlers. App splitting creates a divide-and-conquer theme. The sub-problems (*i.e.* constructing intra-split data-flows and SDS) are significantly easier and smaller in scale than the original problem (*i.e.* constructing data-flows for an entire app, as faced in the conventional methods). The merge process (*i.e.* permuting splits) can be very fast, as shown in Section 5. Moreover, due the mutual independence among SDS, they can be built in parallel and cached for reuse (*e.g.* SDS for common libraries can be built once and reused when analyzing all apps that make use of them) to further improve the performance.

4. IMPLEMENTATION OF DALYSIS AND CHEX

We built a generic Android app analysis framework named Dalysis, which stands for <u>Da</u>lvik bytecode ana<u>lysis</u>. As suggested by its name, Dalysis directly works on off-the-shelf app packages (or Dalvik bytecode) without requiring source code access or any decompilation assistance. Previous app analysis efforts that relied on decompiled source code have two major drawbacks — heavy performance overhead and incomplete code coverage. As reported by Enck *et al.* [16], the state of the art technique to decompile an app, on average, takes about 27 minutes and leaves 5.56% of the source code failed to be recovered. Conducting analysis at the dalvik bytecode level overcomes these issues. In addition, unlike x86 binary code, bytecode retains sufficient program information from the high level language and does not have any parsing ambiguity, thus serves as an ideal analysis subject.

To our best knowledge, Dalysis is the first generic analysis framework that operates on Dalvik bytecode and intended to support multiple types of program analysis tasks. Next, we introduce the internals of Dalysis that can facilitate the understanding of the implementation of CHEX, our component hijacking analyzer built based on Dalysis. We leave out the low-level system building details as they are out of the scope of this paper.

4.1 Dalysis framework

The front end of Dalysis consumes an Android app package (`.apk`) at a time. It retrieves package information from metadata files and translates the Dalvik bytecode into an intermediate representation (IR), based on which the back end analyzers carry out their tasks. The front end starts the IR generation process by parsing the input bytecode file. Dalysis employs an open source Dalvik bytecode parser named `DexLib`, part of a well-known disassembler for Android apps [2]. `DexLib` provides useful interfaces to programmatically read embedded data, type information, and Dalvik instructions from a bytecode file. Dalysis allows different analysis to choose either include the entire Android framework code or model its external behaviors, which is achieved by linking two different versions of the runtime library into the analysis scope. The front end constructs the class hierarchy, performs an semantical IR translation from Dalvik and Java bytecode (Android framework libraries are compiled into java bytecode), and then hands over the IR to backend analyzers.

We adopted our IR from the `WALA` project [5], a popular static analysis framework for Java, for two reasons: the semantic proximity between Dalvik bytecode and the IR and a wide selection of basic analyzers developed for the IR by the `WALA` community. The translation process is mostly straightforward, since both instruction sets follow the register-machine model and retain a similar amount of information from the same high level language (*i.e.* Java). However, a handful of instructions that are unique to Dalvik virtual machine require special handling during the translation process. For example, the `filled-new-array` instruction allocates and initializes an array in one step; And the `move-result` instruction retrieves the result of the previous call from the special `result-register`. Following the semantic translation is the final task for the front end – static single assignment (SSA) conversion. The conversion performs an abstract interpretation on each method, wherein the define-use chain is determined for each Dalvik register as well as its mapping to the local variable on Java level. New instructions are generated, as a side effect incurred by the flow function of the abstract interpretation. As a result of variable renaming (*i.e.* a register model conversion), newly generated
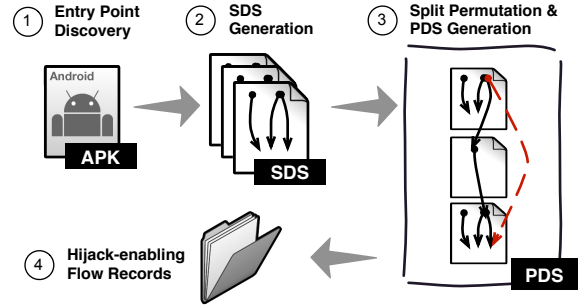

Figure 4: *CHEX workflow*

instructions operate on a conceptual register model with unlimited registers, each of which can only be assigned once as required by SSA form. `Meet` operations happen at basic block boundaries. As a result, ϕ variables are generated to merge two or more values that may flow into a same variable in the current basic block from predecessors in the control flow graph. Converting the IR into SSA form can simplify various types of program analysis, especially data-flow related ones, such as definition reachability test, constant propagation and etc. In fact, many existing analyzers for `WALA` assume an SSA IR.

The back end of Dalysis hosts a variety of analyzers and provides them the interfaces to access the IR, the class hierarchy, and other useful information. Some basic analyzers released by `WALA`, such as the point-to analysis and the call graph builders, are included in Dalysis. These building-block analyzers can be found useful by many advanced analyzers. Dalysis itself is not specific to any particular flavor of app analysis — it is designed to be a generic framework that can enable as many types of analysis as possible on Android apps. For example, CHEX demonstrates how we implemented the data-flow analysis methods, introduced in Section 3.2, by using the Dalysis framework.

Dalysis is implemented in Java with 15,897 lines of source code, excluding 3rd party libraries. The building process took us a significant amount of efforts due to a lack of similar work and reusable code. But most efforts were spent on tackling engineering related issues or implementing existing algorithms from the programming language community, therefore we do not intend to claim these efforts as contributions in this paper. We also omit the implementation details of Dalysis that should be oblivious to analyzer designers, which is out of the scope of this paper.

4.2 CHEX: Component hijacking examiner

CHEX realizes our data-flow analysis methods and models discussed in Section 3.2. It detects hijack-enabling flows based on policies *P1-3*, with a set of 180 sources and sinks that match the tags defined by these two policies. This set was constructed semi-automatically to cover a relatively wide range of hijack-enabling flows that affect the sensitive resources managed by the system (*i.e.* protected by Android permissions and accessed uniformly across apps). Parts of the sensitive sources (`Tag_SensSrc`) were selected based on the API-to-permission mapping provided by [19]. This set is adequate for our testing and evaluation purpose, but it is not meant to be complete. In fact, it can be extended with source and sinks specific to individual apps, so that CHEX can capture hijack-enabling flows in app's semantics.

As shown in Figure 4, entry point discovery starts at first. It queries Dalysis front-end for information necessary to the ini-

tialization process, such as event listeners defined in manifest and method overloading relationships (shown in Algorithm 1). CHEX makes multiple different uses of the call graph builder from `WALA`, which can be configured to have different degrees of context sensitivity. For each iteration in the entry point discovery process, we generate a context-insensitive call graph, for the least performance overhead and the unnecessity of context sensitivity in this scenario (*i.e.* we use the call graph only to conservatively estimate if a method was called or a class was instantiated before).

For each discovered entry point, or more specifically, the split started by that entry point, CHEX builds an SDS to summarize its data-flow behaviors that may contribute to forming any hijack-enabling flow (Step 2 in Figure 4). Building SDS is a computation heavy step in the entire analysis because it is where all intra-split data-flows are constructed directly by analyzing the IR. In comparison, in a later step, the permuter generates inter-split flows and PDS based on simple rules determining the connectivity between two intra-split flows.

Conventional data-flow analysis approaches solve data-flow equations through an iterative process. This process is expected to reach a fix-point after limited iterations of basic-block state changes made by transfer functions. However, for the purpose of building SDS, we can safely avoid this procedure and still be able to check interesting-flows, thanks to the SSA IR and our abstraction of the flow checking problem. Specifically, the SSA conversion carried out by the front end has already conducted a basic data-flow analysis and saved information (*e.g.* variable use-define chains and *etc.*) that can greatly facilitate the construction of system dependence graphs. Inspired by the way of utilizing system dependence graphs in the classic program slicing algorithm [25], we convert the problem of checking interesting data-flows into an equivalent graph reachability test problem. We test the connectivity of source-sink pairs on customized system dependence graphs that only have data-dependence edges (referred as *data-dependence graph*, or DDG). A source-sink pair that is connected on a DDG indicates the existence of a data-flow from the source to the sink. Compared with the conventional approaches, this abstraction offers us a better leverage on the existing IR and avoids unnecessary analysis work, yet still achieving the same goal.

DDG is constructed in a similar way as system dependence graph is in [25], but without generating control-dependence edges. Each node in DDG represents either a normal SSA statement or an artificial statement to model inter-procedure parameter passage. An edge is drawn from node S_1 to node S_2 only when the variable defined by S_1 is directly used by S_2. Intra-procedural edges between scalar variables are drawn with the help of local use-define chains implied from the SSA IR. Identifying inter-procedural dependencies among heap variables requires a call graph with a proper degree of context sensitivity and an inter-procedural definition reachability analysis. We chose a `0-1-CFA` call graph builder with the `call-string` context sensitivity (*i.e.* using the calling string to identify a particular node in the call graph), for its sufficient accuracy and acceptable performance overhead. With the call graph, regular parameter and return passing edges can be added between the corresponding callers and callees. The definition reachability analysis provides information about (transitive) heap variable accesses in a method, which is needed to create heap related nodes and draw edges between them (inter-procedural heap variable accesses are modeled as artificial parameters or returns).

Before used for the interesting-flow discovery, a DDG needs to go through an edge inflation process, as a way to model data dependencies that are still missing. Missing edges are resulted from out-of-scope code (*i.e.* methods defined outside of the analysis scope). Thus we need to model the external data-flow behavior of such code. The modeling can be easily done by means of adding artificial edges into the DDG, bases on two simple rules: (i) for methods with returns, the return value is dependent on all parameters (*i.e.* drawing edges from each `ParameterCaller` node to the `ReturnCaller` node); and (ii) for return-less methods, the first parameter (*i.e.* `this*` for non-static methods) is dependent on all other parameters, if any (*i.e.* drawing edges to the define node of the first parameter from other `ParameterCaller`). Exceptions to these rules do exist, but only very few happen frequently enough that we need to specially handle, such as several methods of strings and collection types.

With the DDG is generated, searching for interesting flows becomes intuitive. CHEX first picks two sets of nodes from the graph, S_{start} and S_{end}, where S_{start} contains pre-defined sources (*i.e.* start points of inter-split flows), and S_{end} contains pre-defined sinks (*i.e.* end points of inter-split flows). CHEX then constructs the SDS as it traverses the DDG – a flow is added to the SDS if it starts from a node in S_{start} and ends with a node in S_{end}. The resulting SDS serves as a gadget for the permuter to compute PDS (Step 3 in Figure 4). Although the SDS building process is the most computation-intensive step during the entire analysis, the problem size is already greatly reduced, comparing with conducting the similar analysis on the whole app without app splitting. Tasks performed during the SDS construction, such as point-to analysis, generally scale poorly as the app size increases. Therefore, dividing the app into smaller but self-contained splits can help with the performance, and alleviates the scalability issue for large apps. In addition, due to their independence, SDS constructions for different splits can be carried out in parallel in performance-critical and computing-resource-rich scenarios, to further reduce the overhead.
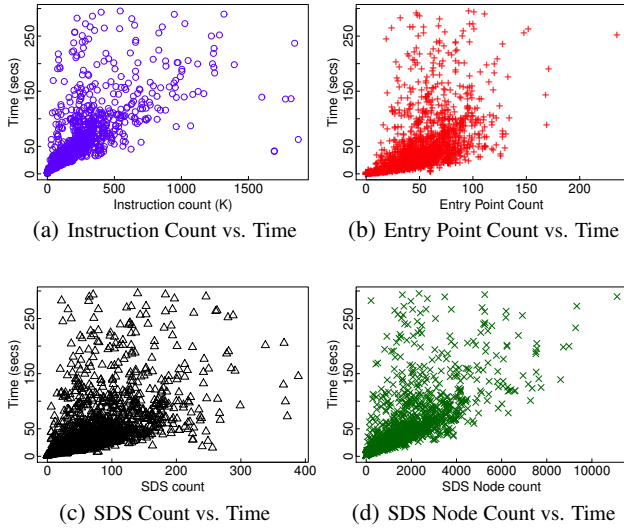
The split permuter always starts a new sequence with a split from an exported component, a constraint to reflect the causal relationship between external requests and potential hijack-enabling flows. The permutation is implemented as a regular depth-first-search with pruning and configurable search space. For example, the maximum DFS depth specifies the maximum number of splits a feasible hijack-enabling flow can span through, a practical trade-off between performance and completeness. As the permutation proceeds, interesting flows in the current PDS are matched with policies *P1-P3* for hijack-enabling flows. Node tags and the $\rightsquigarrow$ relation can be simply checked on individual interesting flows. As for the $\bowtie$ relationship, we test if two interesting flows merge into a new variable or join at a same method call site. Discovered hijack-enabling flows are recorded (Step 4 in Figure 4) with detailed information, such as the corresponding paths in PDS, the split sequence, and the policy they satisfy. Such information can assist app developers or security researchers to verify and fix vulnerabilities.

CHEX consists of 5,945 lines of Java code. When linked with the dependencies in Dalysis, it can be built into a standalone program and deployed to vet real-world apps, as we did when evaluating it.

5. EVALUATIONS AND EXPERIMENTS

We carried out an in-depth evaluation on CHEX in terms of its performance and accuracy. In addition, our large-scale empirical experiment revealed interesting facts about vulnerable Android apps, which are expected to contribute to a community awareness

(a) Instruction Count vs. Time   (b) Entry Point Count vs. Time

(c) SDS Count vs. Time   (d) SDS Node Count vs. Time

Figure 5: *Execution Time Characteristics of CHEX*

Figure 6: *Performance Decomposition*

of real-world component hijacking vulnerabilities and caveats of analyzing them.

5.1 System evaluation

We exercised CHEX with a large set of real-world apps, S_{pop}, containing about 5,486 free popular apps we collected in late 2011. S_{pop} consists of around 3,486 apps from the official Android market and 2,000 from alternative markets. The experiments were conducted on a cluster of three computers, each equipped with an Intel Core i7-970 CPU and 12GB of RAM. During the experiments, we launch concurrent CHEX instances on 64-bit JVM with a maximum heap space of 4GB. To optimize the throughput, we limit the processing time of each app within 5 minutes.

Performance: We instrumented CHEX to measure its execution time while it examining apps in S_{pop}. The median processing time for an app is 37.02 seconds with the interquartile range (IQR) of 161.87 seconds, which suggests that CHEX can quickly vet a large amount of apps for component hijacking vulnerabilities. 22% apps needed more than 5 minutes to be analyzed thus timed out in our experiments. In practice, with more computing resources available, a more generous time-out value should be used.

We found that CHEX's execution time varies significantly across different apps. As a result, we studied the impact of four app-specific factors that may affect CHEX's execution time the most (see Figure 5). Although these factors are in a strong correlation with the execution time, no single factor dominates it (*i.e.* none poses major bottleneck to the performance). Furthermore, we decomposed the execution time into three parts, corresponding to the three analysis phases each app goes through, as shown in Figure 6. In general, SDS construction (or split permutation) causes the majority of the time overhead, whereas entry point discovery and DFS generation often finish fast.

Some findings acquired during the evaluation also prove that the app analysis challenges we tackled in this work ($C1-C4$) are very common to encounter when analyzing real apps. On average, we found 50.37 entry points of 44 unique class types in an app. Moreover, the number of entry points is not directly related to the app size. Apps implementing complex user interfaces or requiring frequent user interactions (*e.g.* games) tend to have more entry points than others. About 99.70% of apps contain inter-split data flows,
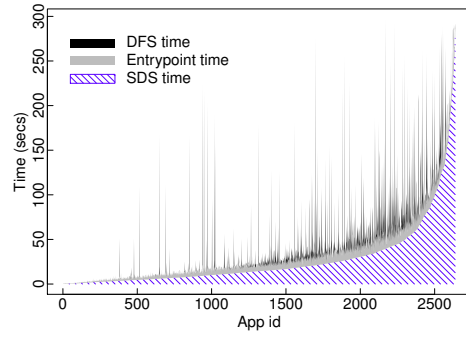
which strongly indicates the necessity of analyzing such flows the contexts created by different entry points.

Accuracy: Among the 5,486 apps in S_{pop}, CHEX flagged 254 as potentially vulnerable to component hijacking attacks. Due to the lack of a ground truth, we manually verified all the flagged apps by checking if the discovered hijack-enabling flows are indeed feasible and exploitable by attackers. This verification process largely replied on human expert knowledge with the assistance of well-known Android app disassemblers and decompilers. In the end, we identified 48 flagged apps as false positives, which yields a true positive rate above 81%. The main causes for the false positives are infeasible split permutations and apps' complicated input validations that CHEX cannot understand. Although the false positive rate is acceptable in a vulnerability filtering scenario, we argue that the first cause can be minimized by incorporating Android domain knowledge into the permutation pruning logic, while the second cause is a difficult but orthogonal issue to this work (*i.e.* checking the quality of program's input validation).

5.2 Case studies

Our manual verification process also helped us gain practical insights into the component hijacking vulnerabilities. All 206 apps that are confirmed as vulnerable can be roughly categorized into five classes, as shown in the first column of Table 1. It clearly shows that, in addition to vulnerabilities exploited by confused deputy attacks on the Android permission system, other vulnerability classes also fall into the scope of component hijacking and can be detected by CHEX. The second column refers to Definition 1 and indicates the hijacking type for each vulnerability class. To improve the community's awareness and understanding of component hijacking, we selected at least one app from each class and conducted the following case studies. We hide part of the app package names as a precaution to not leak undisclosed vulnerability information.

Case A1 in the data theft class resembles the example app we used in Figure 2. One of its components obtains the GPS location and saves it to a global variable. Another component initializes a URL parameter using a string provided by an arbitrary app via Intent, and sends the GPS information to the URL. An attacker thus can steal the sensitive location information by sending a crafted Intent to the second component, causing the GPS location to be sent to the attacker controlled server.

Apps can also leak their private, permission-protected capabilities through public components, as previously reported. Case B1 has a public component that takes a string from another app's Intent and uses it as a URL for Internet connection. Likewise, a public component of case B2 uses a string from an Intent as the host name for socket connections. These vulnerable apps essentially give out the Internet permission to all other apps who may

Vulnerability Class	Hijacking Type	Case-Studied Vulnerable App
Data Theft	G1/G3 (Read Write)	**[Case A1] de.cellular.xxx:** Sending GPS data to URL specified by input string
Capability Leak	G2 (Write)	**[Case B1] com.appspot.xxx:** Input string used as URL for Internet connection **[Case B2] com.gmail.xxx:** Input string used as hostname for socket connection
Intent Proxy	G1/G2 (Read Write)	**[Case C1] com.outfit7.xxx**: Object embedded in input used to start Activity
Code Injection	G2 (Write)	**[Case D1] com.utagoe.xxx:** Input string used for raw SQL query statement **[Case D2] cn.myprivate.xxx:** Input string used as shell command
Data tampering	G2 (Write)	**[Case E1] com.akbur.xxx:** Input string submitted to server as game score

Table 1: *Detected vulnerability class and case-studied apps*

not have it. For example, a malicious app can exploit these apps to transmit information to an arbitrary Internet server, or even launch network attacks against a victim server. We have observed Internet capability leakages in both Activity and Service components of vulnerable apps. In the cases of Activity components, the exploited components can be forced to display specified remote content to the user; Whereas exploits on vulnerable Service components can be carried out more stealthily, because Service components execute in the background (in this case, communicating with attacker controlled servers) without interacting with users.

Intent proxy is another class of vulnerabilities that can be exploited in a fashion similar to capability leak. Case C1 accepts an input Intent (X) that embeds another Intent (Y). It then starts a new Activity per Y's request using its own identity. More specifically, in the `OnResume()` function of C1, the Intent Y is retrieved from the `Bundle` object through the key "intent". Next, Intent Y is directly passed to `startActivity` without checking any properties of Intent Y. With this proxy, an attacking app can hide its identity and start activities, even those protected by permissions that C1 has but the attacking app does not.

Android heavily relies on internal SQL database to organize system and app data, such as contacts and app private information. Apps can interact with its database using APIs that take SQL statements as arguments. Case D1 passes an input string from an Intent directly into a raw SQL query, which allows attackers to inject SQL statements to manipulate the database or even cause system comprises. In addition, we have also uncovered more subtle SQL injection vulnerabilities in many apps, which use parameterized query instead of raw query, but in a non-parametric form. In particular, the vulnerable apps construct the selection clause of a query by directly inserting unescaped strings from Intent, instead of passing them in a parameter array. Such practices allows the attackers to inject an arbitrary condition into the selection clause, and derail the execution of the query, causing unexpected behaviors of the victim app. Besides SQL injections, a similar but more harmful vulnerability, as in Case D2, is the shell command injection, where app issues Linux shell commands using unchecked input strings.

The last class, data tampering, leads to private or critical data being overwritten by attackers. Case E1 is a game that reports user's score to a remote server for ranking purposes. However, the reporting component is made public and reports arbitrary scores specified by a requestor, which creates an easy way for cheating the game's online scoreboard. We also observed a more security-critical case where the payment URL of an online shopping app can be modified by attackers. The extent of damage by this type of vulnerability is highly dependent on the function of individual apps, as well as the robustness of client-server interactions of the apps.

6. DISCUSSIONS

As the evaluation shows, CHEX do have false positives. However, they can be reduced by addressing two limitations of our current prototype. First, our prototype does not leverage on much domain knowledge about the partial orders in which Android components and their entry points can run or interleave. This design choice was made because PDS construction enforces the data-flow continuity between splits, which sorts out the majority of infeasible split permutations but not all. In addition, building such domain knowledge, possibly time-consuming and error-prune, is out of this work's scope. We argue that when adopted in practice, CHEX can always incorporate new constraints into the split permeation, which not only reduce the false positive rate but also improve the performance. Second, our current prototype is unable to recognize false hijack-enabling flows that are sanitized by complicated logic (*e.g.* regular expression matching and etc.), because it by itself is an open research problem. On the other hand, we observe that the majority of apps rely on simple framework APIs (*e.g.* `checkCallingPermission`) and constant string matching to carry out effective input validation, which are already handled by CHEX.

The fact that CHEX only checks data-flows to detect vulnerabilities may cause false negatives. Rare vulnerable components may exist that enable hijacking attacks without explicit data-flows. In these cases, date dependencies are essentially encoded into control dependencies and thus sources and sinks are no longer connected via data-flows. We could selectively track control dependence for certain sources (*e.g.* `Tag_InputSrc`) in our SDS, so that implicit intra-split data-flows can be considered during analysis. On the other hand, control dependency analysis can also easily bring false positives. The study of this trade-off is out of the scope of this paper.

7. RELATED WORK

Event-driven (callback-based) programming is widely used in implementing graphical user interface (GUI) and web systems. To statically analyze GUI systems, previous work [30, 31] leverage on domain knowledge to identify and to configure the entry point (callback) methods. In web systems, event handler functions are easy to identify given the uniform ways to define them. However, in Android, the large number of entry point types makes it difficult to identify them completely—previous work relied on specific domain knowledge to detect common component entry points without guarantee for completeness [21]. We devise a heuristic-based approach to discover all possible entry points to the apps with low false positives. To model the execution of multiple entry points, previous work [30, 31] employ a synthetic main function to mimic

the event loop dispatcher in GUI systems. We introduce SDS to summarize intra-split data-flows and permute the splits to model their asynchronous invocations and derive the inter-split data-flow behaviors. Comparing with [30, 31], we divided the global data-flow analysis problem into much smaller but self-contained sub-problems, which improves the performance and scalability.

Static analysis and model checking have a history in assisting vulnerability discoveries [9, 18, 26, 28]. For web systems, Jovanovic *et al.* designed Pixy [26] to detect input validation flaws in server side scripts written in PHP through an inter-procedural context-sensitive data flow analysis. A similar study has been carried for cross site scripting vulnerabilities [33]. Bandhakavi *et al.* applied a context-sensitive and flow-sensitive static analysis for analyzing the security vulnerabilities of Firefox plugins written in JavaScripts [7]. For Java programs, Livshits *et al.* designed a datalog language to describe the security policies that direct vulnerability detection [28]. Tripp *et al.* built an industrial strength static taint analysis tool [32]. Comparing with the aforementioned efforts, we focused on detecting component hijacking vulnerabilities in Android apps. We first tackled general challenges faced by static app analyzers due to Android's special programming paradigm, and then proposed a data-flow-based detection approach.

Security mechanism based on information flows, such as JIF [29], HiStar [34] and Asbestos [13], are also related in that our work define and detect component hijacking by means of data-flow policies, despite that we do not enforce the policies in runtime.

Mobile security issues have gained much attention recently. Malware are not strangers for both the official Android market and alternative ones [36]. Research efforts were made on detecting repackaged apps [35] or apps with known malicious behavior [21, 37]. Recently Google also launched its malware filtering engine [1]. Information leakage is another major security threat for mobile devices. Kirin [17] detects apps whose permissions might indicate potential leakage. TaintDroid [15] leverages dynamic taint analysis to detect information leakage at runtime. PiOS [14] addressed the same problem using static analysis for iPhone app. In general, information leakage detection reveals the potential out bound propagation of sensitive information, which might be benign in many cases. Instead, component hijacking detection captures the information leakages resulted from an exploitation (*i.e.* sensitive data theft), in addition to other hijacking types.

Enck *et al.* introduced Ded [16] to convert Dalvik bytecode back to Java bytecode, and then used existing decompilers to obtain the source code of the apps for analysis. Our Dalysis framework directly converts Dalvik byte code to an SSA IR and enables various types of static analysis. Unlike the decompilation process, our IR conversion is sound (*e.g.* no heuristics or failures) and costs much less time. We model the Android framework and its special program paradigm rather than coarsely treating apps as traditional Java programs. As a result, our analysis is more tailored for Android apps and thus has better precision.

Android mediates access to protected resources using a permission system. However, its effectiveness hinges on app developers correctly implementing it. Chin *et al.* showed that apps may be exploitable when servicing external intents [10]. They built ComDroid to identify publicly exported components and warn developers about the potential threats. For this purpose, it is sufficient for ComDroid to only check app metadata and specific API usages, rather than performing an in-depth program analysis as CHEX does. As a result, warned public components are not necessarily exploitable or harmful (*i.e.* the openness can be by design or the component is not security critical). On the other hand, Android permission system is subject to several instances of the classic confused deputy attack [23]. As demonstrated by [11, 20, 21, 27], an unprivileged malicious app can access permission-protected resources through privileged agents (or app components) that do not properly enforce permission checks. Recently proposed runtime mitigations either reduce the agent's effective permissions to that of the original requestor [20] or inspect the IPC chains for implicit permission escalations [8, 12]. While these runtime solutions are effective at protecting end users adopting them, scalable detection methods for the problematic agents in question (*i.e.* hijack-able components) are still important to have in order to prevent vulnerable apps from reaching the vast users in the first place. Grace *et al.* [21] employed an intra-procedural path-sensitive static analysis to discover permission leaks specific to stock apps from multiple device vendors. In comparison, CHEX targets at a more general vulnerability in all types of Android apps and performs inter-procedural analysis with high degrees of sensitivity. Thanks to our novel entry point discovery and app-splitting techniques, CHEX is capable of accommodating Android's special programming paradigm and finding complex hijack-enabling flows. It is also noteworthy that the component hijacking attacks we address includes but is not limited to attacks targeting at permission-protected resources.

8. CONCLUSIONS

In conclusion, we defined and studied the component hijacking problem, a general category of vulnerabilities found in Android apps. By modeling the vulnerabilities from a data-flow perspective, we designed a static analyzer, CHEX, to detect hijack-enabling data-flows in a large volume of apps. In doing so, we introduced our method to automatically discover entry points in Android app, as well as the novel analysis technique, app splitting, as an efficient and accurate way to model executions of multiple entry points and facilitate global data-flow analysis. We also built the Dalysis framework to support various types of static analysis directly performed on Android bytecode. CHEX prototype was implemented based on Dalysis and was evaluated with 5,486 real-world apps. The empirical experiment demonstrated a satisfactory scalability and performance of our analysis method, as well as provided an insight into the real-world vulnerable apps we detected.

9. ACKNOLEDGEMENT

10. REFERENCES

[1] Android and security.
    http://googlemobile.blogspot.com/2012/02/android-and-security.html.
[2] Baksmali: a disassembler for Android's dex format.
    http://code.google.com/p/smali/.

[3] Google's 10 billion android app downloads. www.wired.com/gadgetlab/2011/12/10-billion-apps-detailed/.

[4] Quality of Android market apps is pathetically low. http://www.huffingtonpost.com/2011/06/20/android-market-quality_n_880478.html.

[5] WALA: T.J. Watson libraries for analysis. http://wala.sourceforge.netl.

[6] Android application components. http://developer.android.com/guide/topics/fundamentals.html#Components, 2012.

[7] BANDHAKAVI, S., KING, S. T., MADHUSUDAN, P., AND WINSLETT, M. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX Security Symposium* (2010).

[8] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Tech. Rep. TR-2011-04, Technische Universitat Darmstadt, 2011.

[9] CHEN, H., AND WAGNER, D. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM CCS* (2002).

[10] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in android. In *Proceedings of the 9th MobiSys* (2011).

[11] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege escalation attacks on android. In *Proceedings of the 13th ISC* (2010).

[12] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[13] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM SOSP* (2005).

[14] EGELE, M., KRUEGEL, C., KIRDA, E., AND VIGNA, G. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the 19th NDSS* (2011).

[15] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX OSDI* (2010).

[16] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[17] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM CCS* (2009).

[18] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX Security Symposium* (2010).

[19] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM CCS* (2011).

[20] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[21] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th NDSS* (2012).

[22] GUNDOTRA, V., AND BARRA, H. Android: Momentum, mobile and more at Google I/O. http://www.google.com/events/io/2011/.

[23] HARDY, N. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev. 22*, 4 (1988), 36–38.

[24] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM CCS* (2011).

[25] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *SIGPLAN Not. 23*, 7 (1988), 35–46.

[26] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the IEEE S&P'06* (2006).

[27] LINEBERRY, A., RICHARDSON, D. L., AND WYATT, T. These aren't permissions you're looking for. In *Proceedings of the Blackhat'10* (2010).

[28] LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[29] MYERS, A. C. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM POPL* (1999).

[30] STAIGER, S. Reverse engineering of graphical user interfaces using static analyses. In *Proceedings of the 14th IEEE WCRE* (2007).

[31] STAIGER, S. Static analysis of programs with graphical user interface. In *Proceedings of the 11th IEEE CSMR* (2007).

[32] TRIPP, O., PISTOIA, M., FINK, S. J., SRIDHARAN, M., AND WEISMAN, O. TAJ: effective taint analysis of web applications. In *Proceedings of the ACM PLDI '09* (2009).

[33] WASSERMANN, G., AND SU, Z. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th ACM ICSE* (2008).

[34] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *Proceedings of the 7th USENIX OSDI* (2006).

[35] ZHOU, W., ZHOU, Y., JIANG, X., AND NING, P. DroidMOSS: Detecting repackaged smartphone applications in third-party android. In *Proceedings of ACM CODASPY'12* (2012).

[36] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on S&P'12* (2012).

[37] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 20th NDSS* (2012).