AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications

Mu Zhang
Department of EECS
Syracuse University
muzhang@syr.edu

Heng Yin
Department of EECS
Syracuse University
heyin@syr.edu

***Abstract*—*Component hijacking* is a class of vulnerabilities commonly appearing in Android applications. When these vulnerabilities are triggered by attackers, the vulnerable apps can exfiltrate sensitive information and compromise the data integrity on Android devices, on behalf of the attackers. It is often unrealistic to purely rely on developers to fix these vulnerabilities for two reasons: 1) it is a time-consuming process for the developers to confirm each vulnerability and release a patch for it; and 2) the developers may not be experienced enough to properly fix the problem. In this paper, we propose a technique for automatic patch generation. Given a vulnerable Android app (without source code) and a discovered component hijacking vulnerability, we automatically generate a patch to disable this vulnerability. We have implemented a prototype called *AppSealer* and evaluated its efficacy on apps with component hijacking vulnerabilities. Our evaluation on 16 real-world vulnerable Android apps demonstrates that the generated patches can effectively track and mitigate component hijacking vulnerabilities. Moreover, after going through a series of optimizations, the patch code only represents a small portion (15.9% on average) of the entire program. The runtime overhead introduced by AppSealer is also minimal, merely 2% on average.**

I. Introduction

With the boom of Android devices, the security threats in Android are also increasing. Although the permission-based sandboxing mechanism enforced in Android can effectively confine each app's behaviors by only allowing the ones granted with corresponding permissions, a vulnerable app with certain critical permissions can perform security-sensitive behaviors on behalf of a malicious app. It is so called confused deputy attack. This kind of security vulnerabilities can present in numerous forms, such as privilege escalation [1], capability leaks [2], permission re-delegation [3], content leaks and pollution [4], component hijacking [5], etc.

Prior work primarily focused on automatic discovery of these vulnerabilities. Once a vulnerability is discovered, it can be reported to the developer and a patch is expected. Some patches can be as simple as placing a permission validation at the entry point of an exposed interface (to defeat privilege escalation [1] and permission re-delegation [3] attacks), or withholding the public access to the internal data repositories (to defend against content leaks and pollution [4]), the fixes to the other problems may not be so straightforward.

In particular, component hijacking may fall into the latter category. When receiving a manipulated input from a malicious Android app, an app with a component hijacking vulnerability may exfiltrate sensitive information or tamper with the sensitive data in a critical data repository on behalf of the malicious app. In other words, a dangerous information flow may happen in either an outbound or inbound direction depending on certain external conditions and/or the internal program state.

A prior effort has been made to perform static analysis to discover *potential* component hijacking vulnerabilities [5]. Static analysis is known to be conservative in nature and may raise false positives. To name a few, static analysis may find a viable execution path for information flow, which may never be reached in actual program execution; static analysis may find that interesting information is stored in some elements in a database, and thus has to conservatively treat the entire database as sensitive. Upon receiving a discovered vulnerability, the developer has to manually confirm if the reported vulnerability is real. It may also be nontrivial for the (often inexperienced) developer to properly fix the vulnerability and release a patch for it. As a result, these discovered vulnerabilities may not be addressed for long time or not addressed at all, leaving a big time window for attackers to exploit these vulnerabilities. Out of the 16 apps with component hijacking vulnerabilities we tested, only 3 of them were fixed in one year.

To close this window, we aim to automatically generate a patch that is specific to the discovered component hijacking vulnerability. In other words, we would like to automatically generate a *vulnerability-specific* patch on the original program to block the vulnerability as a whole, not just a set of malicious requests that exploit the vulnerability.

While automatic patch generation is fairly new in the context of Android applications, a great deal of research has been done for traditional client and server programs (with and without source code). Many efforts have been made to automatically generate signatures to block bad inputs, by performing symbolic execution and path exploration [6]–[10]. This approach is effective if the vulnerability is triggered purely by the external input and the library functions can be easily modeled in symbolic execution. However, for android applications, due to the asynchronous nature of the program execution, a suc-

NDSS '14, 23-26 February 2014, San Diego, CA, USA

http://dx.doi.org/doi-info-to-be-provided-later

cessful exploitation may depend on not only the external input, but also the system events and API-call return values. Other efforts have been made to automatically generate code patches within the vulnerable programs to mitigate certain kinds of vulnerabilities. To name a few, AutoPaG [11] focused on buffer overflow and general boundary errors; IntPatch [12] addressed integer-overflow-to-buffer-overflow problem; To defeat zero-day worms, Sidiroglou and Keromytis [13] proposed an end-point first-reaction mechanism that tries to automatically patch vulnerable software by identifying and transforming the code surrounding the exploited software flaw; and VSEF [14] monitors and instruments the part of program execution relevant to specific vulnerability, and creates execution-based filters which filter out exploits based on vulnerable program's execution trace, rather than solely upon input string.

In principle, our automatic patch generation technique falls into the second category. However, our technique is very different from these existing techniques, because it requires a new machinery to address this new class of vulnerabilities. The key of our patch generation technique is to place *minimally* required code into the vulnerable program to *accurately* keep track of dangerous information originated from the exposed interfaces and effectively block the attack at the security-sensitive APIs.

To achieve this goal, we first perform static bytecode analysis to identify small but complete *program slices* that lead to the discovered vulnerability. Then we devise several shadowing mechanisms to insert new variables and instructions along the program slices, for the purpose of keeping track of dangerous information at runtime. In the end, we apply a series of optimizations to remove redundant instructions to minimize the footprint of the generated patch. Consequently, the automatically generated patch can be guaranteed to completely disable the vulnerability with minimal impact on runtime performance and usability.

We implement a prototype, *AppSealer*, in 16 thousand lines of Java code, based on the Java bytecode optimization framework Soot [15]. We leverage Soot's capability to perform static dataflow analysis and bytecode instrumentation. We evaluate our tool on 16 real-world Android apps with component hijacking vulnerabilities. Our experiments show that the patched programs run correctly, while the vulnerabilities are effectively mitigated.

Contributions. In summary, this paper has made the following contributions:

- We propose an automatic patch generation technique to defeat component hijacking attacks in Android applications. Its core is to place minimally required code into the vulnerable program to accurately keep track of dangerous information and block the attack right at the sink.
- We implement a prototype system called *AppSealer* in 16 thousand lines of Java code. AppSealer firstly performs static dataflow analysis to generate program slices of dangerous information flows. Then, it automatically patches the slices by devising shadowing mechanisms and inserting new statements. Finally, it applies a series of optimizations to remove redundant patch statements and minimize the patch size.
- We evaluate AppSealer on 16 vulnerable Android apps. The experiments show that our approach is both effective and efficient. We demonstrate that patched apps run correctly, while the vulnerability is effectively mitigated. The size of patched program increases only 15.9% on average and the added runtime overhead is minimal, merely 2% on average.

Paper Organization. The rest of the paper is organized as follows. Section II describes the problem of automatic patch generation for preventing component hijacking attacks, and gives an overview of our patch generation technique. Section III, IV, and V present the three steps of our proposed technique, which are taint slice computation, patch statement placement, and patch optimization. Section VI presents our experimental results over real Android apps. Discussion and related work are presented in Section VII and Section VIII, respectively. Finally, the paper concludes with Section IX.

II. Problem Statement & Approach Overview

In this section, we start with a motivating example to explain the problem that this paper aims to address and provide an overview of our proposed technique.

A. Running Example

Figure 1 presents a synthetic running example in Java source code, which has a component hijacking vulnerability. More concretely, the example class is one of the Activity components in an Android application. It extends an Android Activity and overrides several callbacks including `onCreate()`, `onStart()` and `onDestroy()`.

Upon receiving a triggering Intent, the Android framework creates the Activity and further invokes these callbacks. Once the Activity is created, `onCreate()` retrieves the piggybacked URL string from the triggering Intent. Next, it saves this URL to an instance field `addr` if the resulting string is not null, or uses the `DEFAULT_ADDR` otherwise. When the Activity starts, `onStart()` method acquires the latest location information by calling `getLastKnownLocation()`, and stores it to a static field `location`. Further, `onDestroy()` reads the location object from this static field, encodes the data into a string, encrypts the byte array of the string and sends it to the URL specified by `addr` through a raw socket.

This program is subject to component hijacking attack, because a malicious app may send an Intent to this Activity with a URL specified by the attacker. As a result, this vulnerable app will send the location information to the attacker's URL. In other words, this vulnerability allows the malicious app to retrieve sensitive information without having to declare the related permissions (i.e., `android.permission.ACCESS_FINE_LOCATION` and `android.permission.INTERNET`).

Besides information leakage, a component hijacking attack may happen in the reverse direction. That is, a vulnerable program may allow a malicious app to modify the content of certain sensitive data storage, such as the Contacts database,

```
1  public class VulActivity extends Activity{
2    private String DEFAULT_ADDR = "http://default.url";
3    private byte DEFAULT_KEY = 127;
4
5    private String addr;
6    private static Location location;
7    private byte key;
8
9    /* Entry point of this Activity */
10   public void onCreate(Bundle savedInstanceState){
11     this.key = DEFAULT_KEY;
12
13     this.addr = getIntent().getExtras().getString("url");
14     if(this.addr == null){
15       this.addr = DEFAULT_ADDR;
16     }
17   }
18
19   public void onStart(){
20     VulActivity.location = getLocation();
21   }
22
23   public void onDestroy(){
24     String location =
25       Double.toString(VulActivity.location.getLongitude()) +
       "," + Double.toString(VulActivity.location.getLatitude
       ());
26     byte[] bytes = location.getBytes();
27     for(int i=0; i<bytes.length; i++)
28       bytes[i] = crypt(bytes[i]);
29     String url = this.addr;
30     post(url, bytes);
31   }
32
33   public byte crypt(byte plain){
34     return (byte)(plain ^ key);
35   }
36
37   public Location getLocation(){
38     Location location = null;
39     LocationManager locationManager = (LocationManager)
        getSystemService(Context.LOCATION_SERVICE);
40     location = locationManager.getLastKnownLocation(
        LocationManager.GPS_PROVIDER);
41     return location;
42   }
43
44   public void post(String addr, byte[] bytes){
45     URL url = new URL(addr);
46     HttpURLConnection conn = (HttpURLConnection)url.
        openConnection();
47     ...
48     OutputStream output = conn.getOutputStream();
49     output.write(bytes, 0, bytes.length);
50     ...
51   }
52 }
```

Fig. 1. Java Code for the Running Example

even though the malicious app is not granted with the related permissions.

B. Problem Statement

We anticipate our proposed technique to be deployed as a security service in the Android marketplace, as illustrated in Figure 2. Both the existing apps and the newly submitted apps must go through the vetting process by using static analysis tools like CHEX [5]. If a component hijacking vulnerability is discovered in an app, its developer will be notified, and a patch will be automatically generated to disable the discovered vulnerability. So the vulnerable apps will never reach the end users. This approach wins time for the developer to come up with a more fundamental solution to the discovered security problem. Even if the developer does not have enough skills to fix the problem or is not willing to, the automatically generated

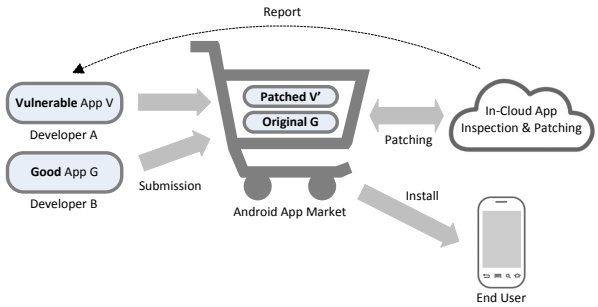

Fig. 2. Deployment of AppSealer

patch can serve as a permanent solution for most cases (if not all).

In addition, it is also possible to deploy our technique with more ad-hoc schemes. For instance, an enterprise can maintain its private app repository and security service too. The enterprise service conducts vetting and necessary patching before an app gets into the internal app pool, and thus employees are protected from vulnerable apps. Alternatively, establishing third-party public services and repositories is also viable and can benefit end users.

Specifically, we would like to achieve the following design goals:

- **No source code access.** Our technique should not require the application source code for patch generation. It is particularly important for Android ecosystem, because the developers only submit the executables (in form of .APK files) to the marketplace.
- **Vulnerability-specific patching.** The generated patch using our technique should effectively disable the vulnerability as a whole, not just a particular set of exploitation instances.
- **Minimal performance overhead.** The extra performance overhead induced by the patch should be minimal, such that the user experience is not affected. In the context of patch generation, it means that the inserted instructions (as the patch) should be minimally necessary to fix the vulnerability.
- **Minimal impact on usability.** The normal operations on the patched app should remain intact. The intervention introduced in the patch should only take place while an attack is about to take into effect.

C. Approach Overview

Figure 3 depicts the workflow of our automatic patch generation technique. It takes the following steps:

(1) **IR Translation.** An Android app generally consists of a Dalvik bytecode executable file, manifest files, native libraries, and other resources. Our patch generation is performed on the Dalvik bytecode program. So the other files remain the same, and will be repackaged into the new app in the last step. To facilitate the subsequent static analysis, code insertion, and code optimization, we first translate Dalvik bytecode into an Intermediate Representation (IR).

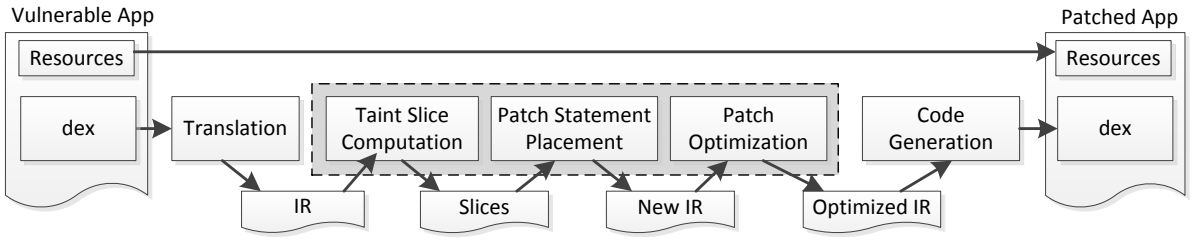

Fig. 3. Architecture of AppSealer

In particular, we first convert the DEX into Java bytecode program using dex2jar [16], and then using Soot [15], translate the Java bytecode into Jimple IR.

(2) **Slice Computation.** On Jimple IR, we perform *flow-sensitive context-sensitive inter-procedural* dataflow analysis to discover component hijacking flows. We track the propagation of certain "tainted" sensitive information from sources like internal data storage and exposed interfaces, and detect if the tainted information propagates into the dangerous data sinks. By performing both forward dataflow analysis and backward slicing, we compute one or more program slices that directly contribute to the dangerous information flow. To distinguish with other kinds of program slices, we call this slice *taint slice*, as it includes only the program statements that are involved in the taint propagation from a source to a sink.

(3) **Patch Statement Placement.** With the guidance of the computed taint slices, we place shadow statements into the IR program. The inserted shadow statements serve as runtime checks to actually keep track of the taint propagation while the Android application is running. In addition, guarding statements are also placed at the sinks to block dangerous information flow right on site.

(4) **Patch Statement Optimization.** We further optimize the inserted patch statements. This is to remove the redundant statements that are inserted from the previous step. As Soot's built-in optimizations do not apply well on these patch statements, we devise three custom optimization techniques to safely manipulate the statements and remove redundant ones. Thereafter, the optimized code is now amenable to the built-in optimizations. Consequently, after going through both custom and built-in optimizations, the added patch statements can be reduced to the minimum, ensuring the best performance of the patched bytecode program.

(5) **Bytecode Generation.** At last, we convert the modified Jimple IR into a new package (.APK file). To do so, we translate the Jimple IR to Java package using Soot, and then re-target Java bytecode to Dalvik bytecode using Android SDK. In the end, we repackage the patched DEX file with old resources and create the new .APK file.

III. Taint Slice Computation

Taking a Jimple IR program and the sources and sinks specified in the security policies as input, our application-wide dataflow analysis will output one or more taint slices for dangerous information flows. Our analysis takes the following steps: 1) we locate the information sources in the IR program, and starting from each source, we perform forward dataflow analysis to generate a taint propagation graph; 2) if a corresponding sink is found in this taint propagation graph, we perform backward dependency analysis from the sink node and generate a taint slice.

A. Forward Dataflow Analysis

For each identified taint source, we perform context-sensitive flow-sensitive forward dataflow analysis to generate a taint propagation graph. This application-wide dataflow analysis leverages intra-procedural def-use chain analysis and call graph analysis, which are provided in Soot. The basic algorithm is fairly standard and similar to other static dataflow analysis systems, such as CHEX [5]. In comparison, our analysis serves as the foundation for automatic patch generation, and thus needs to be conservative and complete.

Basic Algorithm. More concretely, the algorithm proceeds as follows. For each located taint source, we perform def-use chain analysis to identify how this taint propagates in the hosting function. If a function call is identified to be included in this def-use chain, we further perform def-use chain analysis within the callee function. If the callee function is an API, we will apply our pre-defined Android API model. After completing def-use chain analysis for one function, its return value, "this" pointer, some class members, and/or some function parameters, may be identified to be tainted. In this case, we will pass this result back to the caller function, and def-use chain analysis will resume in the context of caller function. Therefore, this analysis process is recursive in nature.

Special Considerations for Android Apps. To ensure completeness in our dataflow analysis, we have several special considerations for analyzing Android applications, due to their object-oriented and event-driven characteristics.

Static field, serving as global variable in Java, can be shared by multiple methods for asynchronous communication. In order not to miss the information flow through shared global region, we conservatively treat all the tainted static fields as new sources. Though this may yield false positives for analysis, our inserted runtime checks will guarantee the correct order of "set" and "get" operations, eliminating the uncertainty during patch generation phase.

Instance field, unlike a static field, is only accessible through one certain instance object. In a sense, an instance

field in Java is analogous to a structure field of a heap object in C. Although it is possible to statically distinguish certain accesses to an instance field of two different instance objects by performing pointer analysis, it is rather difficult for Android applications. For each Android application, a significant portion of the program execution is actually done by the Android framework. The application only serves as a "plug-in" to this framework. The analysis result that is only based on the application code will not be complete and correct. In this work, we adopt a rather simple memory model. That is, any access to an instance field of an object may reflect to all the objects of the same class. Again, we take a simple but conservative analysis approach, and the subsequent patch generation can ensure the correctness of information flow detection. Notice that the primitive wrapper class (e.g., Byte, Integer, etc.) and their combination such as String or Integer array essentially also contain instance fields. However, static analysis on bytecode level does not see their internal fields and always treats them as atomic primitive types. Therefore, this simple memory model does not apply to them and will not cause false positives.

Intent is a unique feature in Android applications for message passing, and thus a possible medium for propagating information. Explicit Intents are used for inter-class communications. The sender class can create an explicit Intent by specifying the name of receiver class, and trigger the receiver with the Intent via for example `startActivity(Intent)` or `startActivityForResult(Intent, int)`. As long as the receiver's name is resolvable, the subsequent calling sequence can be determined statically. Otherwise, we conservatively consider all the potential targets, reading Intent information via `Intent.getData()` or `Intent.getExtras()`, as propagation destinations. Implicit Intents are used to broadcast. The manifest.xml, which specifies the registration of BroadcastReceiver, thus helps construct the relationship between sender and receiver. Senders and listeners of Android components are so far handled manually. It is possible to leverage prior work [17] for better target resolution.

Class inheritance is an object-oriented feature, which allows method overriding. Again, it requires points-to analysis to determine the type of object instance and thus the call target. In practice, we leverage Soot's capability to do call-graph analysis, which performs class hierarchy analysis. We then conservatively go through all the possible call targets for data propagation.

Thread is not unique to Android but needs special considerations. The calling convention for Thread is that, while the caller makes a call to `Thread.start()`, the control is passed over to a corresponding `Thread.run()`. Therefore, we need to consider the start-run pair and handle this case in call-graph analysis.

B. Backward Dependency Analysis

After one or more sinks are identified in the taint propagation graph, we further remove the nodes that can reach none of the sink nodes. To do that, we first compute a reverse graph and then start from each sink node to traverse the graph. After traversal, any unvisited nodes will be removed from the original graph. In this way, we compute the taint slice, which encompasses all the IR statements involved in the dangerous information flow from a given source to a given sink.

C. Running Example

Figure 4 illustrates the taint slices for the running example, after using both forward dataflow analysis and backward dependency analysis. There are two slices in this graph, each one of which represents the data propagation of one source. The left branch describes the taint propagation of "gps" information, which originates from the invocation of `getLastKnownLocation()`. The data is then saved to a static field `location`, before a series of long-to-string and string-to-bytearray conversions in `onDestroy()`. Converted byte array is further passed to `crypt()` for byte-level encryption. The right branch begins with Intent receiving in `onCreate()`. The piggybacked "url" data is thus extracted from the Intent and stored into an instance field `addr`. The two branches converge at `post(String,byte[])`, when both the encrypted byte array and "addr" string are fed into the two parameters, respectively. "addr" string is used to construct an URL, then a connection, and further an OutputStream object, while the byte array serves as the payload. In the end, both sources flow into the sink `OutputStream.write(byte[],int,int)`, which sends the payload to the designated address.

IV. Patch Statement Placement

With taint slices in hand, we have a big picture of how the taint may propagate within the program. The next step is to place patch statements with the guidance of the taint slices. We first briefly introduce our tainting policy. Then we describe how we create shadow variables to record taint status for different kinds of data variables in the program. Finally, we explain how to place shadow instructions to keep track of taint propagation at runtime and block the dangerous tainted information at the sink nodes.

A. Tainting Policy

We adopt a tainting policy similar to the one in TaintDroid [18], striking a balance between precision and performance. However, a key difference between our technique and TaintDroid is that our technique directly modifies the bytecode program to keep track of selected tainted information, whereas TaintDroid modifies the Dalvik Virtual Machine to monitor taint propagation on the execution of every single Dalvik instruction. To be more specific, we aim to monitor dangerous information flow on multiple granularity levels: we perform variable-level tainting for local variables and objects, field-level tainting for instance and static fields, message-level tainting for IPC and file-level tainting for external storage. In other words, we store one shadow variable (taint status) for each single local variable, field, message or file. In addition, we store one shadow variable for entire array, in order to achieve memory efficiency. We also support table lookup, meaning if either an array base `b` or an array index `i` is tainted, and an element is obtained from `b[i]`, this element is also tainted. For structural objects, such as `Vector`, `Hashtable`, etc, we model corresponding APIs for taint propagations. Further discussion is in Section IV-D.

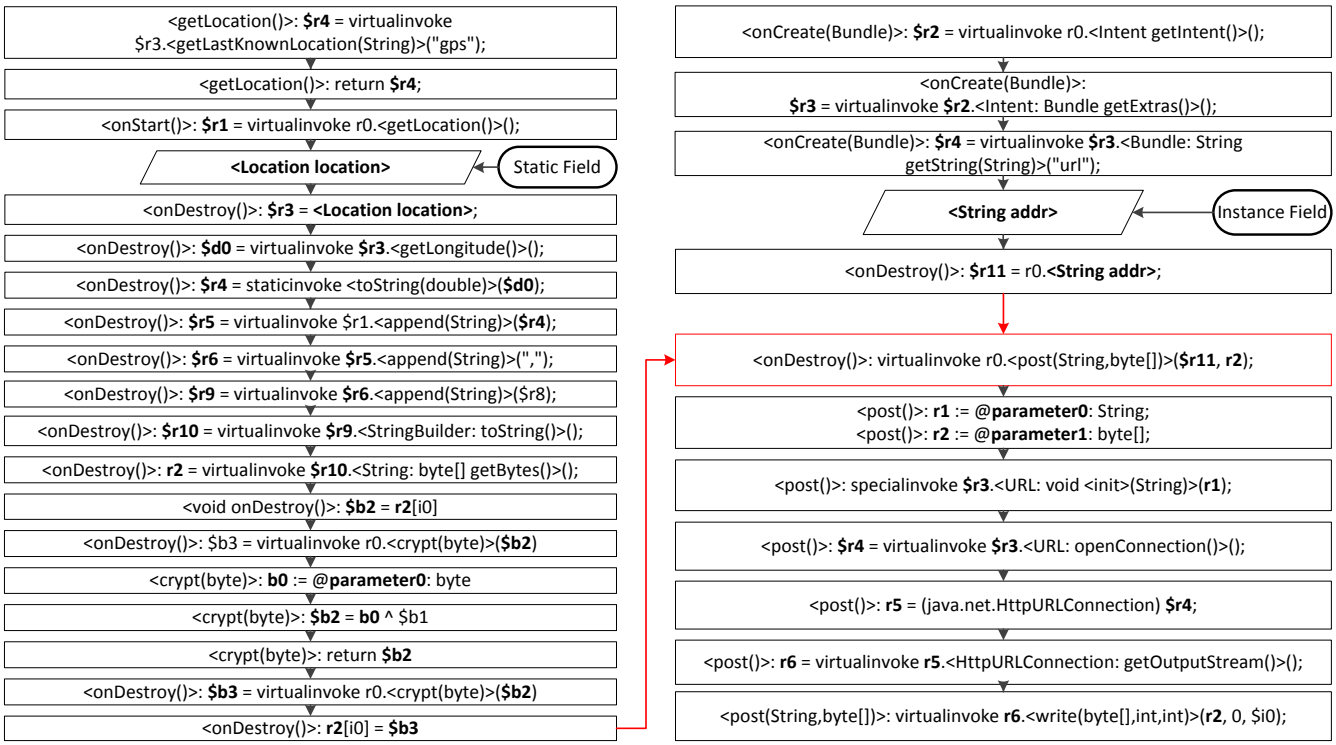

Fig. 4. Taint Slices for the Running Example.

B. Creating Shadow Variables

In order to keep track of taint propagation at runtime, we need to create a shadow for each data entity that may become tainted. In other words, these data entities must appear in the taint slices. The data entities outside the slices do not need to have their shadows, because they are irrelevant to the taint propagation. More specifically, we need to shadow the following types of data entities: 1) local variable; 2) function parameter; 3) function return value; and 4) class data field (including both static field and instance field). They need to be shadowed in different ways.

Local Variables. To shadow local variables, we can create a boolean variable for each local variable within the same function scope. To support taint tracking of multiple sources, we introduce one separate shadow variable for one single taint source. For example, to shadow a local variable $r4$, we create a boolean variable $r4\_s0\_t$ for one taint source and $r4\_s1\_t$ for another. The shadow variable $r4\_s0\_t$ is set to 1 (true) if $r4$ is tainted from the first source, and 0 (false) otherwise. The code snippet below depicts this case.

```
$r4 = virtualinvoke $r3.<android.location.
    LocationManager: android.location.Location
    getLastKnownLocation(java.lang.String)>("gps");
r4_s0_t = 1;
```

Notice that another design option is to always use a collection object (e.g., Vector) to hold and pass all the taints of a variable. We do not take this design option for two reasons: 1) operations on collection objects are more expensive than simple operations on boolean variables; and 2) the compiler normally refuses to optimize collection objects due to their side effects. Using boolean variables as shadow, on the contrary, is more lightweight and amenable to various standard compiler optimization methods such as constant propagation, dead code elimination, etc.

Static/Instance Fields. Static and instance fields are shadowed by adding boolean fields in the class definition. The code snippet below shows how we add a boolean static field to shadow a static field in the class definition. Notice that, fields, that are declared in different children classes of the same parent, are associated with different dedicated shadow fields. This helps to accurately determine the taint status for objects of derived classes.

```
private static android.location.Location location;
public static boolean location_s0_t;
```

Parameters and Return Value. To shadow function parameters and the return value, we have to modify the function prototype to add parameters. These new parameters are used to shadow the function parameters and the return value. However, we cannot directly add boolean parameters like we do for local variables and static and instance fields. This is because primitive boolean parameters are passed *by value* in Java's calling convention. It means that the changes in these boolean parameters will not be reflected back to the caller, resulting in broken taint propagation. To solve this problem, we have to pass an object reference instead of a simple boolean variable as a function parameter. We define a new class `BoolWrapper` for this purpose. This class only contains one boolean instance

field, which holds the taint status, and a default constructor, as shown below. Notice that the Java `Boolean` class cannot serve the same purpose because it is treated the same as primitive boolean type once passed as parameter.

```
public class BoolWrapper extends java.lang.Object{
  public boolean b;
  public void <init>(){
    BoolWrapper r0;
    r0 := @this: BoolWrapper;
    specialinvoke r0.<java.lang.Object:
                  void <init>()>();
    return;
  }
}
```

C. Instrumenting the Source

Right after the taint source statement, we can simply insert an assignment to set the corresponding shadow variable to 1 (true), indicating that the tainted source has just been introduced.

To address component hijacking problem, we consider external Intents as sources. In other words, if an Intent comes from inside the app, we do not set the taint. Consequently, we need to differentiate the internal requests and external ones. Unfortunately, except for IPC through a bound Service or Activities started by `startActivityForResult()`, so far, there exists no way to know the origin of an Intent by Android design. Thus, we have to rewrite existing Intent code in the app so that internal Intent can carry an extra secret integer, and further, at the public interfaces (e.g., Activites, BroadcastReceivers), we distinguish internal and external Intents by checking the secret with an introduced method `isExternalIntent()`. Notice that this method does not exist in the original app, so the attacker can hardly exploit it. Further, the method name can be randomized during rewriting process. To prevent an attacker from forging an internal Intent, we generate a random secret number every time the app starts. Of course, it is always possible to adjust our design provided Intent origin is supported by future Android framework.

D. Instrumenting Taint Propagation

For each node within the taint slices, depending on its statement type, we need to instrument it in different way.

Simple Assignments. For each simple assignment statement in the taint slice, we need to insert a statement thereafter to propagate taint accordingly. For definition statement and unary operation, we insert an definition statement on the shadow variables. For binary operation, we insert a binary OR operation on the shadow variables. Note that in the original statement, not all variables are shadowed, because they do not appear in the def-use chain and will certainly not be tainted. In this case, we replace their shadow variable with a constant 0 (false), indicating that they are always clean. The code snippet below illustrates these cases. Hereafter, we will omit the source label (e.g., s0, s1) for convenience when presenting shadow variable names, if all taint propagations in the snippet are for one single source.

```
r1 = r2; //definition statement
r1_t = r2_t;
r3 = UNOP r4; //unary operation
r3_t = r4_t;
r3 = r4 BINOP r5 //binary operation
r3_t = r4_t | r5_t;
r6 = r7 BINOP r8
r6_t = r7_t | 0; //r8 is not shadowed
```

Function Calls. If the statement to be instrumented is a function call defined within the application, we need to insert statements to bind shadow variables for actual arguments to the shadow variables for the formal parameters. As mentioned earlier, in order to reflect the value changes from the callee back to the caller, we create new instances of `boolWrapper` class and pass the references into the extended parameter list. The code snippet below shows how we instrument an invocation to `crypt` in our running example.

```
$b2_t_w = new BoolWrapper;
specialinvoke $b2_t_w.<BoolWrapper: void <init>()>();
$b2_t_w.<BoolWrapper:boolean b> = $b2_t;

r0_t_w = new BoolWrapper;
specialinvoke r0_t_w.<BoolWrapper: void <init>()>();
r0_t_w.<BoolWrapper:boolean b> = r0_t;

$b3_t_w = new BoolWrapper;
specialinvoke $b3_t_w.<BoolWrapper: void <init>()>();
$b3_t_w.<BoolWrapper:boolean b> = $b3_t;

$b3 = virtualinvoke r0.<VulActivity: byte crypt(
   byte,BoolWrapper,BoolWrapper,BoolWrapper)>
      ($b2, $b2_t_w, r0_t_w, $b3_t_w);

$b3_t = $b3_t_w.<BoolWrapper:boolean b>;
r0_t = r0_t_w.<BoolWrapper:boolean b>;
$b2_t = $b2_t_w.<BoolWrapper:boolean b>;
```

More specifically, we have expanded the parameter list of `crypt` to include three extra `BoolWrapper` references, one for shadowing the first argument, one for shadowing "this" pointer, and the last one for the return value. Before the invocation, the three `BoolWrapper` instances are created and initialized to receive their taint statuses. Then after the invocation, the taint statuses in these `BoolWrapper` instances are passed back to the shadow variables in the caller's context.

We also add instrumentation code in the function body of `crypt` to pass taint status from formal parameters into local variables and in the end pass the new status back, as shown below.

```
public byte crypt(byte, BoolWrapper,
  BoolWrapper, BoolWrapper) {
    r0 := @this: VulActivity;
    b0 := @parameter0: byte;
    w_p0 := @parameter1: BoolWrapper;
    w_t := @parameter2: BoolWrapper;
    w_r := @parameter3: BoolWrapper;
    r0_t = w_t.<BoolWrapper: boolean b>;
    b0_t = w_p0.<BoolWrapper: boolean b>;
    $b2_t = w_r.<BoolWrapper: boolean b>;
    $b1 = r0.<VulActivity: byte key>;
    $b2 = b0 ^ $b1;
    $b2_t = b0_t | 0;
    w_t.<BoolWrapper: boolean b> = r0_t;
    w_p0.<BoolWrapper: boolean b> = b0_t;
    w_r.<BoolWrapper: boolean b> = $b2_t;
    return $b2;
}
```

API Calls. Some statements involve calling Android APIs. Since the method body of an Android API is not included in the program, we have to add instrumentation code to implement the taint propagation logic for that API. These APIs can be generally put into the following categories:

- APIs like `getString()` and `toString()`, have very simple taint propagation logic, always propagating taint from parameters to their return values. Therefore, we generate a default rule, which propagates taint from any of the input parameters to the return value.
- APIs like `android.widget.TextView.setText()` can be modeled as a simple assignment. They propagate taint from one parameter to another reference or "this" reference.
- APIs like `Vector.add(Object)` can be modeled as a binary operation, such that the object is tainted if it is already tainted or the newly added element is tainted.
- APIs like `android.content.ContentValues.put(String key, Byte value)` that operate on (key, value) pairs can have more precise handling. In this case, the elements are stored and accessed according to a "key". To track taint more precisely, we keep a taint status for each key, so the taint for each (key, value) pair is updated individually.

Tracking References. We shadowed local variable in our design, while some of local variables are references to objects. Further, some of them are referencing the same object. If one of the references is tainted, all other references should also be tainted. In order to maintain the internal equivalence of such references, we leverage the common origin of the references as a bridge. To be more specific, once the shadow variable of one reference is set, we set the taint variable of its origin; whenever the taint status of a reference needs to be examined, we evaluate the shadow variable of corresponding origin instead. Simply put, instead of tracking multiple references, we rely on the taint status of the unique origin.

E. Cleaning the Taint

It is not enough to only instrument the statements in the taint slice, because the slice only includes the statements involved in taint propagation. Statements outside the slice may clean a tainted variable in the slice. To properly clean the taint, for each variable appearing in the def-use chain inside the slice, we need to find all its definitions. Then for the definitions outside the slice, we need to insert a statement after that definition to set its shadow variable to 0 (false). The code snippet below shows such a case in our running example.

```
$r4 = virtualinvoke $r3.<android.os.Bundle: java.
    lang.String getString(java.lang.String)>("url");
r4_s1_t = 1;
if $r4 != null goto label0;
$r4 = r0.<VulActivity: java.lang.String DEFAULT_ADDR>
r4_s1_t = 0;

label0:
```

F. Instrumenting the Sink

We instrument the sink to check the taint status of the sink variables. If they are tainted by certain sources, we can raise a pop-up dialog to the user, asking for decision. Our decision dialog offers the user two choices, "restart the app" or "continue running". If the user chooses the first option, we defeat the dangerous execution by immediately restarting the app. Though this solution is not perfect, it is acceptable in the Android environment because application context can be saved to *Shared Preferences* programmatically for restore. It is possible that malicious message also gets saved, but so does the corresponding shadow variable. Therefore, whenever program state is recovered, taint tracking resumes and the attack is still blocked. Again, our goal is not to replace the final patch from developers, but rather to automatically and quickly offer a viable solution, which can serve users' need temporarily.

Below shows how we instrument the raw socket sending as the sink in our running example. In this case, we trigger decision making procedure if both the outgoing data parameter is tainted by source s0, and OutputStream (constructed with URL) is tainted by source s1.

```
public void post(java.lang.String, byte[],
    BoolWrapper, BoolWrapper, BoolWrapper) {
    ....
  if r2_s0_t == 1 && r6_s1_t == 1 goto label0;
  goto label1;
label0:
  staticinvoke <Policy:
      void promptForUserDecision()>();
label1:
  virtualinvoke r6.<java.io.OutputStream:
      void write(byte[],int,int)>(r2, 0, $i0);
  ...
}
```

V. Patch Optimization

A. Optimization Phases

After the patch statements being placed in right positions, we further perform a series of optimizations to reduce the amount of patch statements as much as possible. As an optimization framework, Soot is capable of conducting common optimizations on Java bytecode program. However, directly applying these existing optimization methods will not generate good results, because of the uniqueness in our inserted patch statements. Therefore, we develop three custom optimizations and apply them sequentially on the instrumented code. After going through these custom optimizations, the program is more amenable to Soot's built-in optimizations. Therefore, we apply Soot's built-in optimizations in the end to generate a nearly optimal patch.

Removing Redundant BoolWrappers (O1). As described earlier, we introduce a new class `BoolWrapper` to enable taint propagation across the function boundary. Many BoolWrapper related statements are redundant and can be optimized. However, Soot refuses to optimize these statements, because these statements operate on an instance field in the class, which may cause side effects to the rest of the program execution. In this case, we are confident that redundant BoolWrapper related statements can be removed safely. So, we force to optimize these statements. In particular, to remove

redundant BoolWrapper-related statements, we perform *copy propagation* and *dead assignment elimination* on them within each instrumented function.

Removing Redundant Function Parameters (O2). After removing redundant BoolWrapper related statements, a BoolWrapper parameter in the function prototype may become completely useless. In other words, in the function body, the BoolWrapper parameter is only used in the "Identity" statement (the Jimple IR statement binding formal parameter to corresponding local variable), and is never used later. If this is true, we can remove this parameter from the function prototype. In addition, we need to adjust the remaining Identity statements in the function body, because the indexes of the subsequent parameters have changed. Further, because of the change in the function prototype, all the invocations to this function also need to be adapted accordingly. The optimized `crypt` is then shown below, in which the second parameter is removed and the function size is reduced from 15 to 10 statements.

```
public byte crypt(byte, BoolWrapper, BoolWrapper)
{
  r0 := @this: VulActivity;
  b0 := @parameter0: byte;
  w_p0 := @parameter1: BoolWrapper;
  w_r := @parameter2: BoolWrapper;
  b0_t = w_p0.<BoolWrapper: boolean b>;
  b1 = r0.<VulActivity: byte key>;
  $b2 = b0 ^ $b1;
  b2_t = b0_t | 0;
  w_r.<BoolWrapper: boolean b> = $b2_t;
  return $b2;
}
```

Inlining Instrumentation Code (O3). Code inlining is a standard compiler optimization technique. By inlining the body of a small function into its callers, the function call overhead (including parameter passing, function prologue and epilogue, etc.) can be avoided, and the inlined code can be further specialized under the caller's context. We adopt this idea to further optimize the added instrumentation code. That is, we want to inline the added instrumentation code (i.e., taint logic) from the callee's function body into the caller's context. This is feasible only if the added instrumentation statements are not influenced by the other statements in the callee's function body. More precisely, within the function scope, we compute a backward slice [19] for these instrumentation statements. In order to inline these statements into the caller's context without side effects, the backward slice should not have statements that may cause side effects, such as function calls, static/instance fields, etc. If this is true, it is safe to inline the computed slice into the caller's context, and the callee's function prototype and function body can then be recovered to their original form, meaning that the callee's function body is no longer instrumented.

In our running example, `crypt` is such a case. Its taint logic is simple enough and has no side effect. So the instrumentation statements are removed and inlined into the caller's context. The `crypt` implementation is then recovered in its original form. The code below illustrates that the instrumentation statements from `crypt` have been inlined into its caller `OnDestroy`, so the taint propagation logic in `crypt` is now enforced in the caller's context.

```
$b3 = virtualinvoke r0.
    <VulActivity: byte crypt(byte)>($b2);
tmp21 = $b2_t;
tmp23 = tmp21 | 0;
$b3_t = tmp23
```

Soot's Built-in Optimizations (O4). After these custom optimizations, we can finally apply Soot's built-in optimizations to further reduce the instrumentation overhead. Soot provides a variety of intra-procedural optimizations. In particular, *constant propagation*, *copy propagation*, *dead assignment elimination*, *unreachable code elimination*, and *unused local elimination* play important roles in this optimization process. In the running example, the instrumentation statements, which have been extracted from `crypt` to its caller, are now optimized into only one statement, as shown below.

```
$b3 = virtualinvoke r0.
    <VulActivity: byte crypt(byte)>($b2);
$b3_t = $b2_t | 0;
```

Note that in this example, the inserted statement can be further optimized to `$b3_t = $b2_t`. However, Soot does not optimize arithmetic expressions. To achieve the optimal instrumentation performance, we have implemented this optimization in Soot. In the end, only one definition statement is needed to propagate taint for the `crypt` function.

B. Optimized Patch for Running Example

Figure 5 presents the optimized patch for the running example. For the sake of readability, we present the patch code in Java as a "diff" to the original program, even though this patch is actually generated on the Jimple IR level. Statements with numeric line numbers are from the original program, whereas those with special line number "**P**" are patch statements. The underlines mark either newly introduced code or modified parts from old statements.

We can see that a boolean variable "addr_s0_t" is created to shadow the instance field "addr", and another boolean variable "location_s1_t" is created to shadow the static field "location". Then in `onCreate()`, the shadow variable "addr_s0_t" is set to 1 (tainted) when the Activity is created upon an external Intent. Otherwise it will be set to 0 (untainted). The shadow variable "location_s1_t" is set to 1 inside `onStart()`, after `getLocation()` is called. Note that this initialization is originally placed inside `getLocation()` after Ln.40, and a BoolWrapper is created for the return value of `getLocation()`. After applying the inlining optimization(O3), this assignment is lifted into the caller function `onStart()` and the BoolWrapper variable is also removed.

Due to the optimizations, many patch statements placed for tracking tainted status have been removed. For example, the taint logic in `crypt()` has been lifted up to the body of `onDestroy()` and further optimized there. The tainted values should also be properly cleaned up. For instance, "addr_s0_t" is set to 0 after Ln.15, where "addr" is assigned a constant value, which means that if no "url" is provided in the Intent, the "addr" should not be tainted.

In the method `onDestroy()`, when the information flows through the `post()` method, we wrap the local shadow

```
1  public class VulActivity extends Activity{
     ...
5    private String addr;
P    public boolean addr_s0_t;
6    private static Location location;
P    public static boolean location_s1_t;
     ...
10   public void onCreate(Bundle savedInstanceState){
       ...
13     this.addr=getIntent().getExtras().getString("url");
P      if(isExternalIntent()){
P        this.addr_s0_t = true;
P      }else{
P        this.addr_s0_t = false;
P      }
14     if(this.addr == null){
15       this.addr = DEFAULT_ADDR;
P        this.addr_s0_t = false;
16     }
17   }
18
19   public void onStart(){
20     VulActivity.location = getLocation();
P      VulActivity.location_s1_t = true;
21   }
22
23   public void onDestroy(){
       ...
29     String url = this.addr;
P      BoolWrapper bytes_s1_w = new BoolWrapper();
P      bytes_s1_w.b = VulActivity.location_s1_t;
P      BoolWrapper url_s0_w = new BoolWrapper();
P      url_s0_w.b = this.addr_s0_t;
P      post(url, bytes, url_s0_w, bytes_s1_w);
31   }
     ...
44   public void post(String addr, byte[] bytes,
        BoolWrapper addr_s0_w, BoolWrapper bytes_s1_w){
P      boolean output_s0_t = addr_s0_w.b;
P      boolean bytes_s1_t = bytes_s1_w.b;
       ...
48     OutputStream output = conn.getOutputStream();
P      if(output_s0_t == true && bytes_s1_t == true)
P        promptForUserDecision();
49     output.write(bytes, 0, bytes.length);
50     ...
51   }
52 }
```

Fig. 5. Java Code for the Patched Running Example

variables for corresponding parameters and pass these BoolWrapper objects to new `post()` as additional parameters. In the end, we retrieve the taints in `post()` and check the taint statuses before the critical networking operation is conducted at Ln.49. Consequently, we stop this component hijacking attack right before the dangerous operation takes place.

VI. Experimental Evaluation

To evaluate the efficacy, correctness and efficiency of AppSealer, we conducted experiments on real-world Android applications with component hijacking vulnerabilities and generated patches for them. We first present our experiment setup and in Section VI-A. We then discuss summarized results in Section VI-B, and study several cases in detail in Section VI-C. Next, we verify the effectiveness of these patches in Section VI-D. Finally we measure the runtime performance in Section VI-E.

A. Experiment Setup

We collect 16 vulnerable Android apps, which expose internal capabilities to public interfaces and are subject to exploitation. Table I describes their exposed interfaces, leaked capabilities and possible security threats.

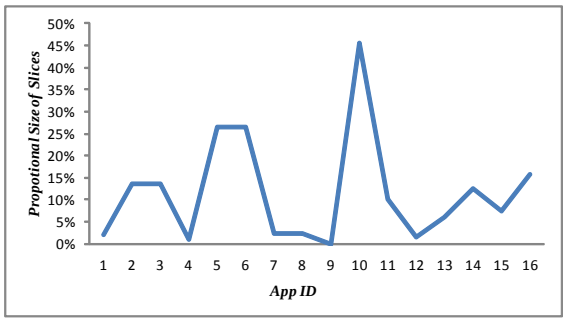

Fig. 6. **Relative Size of Slices in Percentage.**

Most of these vulnerable apps accidentally leave their internal `Activities` open and unguarded. Thus, any `Intent` whose target matches the vulnerable one can launch it. Others carelessly accept any `Intent` data from a public `Service` without input validations. Unauthorized external `Intent` can therefore penetrate the app, through these public interfaces, and exploit its internal capabilities. Such leaked capabilities, including SQLite database query and Internet access, are subject to various security threats. For instance, `Intent` data received at the exposed interface may cause SQL Injection; external Intent data sending to Internet may cause delegation attack.

To detect and mitigate component hijacking vulnerabilities, AppSealer automatically performs analysis and rewriting, and generates patched apps. We conduct the experiment on our test machine, which is equipped with Intel(R) Core(TM) i7 CPU (8M Cache, 2.80GHz) and 8GB of physical memory. The operating system is Ubuntu 11.10 (64bit).

To verify the effectiveness and evaluate runtime performance of our generated patches, we further run them on a real device. Experiments are carried out on Google Nexus S, with Android OS version 4.0.4.

B. Summarized Results

We configure AppSealer to take incoming `Intents` from exposed interfaces as sources, and treat outgoing Internet traffic and internal database access as sinks. A taint slice is then a potential path from the `Intent` receiver to these privileged APIs. We compute the slice for each single vulnerable instance, and conduct a quantitative study on it.

Figure 6 shows the proportional size of the slices, compared to the total size of the application. We can see that most of the taint slices represent a small portion of entire applications, with the average proportion being 11.7%. However, we do observe that for a few applications, the slices take up to 45% of the total size. Some samples (e.g., com.kmshack.BusanBus) are fairly small. Although the total slice size is only up to several thousands Jimple statements, the relative percentage becomes high. Apps like com.cnbc.client operate on incoming `Intent` data in an excessive way, and due to the conservative nature of static analysis, many static and instance fields are involved in the slices.

We also measure the program size on different stages of patch generation and optimizations. We observe that the

TABLE I. OVERVIEW OF VULNERABLE APPS

ID	Package-Version	Exposed Interface	Leaked Capability	Threat Description
1	CN.MyPrivateMessages-52	Activity	Raw Query	SQL Injection
2	com.akbur.mathsworkout-92	Activity	Internet	Delegation Attack
3	com.androidfu.torrents-26	Activity	Selection Query	SQL Injection
4	com.appspot.swisscodemonkeys.paintfx-4	Activity	Internet	Delegation Attack
5	com.cnbc.client-1208	Activity	Selection Query	SQL Injection.
6	com.cnbc.client-1209	Activity	Selection Query	SQL Injection.
7	com.espn.score_center-141	Activity	Internet	Delegation Attack
8	com.espn.score_center-142	Activity	Internet	Delegation Attack
9	com.gmail.traveldevel.android.vlc.app-131	Service	Internet	Delegation Attack
10	com.kmshack.BusanBus-30	Activity	Raw Query	SQL Injection
11	com.utagoe.momentdiary-45	Service	Raw Query	SQL Injection
12	com.yoondesign.colorSticker-8	Activity	Raw Query	SQL Injection
13	fr.pb.tvflash-9	Activity	Selection Query	SQL Injection
14	gov.nasa-5	Activity	Selection Query	SQL Injection
15	hu.tagsoft.ttorrent.lite-15	Service	Internet	Delegation Attack
16	jp.hotpepper.android.beauty.hair-12	Activity	Raw Query	SQL Injection

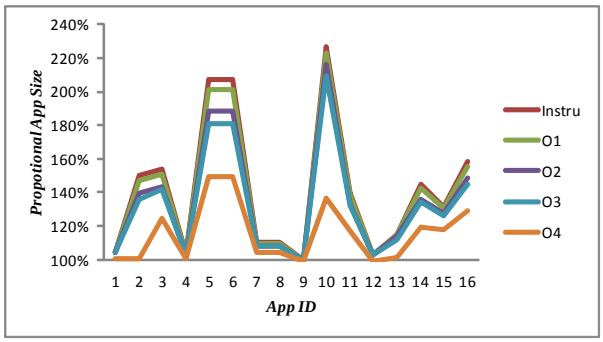

Fig. 7. **Relative App Size in Percentage.** Five curves quantify app sizes at different stages.

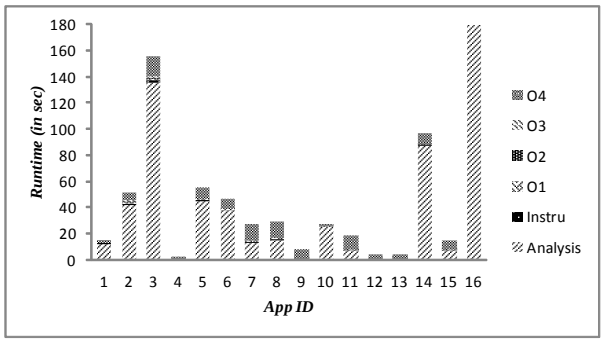

Fig. 8. **Cumulative Runtime.** Runtime for slices computation, patch statement placement and four phases of patch optimizations.

increase of the program size is roughly proportional to the slice size, which is expected. After patch statement placement, the increased size is, on average, 41.6% compared to the original program.

Figure 7 further visualizes the impact of the four optimizations to demonstrate their performance. The five curves on the figure represent the relative sizes of the program, compared to the original app size, on different processing stages, respectively. The top curve is the program size when patch statement placement has been conducted, while the bottom one stands for the app size after all four patch optimizations. We can see that 1) for some of these apps, the increase of program size due to patch statement placement can be big, and up to 130%; and 2) these optimizations are effective, because they significantly reduce the program sizes. In the end, the average size of patch code drops to 15.9%.

C. Detailed Analysis

Here we present detail analysis for these vulnerable apps to discuss the effectiveness and accuracy of our generated patches.

Apps with Simple Exploiting Paths. Some of the apps are vulnerable but exploitation paths are fairly simple. App 4, 7, 8, 9, 11, 12 fall into this category. Upon obtaining Intent data from an open Activity or Service, these apps directly use it either as URL to load a webpage in WebView, conduct a HTTP GET, or as a SQL string to query an internal database. Consequently, the exploitation path is guaranteed to happen every time a malicious Intent reaches the vulnerable interfaces. In this case, simply blocking the exposed interface might be as good as our patching approach.

However, for other apps, a manipulated input may not always cause an actual exploitation.

Apps with Pop-up Dialogs. Some apps ask user for consent before the capable sink API is called. App 2, 3, 5, 6, 10, 13 share this same feature. If the user does not approve further operations, the exploitation will not occur. In this case, blocking at the open interface causes unnecessary interventions. Our approach, on the other hand, disables the vulnerability and requires only necessary mediations.

`com.akbur.mathsworkout` (version code 92) is one of these examples. This app is a puzzle game, which is subject to data pollution and leaks Internet capability. Granted with Internet permission, the app is supposed to send user's "High Score" to a specific URL. However, the Activity to receive "High Score" data is left unguarded. Thus, an malicious app can send a manipulated Intent with a forged "score" to this vulnerable Activity, polluting the latter's instance field. This field is accessed in another thread and the resulting data is sent to Internet, once the thread is started. However, starting this background thread involves human interactions. Unless a "OK" button is clicked in GUI dialog, no exploitation will happen. Our patch correctly addresses this case and only displays the warning when sending thread is about to call the sink API (i.e. `HttpClient.execute()`).

`com.cnbc.client` (version code 1208 and 1209) asks for user's consent in a more straight-forward way. This finance app exposes an Activity interface that can access internal database, and thus is vulnerable to SQL injection attack. The exposed Activity is intended to receive the "name" of a stock, and further save it to or delete it from the "watch list". Malicious Intent can manipulate this "name" and trick the victim app to delete an important one or add an unwanted one. Nevertheless, the deletion requires user's approval. Before deletion, the app explicitly informs the user and asks for decision. Similarly, if the user chooses "Cancel", no harm will be done. Our patch automatically enforces necessary checks but avoids intervention in this scenario. Notice that taint slices of this app take a great portion (27%) of the program, and therefore it is extremely hard to confirm and fix the vulnerability, or further discover aforementioned secure path with pure human effort. In contrast, AppSealer automatically differentiates secure and dangerous paths, and in the meantime manages to significantly reduce the amount of patch statements.

Apps with Selection Views. A similar but more generic case is that apps provide views such as `AdapterView` for selection. The actual exploit only occurs if an item is selected. Apps 1, 14, 16 are of this kind.

`CN.MyPrivateMessages` (version code 52) is a communication app which suffers the SQL injection attack. An vulnerable Activity may save a manipulated Intent data to its instance field during creation. The app then displays an AdapterView for user to select call logs. Only upon selection does the event handler obtain data from the polluted field and use it to perform a "delete" operation in database.

Apps with Multiple Threads. Some samples extensively create new threads during runtime and pass the manipulated input across threads (e.g., app 2, 10, 15). Asynchronous program execution makes it hard for developers or security analysts to reproduce the exploitation and thus to confirm the vulnerability.

D. Effectiveness Evaluation

We run our generated patches on a physical device to verify the effectiveness of our generated patches.

Patch Behavior in Benign Context. Firstly, we test the program execution under normal circumstances. In other words, only internal Intents are delivered to vulnerable public interfaces, and thus they should not cause patch code to raise warnings. Our test combines automatic testing mechanism with interactive manual examination. While automatic testing helps to improve code coverage, manual efforts are made to trigger specific execution paths which lead to component hijacking vulnerability. Our automated testing is conducted with `monkey` [20], which produces random GUI events to feed an Android app. We did not observe any crashes in this test. This demonstrates the feasibility of our approach. Next, we manually explore different parts of the program. In our observation, patched apps act the same as corresponding vulnerable ones, and the original program execution is preserved with no interruption. This shows that our patch does not affect normal usability.

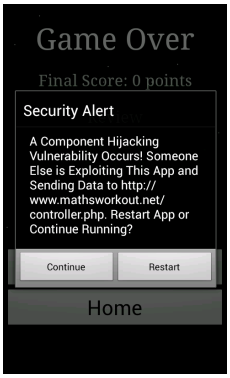

Fig. 9. Hijacking information flow detected during runtime.

Patch Behavior under Attacks. Then, we attempt to launch component hijacking exploits on patched apps to find out whether patch code correctly mitigates the attack by informing user of the risk. To launch the attack, we send custom `Intents` from a testing app. In this custom Intent, a specific vulnerable component is configured to be the receiver and the payload is delicately organized to carry the manipulated inputs. We manage to launch component hijacking attack on 6 vulnerable apps (app 1,2,4,5,6,10). On the vulnerable ones, exploitations succeed; on the patched ones, a warning dialog pops up when an attack is about to happen. Figure 9 illustrates the case when the attack is detected and a warning is raised. It is worth mentioning that launching a successful exploit is nontrivial and time consuming. Thus, it is also hard, if not possible, for inexperienced developers to reproduce the attack. This might explain why many vulnerable apps had not been fixed since they were discovered.

E. Performance Evaluation

We evaluate both offline patching time cost and runtime performance of the patched apps.

Performance of Patch Generation. Figure 8 illustrates the time consumption of patch generation for 16 vulnerable Android apps in our experiment. To be specific, it depicts the execution time of slices computation, patch statement placement, and four phases of patch optimization. We do not show the bytecode conversion times here, because code conversion by dex2jar is usually within several seconds and thus not significant as compared to the other steps. We find that more than 81.3% of the apps (13 out of 16) are processed within 60 seconds and the majority (15 out of 16) can be patched within 3 minute. However, still one app costs excessive time to finish. Slice computation with global dataflow analysis dominates the overall processing time. In comparison, the runtime of patch statement placement and optimization is fairly short.

Runtime Performance. To estimate the runtime overhead of the patched programs, we conduct experiment on Google Nexus S, with Android OS version 4.0.4. We patch the 16 vulnerable apps, run both the original apps and patched ones on the physical device, and compare the runtime performance before and after patching.

We rely on the timestamps of the Android framework debugging information (logcat logs) to compute the

Activity load time as benchmark. The Activity load time is measured from when Android ActivityManager starts an Activity component to the time the Activity thread is displayed. This includes application resolution by ActivityManager, IPC and graphical display. The runtime is thus measured by adding up load time of all Activites that appear in one execution trace. Activity startup is usually fairly fast, and thus this measurement is susceptible to system noise. Noise may even cause negative slowdown. To limit the side-effect of noise, we measure the runtime 10 times and use the average value to calculate runtime overhead for every case.

We first test the runtime performance of all 16 apps under normal circumstances. In other words, no attack is conducted. We start the apps, navigate through several Views, reach the vulnerable interfaces, walk through a series of Views again, and eventually get to the capable components. The program execution thus involves both intensively patched components and other code which is not relevant to component hijacking dataflow. Results show that patched apps usually incur insignificant overhead. The average slowdown of 16 apps is approximately 2%.

Further, we attempt to investigate the worst case. That is, the execution trace involves largely, if not solely, those app components that are heavily patched. To this end, we launch component hijacking attack on the 6 apps (app 1,2,4,5,6,10) to trigger the vulnerable interfaces directly and wait until exploited components start. the overall overhead is still small, with an average of 5% and a standard deviation of 2.8%. The maxmum overhead is relatively higher (9.6%), due to the heavy patching in the onCreate() method of the app (com.kmshack.BusanBus).

VII. Discussion

In this section, we discuss the limitations of our system and possible solutions. We also shed light on future directions.

Soundness of Patch Generation. The soundness of our approach results from that of slice computation, patch statement placement and patch optimizations.

We perform standard static dataflow analysis to compute taint slices. Static analysis, especially on event-driven, object-oriented and asynchronous programs, is known to introduce false positives. However, such false positives can be verified and mitigated during runtime, with our devised shadowing mechanism and inserted patch code.

Our patch statement placement follows the standard taint tracking techniques, which may introduce imprecision. Specifically, our taint policy follows that of TaintDroid. While effective and efficient in a sense, this multi-level tainting is not perfectly accurate in some cases. For instance, one entire file is associated with a single taint. Thus, once a tainted object is saved to a file, the whole file becomes tainted causing over-tainting. Other aggregate structures, such as array, share the same limitation. It is worth noting that improvement of tainting precision is possible. More complex shadowing mechanism (e.g., shadow file, shadow array, etc.) can be devised to precisely track taint propagation in aggregations. However, these mechanisms are more expensive considering runtime cost and memory consumption.

Our optimizations take the same algorithms used in compilers, such as constant propagation, dead code elimination. Thus, by design, original program semantics is still preserved when patch optimization is applied.

In spite of the fact that our approach may cause false positives in theory, we did not observe such cases in practice. Most vulnerable apps do not exercise sophisticated data transfer for Intent propagation, and thus it is safe to patch them with our technique.

Conversion between Dalvik bytecode and Jimple IR. Our patch statement placement and optimizations are performed at Jimple IR level. So we need to convert Dalvik bytecode program into Jimple IR, and after patching, back to Dalvik bytecode. We use dex2jar [16] to translate Dalvik bytecode into Java bytecode and then use Soot [15] to lift Java bytecode to Jimple IR. This translation process is not always successful. Occasionally we encountered that some applications could not be converted. Enck et al. [21] pointed out several challenges in converting Dalvik bytecode into Java source code, including ambiguous cases for type inference, different layouts of constant pool, sophisticated conversion from register-based to stack-based virtual machine and handling unique structures (e.g., try/finally block) in Java.

In comparison, our conversion faces the same, if not less, challenges, because we do not need to lift all the way up to Java source code. We consider these problems to be mainly implementation errors. Indeed, we have identified a few cases that Soot performs overly strict constraint checking. After we patched Soot, the translation problems are greatly reduced. We expect that the conversion failures can be effectively fixed over time.

A complementary implementation option is to engineer a Dalvik bytecode analysis and instrumentation framework, so that operations are directly applied on Dalvik bytecode. Since it avoids conversions between different tools, it could introduce minimal conflicts and failures.

Fully Automatic Defense. For most vulnerable samples in our experiment, we are able to manually verify the component hijacking vulnerabilities. However, due to the object-oriented nature of Android programs, computed taint slices can sometimes become rather huge and sophisticated. Consequently, we were not able to confirm the exploitable paths for some vulnerable apps with human effort, and thus could not reproduce the expected attack. Developers are faced with the same, if not more, challenges, and thus fail to come up with a solution in time. Devising a fully automated mechanism is therefore essential to defend this specific complicated vulnerability.

In principle, our automatic patching approach can still protect these unconfirmed cases, without knowing the real presence of potential vulnerability. That is to say if a vulnerability does exist, AppSealer will disable the actual exploitation on the fly. Otherwise, AppSealer does not interrupt the program execution and thus does not affect usability. With automated patching, users do not have to wait until developers fix the problem.

VIII. Related Work

In this section, we discuss the previous work that is related to automatic patch & signature generation, analysis & mitigation on smartphone privacy issues, vulnerabilities and malware, bytecode rewriting and information-flow control.

Automatic Patch & Signature Generation. Efforts to automatically generate patch for vulnerable program, or signature to filter out malicious input is closely related to our work. AutoPaG [11] automatically analyzes the program source code and identifies the root cause for out-of-bound exploit, and thus generates a fine-grained source code patch to temporarily fix it without any human intervention. IntPatch [12] utilizes classic type theory and dataflow analysis framework to identify potential integer-overflow-to-buffer-overflow vulnerabilities, and then instruments programs with runtime checks. Sidiroglou and Keromytis [13] rely on source code transformations to quickly apply automatically created patches to vulnerable segments of the targeted applications, that are subject to zero-day worms. Newsome et al. [14] propose an execution-based filter which filters out attacks on a specific vulnerability based on the vulnerable program's execution trace. ShieldGen [6] generates a data patch or a vulnerability signature for an unknown vulnerability, given a zero-day attack instance. Razmov and Simon [22] automate the filter generation process based on a simple formal description of a broad class of assumptions about the inputs to an application.

Analysis & Mitigation on Smartphone Privacy Issues, Vulnerabilities and Malware. Many efforts have been made to discover and address emerging threats in smartphones. Privacy leakage has caught attentions and is studied over different platforms, such as iOS, Android and Windows Phone [18], [21], [23], [24]. Mitigation mechanisms are thus proposed. Some work rewrite applications to insert mediation code [24]–[26]; others modify Android framework [27], [28] or operating system [29] to enforce privacy policies. Studies are also carried out to prevent attacks on application vulnerabilities, such as privilege escalation [30], [31], permission re-delegation [3], capability leaks [2], content leaks and pollution [4] and component hijacking [5]. Malware causes enormous damage to smartphone users and thus is a significant threat. Efforts are made to detect unknown malware [32], analyze and understand malware and its evolution [33]–[36] and mitigate malware impact [37].

Bytecode Rewriting. Our approach requires rewriting existing bytecode program, thus is related to prior work with bytecode rewriting techniques. The Privacy Blocker application [26] performs static analysis of application binaries to identify and selectively replace requests for sensitive data with hard-coded shadow data. I-ARM-Droid [38] rewrites Dalvik bytecode to interpose on all the API invocations and enforce the desired security policies. Aurasium [25] repackages Android apps to sandbox important native APIs so as to monitor security and privacy violations. Livshits and Jung [24] implement a graph-theoretic algorithm to place mediation prompts into bytecode program and thus protect resource access. In comparison, our work rewrites the bytecode program in a more extensive way. Inserted patch statements are able to monitor and control specific dataflow in the rewritten app.

Information Flow Control. In effect, our patch statements exercise information-flow control (IFC) during runtime. IFC has been studied on different contexts. Chandra and Franz [39] implement an information flow framework for Java virtual machine which combines static analysis to capture implicit flows. JFlow [40] extends the Java language and adds statically-checked information flow annotations. Jia et al. [41] proposes a component-level runtime enforcement system for Android apps. duPro [42] is an efficient user-space information flow control framework, which adopts software-based fault isolation to isolate protection domains within the same process. Zeng et al. [43] introduces an IRM-implementation framework at a compiler intermediate-representation (IR) level. In constrast, we take static rewriting approach, which requires no support from developers and no modification to runtime. In addition, we focus on a specific vulnerability and enforce control on solely relevant information flow.

IX. Conclusion

We developed a technique to automatically generate patch for Android applications with component hijacking vulnerability. Given a vulnerable Android app, we first perform static bytecode analysis to identify small but complete program slices that lead to the discovered vulnerability. Then we devise several shadowing mechanisms to insert new variables and instructions along the program slices, for the purpose of keeping track of dangerous information at runtime. To further improve performance, we apply a series of optimizations to remove redundant instructions to minimize the footprint of the generated patch. Our evaluation on 16 real-world vulnerable Android applications demonstrates that AppSealer can effectively track and mitigate component hijacking vulnerabilities. Moreover, after going through a series of optimizations, the patch code only represents a small portion (15.9% on average) of the entire program. In addition, the runtime overhead introduced by AppSealer is also minimal, merely 2% on average.

Acknowledgment

We would like to thank anonymous reviewers for their comments. This research was supported in part by NSF Grant #1018217, NSF Grant #1054605 and McAfee Inc. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] 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)*, October 2011.

[2] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[3] 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 Security Symposium*, August 2011.

[4] Y. Zhou and X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[5] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting Android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[6] W. Cui, M. Peinado, and H. J. Wang, "ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing," in *Proceedings of 2007 IEEE Symposium on Security and Privacy (Oakland'07)*, May 2007.

[7] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards Automatic Generation of Vulnerability-Based Signatures," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (Oakland'06)*, May 2006.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," in *Proceedings of the twentieth ACM Symposium on Systems and Operating Systems Principles (SOSP'05)*, October 2005.

[9] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing Software by Blocking Bad Input," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

[10] J. Caballero, Z. Liang, Poosankam, and D. Song, "Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, September 2009.

[11] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, March 2007.

[12] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time," in *Proceedings of the 15th European Conference on Research in Computer Security (ESORICS'10)*, September 2010.

[13] S. Sidiroglou and A. D. Keromytis, "Countering Network Worms Through Automatic Patch Generation," *IEEE Security and Privacy*, vol. 3, no. 6, pp. 41–49, Nov. 2005.

[14] J. Newsome, D. Brumley, and D. Song, "Vulnerability-specific execution filtering for exploit prevention on commodity software," in *Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS'06)*, February 2006.

[15] "Soot: a java optimization framework," http://www.sable.mcgill.ca/soot/.

[16] "dex2jar," http://code.google.com/p/dex2jar/.

[17] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. 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*, August 2013.

[18] 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 Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[19] M. Weiser, "Program Slicing," in *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, March 1981.

[20] "Ui/application exerciser monkey," http://developer.android.com/tools/help/monkey.html.

[21] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th Usenix Security Symposium*, August 2011.

[22] V. Razmov and D. Simon, "Practical Automated Filter Generation to Explicitly Enforce Implicit Input Assumptions," in *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC'01)*, December 2001.

[23] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, February 2011.

[24] B. Livshits and J. Jung, "Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications," in *Proceedings of the 22th Usenix Security Symposium*, August 2013.

[25] R. Xu, H. Sadi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of the 21th Usenix Security Symposium*, August 2012.

[26] "Privacy blocker," http://privacytools.xeudoxus.com/.

[27] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These Aren't The Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, October 2011.

[28] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST'11)*, June 2011.

[29] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile'11)*, March 2011.

[30] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards Taming Privilege-Escalation Attacks on Android," in *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[31] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight Provenance for Smart Phone Operating Systems," in *Proceedings of the 20th Usenix Security Symposium*, August 2011.

[32] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[33] L.-K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[34] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland'12)*, May 2012.

[35] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection," in *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services (MobiSys'12)*, June 2012.

[36] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks," in *Proceedings of the 8th ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'13)*, May 2013.

[37] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, November 2009.

[38] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications," in *Proceedings of the Mobile Security Technologies Workshop (MoST'12)*, May 2012.

[39] D. Chandra and M. Franz, "Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*, December 2007.

[40] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," in *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, January 1999.

[41] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract)," in *Proceedings of 18th European Symposium on Research in Computer Security (ESORICS'13)*, September 2013.

[42] B. Niu and G. Tan, "Efficient User-Space Information Flow Control," in *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS'13)*, May 2013.

[43] B. Zeng, G. Tan, and U. Erlingsson, "Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors," in *Proceedings of the 22th Usenix Security Symposium*, August 2013.