Enhancing Android Security Through App Splitting

Drew Davidson[1(✉)], Vaibhav Rastogi[2], Mihai Christodorescu[3], and Somesh Jha[1,2]

[1] Tala Security, 200 Brown Road, Fremont, CA 94539, USA
drew@talasecurity.io

[2] University of Wisconsin-Madison, 1210 W Dayton Street, Madison, WI 53706, USA
{vrastogi,jha}@cs.wisc.edu

[3] Visa Research, 385 Sherman Avenue, Palo Alto, CA 94306, USA
mihai.christodorescu@visa.com

Abstract. The Android operating system provides a rich security model that specifies over 100 distinct permissions. Before performing a sensitive operation, an app must obtain the corresponding permission through a request to the user. Unfortunately, an app is treated as an opaque, monolithic security principal, which is granted or denied permission as a whole. This blunts the effectiveness of the permissions model. Even the recent enhancements in Android do not account for the interactions between multiple permissions or for multiple uses of a single permission for disparate functionality.

We describe app splitting, a technique that partitions a monolithic Android app into a number of collaborating *minion* apps. This technique exposes information flows inside an application to OS-level mediation mechanisms to allow more expressive security and privacy policies. We implement app splitting in a tool called AppSaw. We describe a method for automatically selecting code partitions that isolate permission uses to distinct minion apps, and use existing security mechanisms to mediate the flow of privileged data. Our partitioning strategy based on vertex multicuts ensures that the minion apps are created efficiently. In our experiments, AppSaw was effective at splitting real-world apps, and incurred a low average performance overhead of 3%.

Keywords: Security · Android · Privilege separation · Permissions

1 Introduction

Smartphones have emerged as ubiquitous computing devices accompanied by unique challenges to security and privacy. Through pervasive access, users present troves of personal data to these devices, both by manual interaction and through numerous sensors onboard the device. The misuse of such data can cause significant harm to a user's privacy. Thus, an important goal of a mobile operating system (OS), such as Android, is to mediate the access that applications (apps) of diverse provenance and trust levels have to this data. A guiding principle in designing mediation mechanisms is the *principle of least privilege* (PLP), which states that a principal, e.g., an app, should be granted no more permissions than necessary to fulfill its intended purpose.

Existing approaches fall short of PLP. Legacy Android versions present the user with a list of all permissions requested by an app at installation time, with no enforceable explanation of purpose, while iOS and recent Android versions present permission requests dynamically while the app runs and when it needs them, in the hope that the UI context hints to the purpose of the permission request. In both cases permissions are granted once and are always available to the app for any purpose, least privilege remains an unachieved goal. Specifically, current mobile OS permission models have two problems:

- **Monolithic apps:** Permissions are granted to an app as a whole: there is no way to approve a permission for one purpose while denying it for another in the same app. A particular case is where application code comes from different sources, e.g., an app including ads and social media integration. A user may want the GPS to be accessible for navigation but not for advertising in a maps app. Previous work has emphasized the importance of isolating these entities in different principals [25].
- **Opaque flows:** Users have no visibility how an app uses its permissions. For instance, permissions cannot help distinguish between a contacts manager app that accesses Internet to show ads and a spyware that leaks contacts to the Internet.

Previous work has attempted to addresses these problems by identifying undesirable information flows through static or dynamic taint analysis [7,12]. Static analysis does not provide any way for users to determine if a flow is actually occurring at runtime. Dynamic taint analysis, on the other hand, has significant runtime overhead. There is also work to rewrite the Android permissions model entirely [14]. However, such approaches require updates to the OS as well as ways developers program apps.

We solve the problems arising due to monolithicity of apps and opaqueness of flows with a technique called *app splitting*. App splitting works by partitioning an app into a number of smaller, collaborating apps called *minions*. Minion apps contain a portion of the original app representing an action that the user can mediate. *Splitting the application into smaller pieces converts sensitive code and data flows from intra-app (invisible to the user and to the OS) to inter-app (visible to the user and the OS for mediation and access-control purposes).* We have implemented AppSaw, which accepts an Android app and a simple, user-defined policy and performs app splitting on the given app. It provides the relevant instrumentation to allow the created minions to communicate with each other via OS-level interprocess communication (IPC) so that they can together provide the functionality provided by the original app while restricting unwanted flows.

Our paper makes the following contributions:

– We formalize app splitting as the problem of finding graph partitions and show how various classes of security policies map to app-splitting strategies. Underlying app splitting is a notion of fine-grained, flow-based permission addressing the entanglement problem.
– We introduce a tool, AppSaw, for performing automatic, optimal app splitting of Android apps based on a specified security policy. AppSaw addresses the monolithic app problem by naturally generalizing the existing work on isolating advertising from the core functionality of an app [25,26].
– We demonstrate experimentally that AppSaw is practical, supports a variety of app types (from book readers to translation apps to social networking tools), and incurs low overhead: operations that use permissions incur a low overhead of less than 3% and the total runtime of the app does not experience any measurable slowdown.

Given that AppSaw works by retrofitting apps, it does not need support of Android OS developers as well as app developers. It is thus amenable to a range of deployment models. AppSaw comes with a number of scripts that ease the task of using it as well as the apps produced by it. Savvy users could thus develop their own policies and use the tool directly. More practically, however, we envision AppSaw to find a unique spot among other mobile app management (MAM) technologies developed as enterprise solutions [1]. An MAM provider could offer AppSaw as a part of their suite to enterprises, where an IT administrator would be able to use it to enforce custom flow policies on existing apps.

The remainder of the paper is structured as follows. Section 2 discusses the problem and provides an overview of our approach. In Sect. 3 we detail our technique for choosing program points at which to split a portion of an app into a minion. Sections 4 and 5 discuss the technical details of how AppSaw preserves app functionality across minions, allowing minion apps to collaborate. In Sect. 6, we evaluate how applications split with AppSaw perform against their monolithic counterparts. We review related work in Sect. 7. Section 8 discusses limitations. We conclude in Sect. 9 with directions for future work.

2 Overview

In this section, we first motivate the need for fine-grained permission controls with an example. We subsequently present our policies, our approach to implement the policies, the challenges involved, and how our approach can be deployed in practice.

Motivating Example: To illustrate the permission problems identified in Sect. 1, we present a running example app, `NetDialer`, that demonstrates the challenges users face in the current Android ecosystem. While simple, this app is representative of many similar apps and requires a set of commonly used permissions. `NetDialer` is an enhanced contact manager app. It allows users to scroll

```
// Flow contacts to the phone
public void makeCall(){
  long id = getLong(CONTACT_ID_INDEX);
  String key = getString(CONTACT_KEY_INDEX);
  Uri cUri = Contacts.getLookupUri(id, key); // P0
  number = getNumberFromContact(mContactUri);
  String action = Intent.ACTION_CALL;
  Uri asUri = Uri.parse(number);
  Intent callIntent = new Intent(action, asUri);
  startActivity(callIntent); // P1
}

// Flow contacts to the network
public void backupContacts(){
  long id = getLong(CONTACT_ID_INDEX);
  String key = getString(CONTACT_KEY_INDEX);
  Uri cUri = Contacts.getLookupUri(id, key) // P2
  number = getNumberFromContact(mContactUri);
  Uri numberUri = Uri.parse(number);
  URL url = new URL(baseURL + numberUri);
  URLConnection conn;
  conn = url.openConnection(); // P3
  conn.connect();
}

// Pull information from the network
public byte[] weatherScreen(){
  URL url = new URL(urlContactIcon + strurl);
  Object content = url.getContent(); // P4
  InputStream is = (InputStream) content
  byte[] buffer = new byte[8192];
  ByteArrayOutputStream bkg;
  int bytesRead;
  bkg = new ByteArrayOutputStream();
  while ((bytesRead = is.read(buffer)) != -1) {
         bkg.write(buffer, 0, bytesRead);
  }
  return bkg.toByteArray();
}
```

Fig. 1. Snippet of code from `NetDialer` demonstrating limitations of the Android permission model. The methods that are shown here use an overlapping set of permissions in different ways that are indistinguishable to the user

through the list of contacts maintained by the operating system and place a phone call to a selected contact. The app also allows the user to access auxiliary information from within the app, such as the day's weather forecast.

There are three functions of `NetDialer` that use permissions, as shown in Fig. 1. These functions illustrate different ways in which the same permissions can be used. The `makeCall` method uses the `READ_CONTACTS` permission to collect contact information at program point P_0 which is used to place a phone call using `CALL_PHONE` permission at P_1. The `backupContacts` method also uses `READ_CONTACTS`, at P_2. The data flows to P_3 which uses the `INTERNET` permission to leak contacts to the network. The `weatherScreen` method also uses `INTERNET` to download weather information from the network at P_4, which it returns. The app can execute all three of the above methods by declaring the use of permissions `READ_CONTACTS`, `CALL_PHONE`, and `INTERNET` in its manifest.

Policies: By installing `NetDialer`, the user grants unconditional permission to the app to read from the contact list and send data to the network, as it does in `backupContacts`. Android does not provide any way to determine existence of and control such a flow in the program. Nor can the user completely shut off network access to the app as it would preclude downloading weather information.

Consider a policy in which a user wants to ensure that their contact information is never leaked to the network. Such mediation policies are expressed as a list of instruction pairs $\langle s, t \rangle$, where s is an instruction that is a source of sensitive information and t is a sink instruction, each such pair written as $s \overset{!}{\rightsquigarrow} t$. The user's policy for `NetDialer` can be expressed as `READ_CONTACTS` $\overset{!}{\rightsquigarrow}$ `INTERNET`. This policy requires that *every* flow from an instruction that uses `READ_CONTACTS` to one that uses `INTERNET` be mediated.[1]

Approach Overview and Challenges: Since Android uses an app as the fundamental security principal, we implement these policies by partitioning an app into multiple sub-apps or minions that are granted permissions individually. IPC across these minions is mediated according to the policies. In the case of `NetDialer`, AppSaw can isolate each of the program points P_0, P_1, P_2, and P_3 into distinct minions, and replace their invocations with inter-process communication code to retrieve the original behavior of these program points. In the policy example above, since P_2, and P_3 are placed in different minions, the flow between them can be mediated.

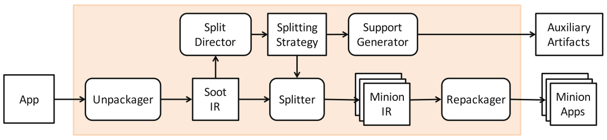


Fig. 2. AppSaw workflow. Rounded components indicate code modules; rectangles indicate artifacts.

AppSaw needs to address two fundamental problems: (1) Given a flow policy, how should the code be split into minions? An important consideration here is to satisfy the policy while keeping the performance impact low. (2) How should the minion apps communicate to collaboratively maintain the functionality of the original app? We address the first challenge by developing formalisms around identifying potential split points using vertex multicuts over the app's control flow graph. For the second challenge we develop a solution rewrites app code to make use of Android IPC to communicate among minions.

The workflow of AppSaw is described in Fig. 2. The input app is first unpackaged with its code converted to the Jimple intermediate representation (IR) using

[1] We can specify any permission pair as a policy and AppSaw ensures that any flow between these permissions will cross a minion boundary. It is up to the user to decide if separating these permissions is meaningful.

dexplar [9]. Jimple is a native IR for the Soot framework [27] and is consumed by our Split Director and Splitter modules. The Split Director module converts user policies to a splitting strategy, which identifies at which points the app should be split. The Splitter module partitions the IR into minions, which are packaged back into native Android apps using the Soot dex compiler and Apktool [6] while restraining the permissions requested by these minion apps. In addition, the Support Generator module uses the splitting strategy to provide artifacts, such as rules that allow the OS to mediate communication among minions. In the next three sections we discuss the workings of the three interesting components: the Split Director, the Splitter, and the Support Generator.

3 Splitting Strategies

In this section, we discuss our algorithm for building the splitting strategies described in Sect. 2. First, we formalize the problem in terms of a *labeled control-flow graph (LCFG)* of an application. Let $G = (V, E, L)$ be a LCFG of an A, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $L : V \rightarrow \mathcal{P}$ is a function that labels each node with an element (called *permission*) from a set $\mathcal{P}$. We assume that there is a special element $\perp \in \mathcal{P}$ which represents the *null* permission. Intuitively $L(v) = \perp$ means that the statement corresponding to node $v \in V$ does not need any special permissions. Formally, the problem, which we call the *permission separation problem (PSP)* can be defined as follows:

Problem 1. Given a LCFG $G = (V, E, L)$ and a relation $X \subseteq \mathcal{P} \times \mathcal{P}$. The problem is to find a partition $\Pi = \{V_1, V_2, \cdots, V_k\}$ of V, which satisfies the following condition: for all pairs of nodes (v_1, v_2), if $(L(v_1), L(v_2)) \in X$, then v_1 and v_2 are in different sets of the partition Π.

Given a partition $\Pi = \{V_1, V_2, \cdots, V_k\}$, we can create k applications $\{A_1, \cdots, A_k\}$ such that A_i consists of all instructions corresponding to nodes in V_i. We call applications A_i $(1 \leq i \leq k)$ *minions.* A naive algorithm for solving PSP creates a partition as follows: each $v \in V$ such that $L(v) \neq \perp$ is put in its own set and there is a set that consists of all nodes w such that $L(w) = \perp$. We call this naive algorithm *permission isolation splitting.* Of course, our naive algorithm can create a lot of minions. Our goal is to construct as few minions as possible and also minimize data transfer between the minions. Next we present our algorithm to accomplish these goals.

Our Algorithm: Our algorithm works in two stages: (1) We compute a vertex multicut using dominators and post-dominators (defined below). (2) We use the vertex multicut found in step (1) to find a solution to the PSP. The two steps of the algorithm are described below.

(Step 1) An Algorithm for Finding Vertex Multicuts. The *vertex multicut problem (VMP)* is defined below.

Problem 2. We are given a graph $G = (V, E)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges and a collection of k pairs of vertices

$H = \{(s_1, t_1), \cdots, (s_k, t_k)\}$. The problem is to remove the minimum number of vertices $V' \subseteq V$ such that in the resulting graph there is no path from s_i to t_i for $1 \leq i \leq k$. In other words, every path from s_i to t_i (for $1 \leq i \leq k$) goes through at least one vertex in V'. This problem is called the *directed graph vertex multicut problem (VMP).*

Although the problem of computing optimal vertex and edge multicuts is NP-complete, there exist approximation algorithms to solve these problems [3,15]. However, these existing algorithms ignore the structure of the program (i.e., the CFGs resulting from an application have a very special structure). We present an algorithm that exploits the structure of the program and can therefore be used to take into account domain-specific considerations (see the discussion towards the end of this section). Specifically, we present here an algorithm for computing vertex multicuts that is based on the concept of dominators and post-dominators. Recall that dominators and post-dominators are used to find *control dependencies* in programs [19] and there are efficient algorithms to compute dominators and post-dominators [16]. We note that our algorithm is *not* provably polynomial time, but can account for program structure. However, incorporating program structure in other algorithms is an interesting avenue for future research.

Assume that we are given a graph $G = (V, E)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges and a collection of k pairs of vertices $H = \{(s_1, t_1), \cdots, (s_k, t_k)\}$. We present an algorithm that demonstrates that an algorithm for finding hitting sets can be used to find a vertex multicut. With each pair (s_i, t_i) we associate a set M_i with the following property: for all $v \in M_i$, every path from s_i to t_i passes through v. The collection of k pairs of vertices $H = \{(s_1, t_1), \cdots, (s_k, t_k)\}$ corresponds to a collection of sets $\mathcal{M} = \{M_1, \cdots, M_k\}$. A *hitting set* Z for $\mathcal{M}$ is a set such that $Z \cap M_i \neq \emptyset$ (for all $1 \leq i \leq k$). Therefore, a hitting set for $\mathcal{M}$ corresponds to a vertex multicut.

The problem now is to associate with a pair of nodes (s, t) a set M such that all vertices in M appear on all paths from s to t. For this, we use the concept of dominators and post-dominators. We assume that the graph $G = (V, E)$ has two distinguished vertices $r \in V$ (called the *start node*) and $e \in V$ (called the *exit node*) such that every vertex in V is reachable from r and e is reachable from every vertex in V.

Dominators and Post-dominators: A vertex v dominates w (denoted as v dom w) iff every path from r to w passes through v. A vertex z post-dominates w (denoted as z pdom w) iff every path from w to e passes through z. The set of dominators and post-dominators of a vertex w are denoted by DOM(w) and PDOM(w), respectively.

Proposition 1. Let (s, t) be a pair of vertices and let $M = \text{DOM}(t) \cap \text{PDOM}(s)$. Every path from s to t passes through every vertex in M.

Proof: Consider a path π from s to t. Since s is reachable from the start node $r \in V$, π can be extended to a path from r to t. Similarly, since the exit node e

is reachable from the node $t \in V$, π can be extended to a path from s to e. Let π_f be a path from r to e that is the extension of path π from s to t. Consider a vertex $z \in M$. Let π_f^r be the fragment of π_f from r to t. Since $z \in \text{DOM}(t)$, z lies on the path fragment π_f^r. Similarly, since $z \in \text{PDOM}(s)$, z lies on the path fragment π_f^e of π_f from s to e. This proves that z lies on the path from s to t. Since z was an arbitrary node in M, the result follows. $\square$

Based on the proposition given above we can formulate an algorithm for finding a vertex multicut, which is based on the dominator and post-dominator structure of the control-flow graph (see Fig. 3).

```
**Input:** A graph $G = (V, E, r, e)$,
set $H$ of $k$ pairs of vertices $\{(s_1, t_1), \cdots, (s_k, t_k)\}$.
Compute $M_i$ (for $1 \leq i \leq k$) as $\text{DOM}(t_i) \cap \text{PDOM}(s_i)$
Compute hitting set $Z$ for the collection $\{M_1, \cdots, M_k\}$
**Output:** The hitting set $Z$.
```

Fig. 3. Finding vertex multicuts using dominators, post-dominators, and hitting sets.

(Step 2) From Vertex Multicut to Partitions

An algorithm for solving VMP can be used to solve PSP. The description is as follows:

- Assume that we are given an *application* A whose LCFG is $G = (V, E, L)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $L : V \rightarrow \mathcal{P}$ is a labeling function. We are also given a relation $X \subseteq \mathcal{P} \times \mathcal{P}$.
- Relation X corresponds to a collection $H(X)$ of pairs of vertices as follows: $(v_1, v_2) \in H(X)$ iff $(L(v_1), L(v_2)) \in X$.
- Now consider the graph $G_1 = (V, E)$ and set $H(X)$. Let $V' \subseteq V$ be a vertex cut for G_1 and $H(X)$. Let G' be the graph obtained from G_1 where outgoing edges from all vertices in V' have been removed. G' induces a partition as shown in Fig. 4. It is not hard to see that the partition $\mathcal{P} = \{V_1, V_2, \cdots, V_k, V_{k+1}\}$ solves the corresponding PSP problem, i.e., for all pairs of nodes (v_1, v_2) such that $(L_A(v_1), L_A(v_2)) \in X$, then v_1 and v_2 are in different sets of the partition $\mathcal{P}$.

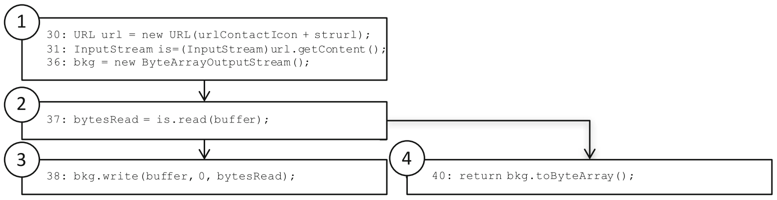
Discussion: Our algorithm based on dominators and post-dominators allows a designer to have control over how the split is performed. First, we introduce some notation from [16]. Vertex v is the *immediate dominator* of w (denoted by v i-dom w), if v dominates w and every other dominator of w dominates v. Similarly, vertex v is the *immediate post-dominator* of w (denoted by v i-pdom w), if v post-dominates w and every other post-dominator of w post-dominates v. The relation i-dom and i-pdom form a directed rooted tree. Intuitively, a node "higher" up in the tree corresponding to i-dom represents a statement closer to the entry point of an application (similar intuition can be applied to the tree corresponding to the relation i-pdom). Therefore, if there are two nodes v and w in a set in the collection Z (see Fig. 3) and v is an ancestor of w in the tree

```
Inputs: A collection $H(X) = \{(s_1, t_1), \cdots, (s_k, t_k)\}$,
a graph $G'$ such that there is no path from $s_i$ to $t_i$
(for all $1 \leq i \leq k$).
Consider the sequence $s_1, s_2, \cdots, s_k$ of source vertices and let $G_0 = G'$.
For $1 \leq i \leq k$,
    define $V_i$ as all vertices reachable from $s_i$ in $G_{i-1}$.
    To construct $G_i$, remove all vertices in $V_i$ from $G_{i-1}$.
Let $V_{k+1}$ be the set $V \setminus \bigcup_{i=1}^{k} V_k$.
Output: $\{V_1, V_2, \cdots, V_{k+1}\}$
```

Fig. 4. Algorithm for creating partitions from vertex multicuts. Removing a vertex v also means we remove all edges of the form (w, v) and (v, w).

corresponding to i-dom, then v can be preferred over w while constructing the hitting set for Z. Similarly, a designer can specify other conditions. For example, some vertices from the collection Z can be eliminated based on certain conditions before computing the hitting set. Examples of some of these conditions are given below (there are several other domain-specific possibilities).

- Eliminate vertices that correspond to statements in some specific functions (e.g., belonging to a third-party library).
- Eliminate vertices from Z that belong to loops (having the split point in the loop might result in expensive IPC calls because of marshaling and unmarshaling of arguments).



```
30: URL url = new URL(urlContactIcon + strurl);
31: InputStream is=(InputStream)url.getContent();
36: bkg = new ByteArrayOutputStream();
```

```
37: bytesRead = is.read(buffer);
```

```
38: bkg.write(buffer, 0, bytesRead);
```

```
40: return bkg.toByteArray();
```

Fig. 5. Control flow graph of the `NetDialer` function `WeatherScreen`.

Figure 5 shows the CFG for the `NetDialer` code of Fig. 1, with line numbers preserved from the original figure. Although simplistic, this example shows the importance of picking good split points: consider the naive solution of including blocks 1 and 2 for a partition: since the variable buffer is live across the boundary from block 2 to 3, making the call to the minion corresponding to the partition will require copying the entire buffer. While this behavior might be acceptable for a single transfer to a minion, but altering the minions in this way causes a transfer on every iteration of the loop that begins on line 32. Thus, the heuristics presented above places blocks 1, 2, and 3 into the minion.

The above example highlights the need to minimize the amount of data that needs to marshaled. This would require us to solve a weighted version of PSP (the PSP version above is unweighted) where the weights on edges correspond to the marshaled data amount and the solution is obtained by minimizing the weights on the multicut. We will investigate this problem in future work.

Split Director Implementation: Our formalism above assumes $L : V \rightarrow \mathcal{P}$, so a node is only mapped to a single permission. In practice, a node may require a finite set of permissions. Extending the strategies support this behavior is trivial and our implementation does not have limitations on the labeling function L. Furthermore, L relies on a mapping from Android API to the requisite permissions. We integrate with PScout [8], Stowaway [13], and Flowdroid [7] to use the mapping produced by these tools.

4 Minion App Generation

The previous section dealt with identifying split points in an application. In this section we discuss the actual refactoring of an app into multiple collaborative minion apps. This level of app rewriting is non-trivial to implement, and relies on a number of unique circumstances that are fortunately present for Android apps. In particular, there needs to be an efficient IPC mechanism that allows for objects to be quickly moved from one app to another. We begin with a a background on Android IPC and then discuss the details on minion app generation.

4.1 Android IPC Background

Android apps comprise a number of collaborating components, which may communicate via IPC. Android provides a fast IPC mechanism called *binder*. The binder model uses a client/server architecture where the client (the app core in the case of AppSaw) requests a connection to a service. Upon success, the client is offered an interface to the service through which it can make calls to it as though it were an object in the local process. Th binder API ensures that the proper steps are taken to translate the call into IPC invocations.

One of the ways that binder achieves fast IPC is by allowing custom marshaling and unmarshaling of objects that are passed through IPC. When an object is passed, a process called *Parcelization* invokes the `WriteToParcel` method, which gives developers the opportunity to choose how to pack the object. The class must also provide a static (non-instance) member that implements the Parcelable. Creator interface, which supplies a `createFromParcel` method, that is invokes in the called component to unpack the parcel. Parcelization stands in contrast to Java's Serialization, in which objects implement a marker interface, but are wholly serialized, with the exception of `transient` fields. Although Serialization is still implemented in Android, Parcelization was specifically designed to meet IPC performance requirements that Serialization lacks [24].

4.2 App Splitting Implementation

Our primary concern here is to ensure that executing minion apps preserves the effect of executing the same code in the original, monolithic app. Because the splitting strategies that our tool uses yield regions in which the entry block to the region dominates the exit of the block, and the exit postdominates the entry, there is no need to worry about relocating control flow transfers. However, AppSaw needs to ensure that data flow that passes through a minion is preserved. While AppSaw could simply instrument an app to copy out all variables that the minion defines, and copy in all variables that the minion uses, doing so would unnecessarily copy data that is not dead. Instead, we perform a simple reaching definition analysis to only copy uses that are live at the beginning minion region and only copy definitions that are live at the end of the minion.

Parceling Objects: Binder IPC provides a means to transfer use and def values across minions but it comes with the additional constraint that objects must be *parcelable*. Unfortunately, it is unlikely that every object that must be transferred to a minion will implement this interface. Thus, AppSaw is faced with a number of challenges:

1. The minion may use or define a non-parcelable user-defined class. In this case, AppSaw has significant power, because it can completely rewrite the definition of the class, adding the requisite `writeToParcel` and `createFromParcel` implementations. A particularly ambitious implementation could even rewrite the class such that it only packs and unpacks the fields used by the minion region, though we leave this as an optimization for future work.
2. The minion may use or define a non-parcelable system-defined class. Here, the previous solution does not apply, because the implementation of the class is not part of the app itself, and therefore cannot be rewritten. A potential solution is to define a parcelable subclass of this class but this fails as the subclass does not have access to the superclass's private fields needed for parcelization or when the class is final.
3. The runtime class of an object may not match its declared class. While the class must be a subclass of the declared type, it may not be an exact match. This is important, because it complicates any attempt to statically insert code to build proxies for the object.

We have developed a solution to the above problem that we call *Parcel Wrappers*. At a high level, a Parcel Wrapper wraps a single object. When the object needs to be transferred to a minion, the Parcel Wrapper uses reflection to decompose the object into its parcelable components. When the component is returned, the Parcel Wrapper uses reflection to get the new values of the object's fields and update it accordingly. Our solution based upon reflection is affected by none of the above problems of private field access (private fields are accessible via reflection) or final classes.

Minion Lifecycle: Android apps operate according to a lifecycle: the system invoke callbacks into the app for events such as creating, starting, pausing, and destroying components. To ensure that the services exposed by minion apps are available to the app as it is launched, AppSaw calls the `bindService` function to binds each minion field at the entry point lifecycle functions of the app. In response to the `bindService` call, the system will invoke the `onServiceConnected` callback of the app (added by AppSaw, as appropriate), where the service is connected. While this works well for most apps, consider the case in which an entrypoint function itself requires the use of a minion: the `onServiceConnected` callback cannot be invoked until the app returns from the function, but the function uses minions initialized in `onServiceConnected`. We resolve this paradox in the following way: For each such callback C that uses a minion, we create a new function C'. The body of C is replaced with code that checks if the service is available, and if so calls C'. If it is not, C raises a global flag f_C indicating that a call to C is necessary, and assigns all arguments of C to newly-added instance fields $args_C$ of the app. When the service is available, `onServiceConnected` checks f_C, and calls C' with $args_C$.

In effect, this modification of the app results in services being connected before any entrypoint function of the app takes effect. Note that because the C' are all called at entry to `onServiceConnected`, any user code in `onServiceConnected` will not run until the entrypoint functions are run. This handles any dependencies in the original body of `onServiceConnected` to data touched in C.

Discussion: AppSaw generates multiple minion apps from one app. This would unnecessarily clutter the device with apps. Our implementation hides the minion apps from the user interface (e.g., from the launcher screen) but not having the minion apps subscribe to the `android.intent.category.LAUNCHER` intent, which is necessary for being launched from the UI.

Our app splitting implementation can transfer variables across minion boundaries. However, the case of transfer through persistent resources, such as files, is different. Since Android isolates persistent resources of apps by default, our tool does not automatically support transfer of data in these ways. Adding support for persistent resources is our future work (Table 1).

Table 1. Characteristics of the apps in the Utility Test Suite.

Display name	Package name	Original instruction count	# Permissions
Bible	com.sirma.mobile.bible.android	575472	16
CNN	com.cnn.mobile.android.phone	440211	13
Duolingo	com.duolingo	562020	14
Facebook	com.facebook.katana	272534	17
Job search	com.indeed.android.jobsearch	153580	8
Original borders	com.aviary.feather.plugins.borders-free	54	0
MyFitnessPal	com.myfitnesspal.android	859176	13
Pandora	com.pandora.android	296037	13
Pocket Manga	com.supo.pocket.mangareader	150417	4
Ringtone maker	com.herman.ringtone	135487	9
Zillow	com.zillow.android.zillowmap	788544	16

5 Minion Support Artifacts

The key advantage of app splitting is that opaque, internal functionality of a single app can be exposed to app-centric security mechanisms. However, app splitting increases the complexity of managing the functionality of the app. In addition to the core functionality, AppSaw provides support artifacts both for the purpose of enforcing security as well as improving usability. This section describes the support artifacts produced by AppSaw.

```
<rules>
  <activity block="true" log="false">
    <component-filter name="com.netdialer.core/" />
  </activity>
  <broadcast block="true" log="true">
    <intent-filter>
      <action name="com.netdialer.minion1" />
    </intent-filter>
  </broadcast>
</rules>
```

Fig. 6. A sample Intent Firewall ruleset blocking broadcast intents from the NetDialer core app to a minion.

Install Script: The most immediate drawback of app splitting is that a user needs to manage multiple apps instead of one. To address this concern, the policy generator outputs a script that can be invoked to install minion apps *en masse*. This script can be incorporated into the user flow according to the deployment model: in an MAM offering, the script will be launched directly by the MAM interface.

Table 2. Minion partitioning for the DroidBench programs. For each of the flows detected by the underlying FlowDroid analysis, AppSaw correctly separates the permission into its own minion.

Category	Number of apps	Average number of minions
Aliasing	1	0.0
AndroidSpecific	12	1.25
ArraysAndLists	7	1.57
Callbacks	15	1.47
EmulatorDetection	3	2.33
FieldAndObjectSensitivity	7	2.14
GeneralJava	23	1.65
ImplicitFlows	4	0.0
InterComponentCommunication	18	1.0
Lifecycle	17	1.35
Reflection	4	2.0
Threading	5	1.2

Interaction Graph: AppSaw produces an interaction graph where a node is included for each minion and there is an edge between two minions if a flow may occur between them. This graph allows the user to visualize permission flows and iteratively develop better policies.

Intent Firewall Rules: The goal of AppSaw is to allow OS-level mediation of flows inside the app. We leverage *Intent Firewall*, an integrated feature of the Android framework that mediates intents (and hence binder IPC) based on certain rules. AppSaw generates Intent Firewall rules to enable this mediation across minions at runtime. Figure 6 shows example rules for `NetDialer`, to enforce the policy that the core app may not send any intent to minion1, which corresponds to GPS use (Table 2).

6 Evaluation

This section empirically evaluates the utility, security, and performance of AppSaw. We ask the following three key questions.

1. *Utility.* Can apps rewritten with AppSaw continue to provide their desired functionality?
2. *Security.* Are apps rewritten with AppSaw prohibited from performing disallowed functionality?
3. *Performance.* Does the rewriting process of AppSaw introduce manageable overhead on apps?

Experimental Highlights: AppSaw preserves the desired functionality of apps while blocking the disallowed functionality in the apps examined. Split apps exhibit an average runtime overhead of 3% over their original variants and use a trivial amount of additional disk space.

6.1 Methodology

We build our experimental setup around three distinct suites of Android apps that evaluate respectively the utility, security, and performance aspects of AppSaw. This allows us to evaluate AppSaw in depth on a small number of apps and also perform tests in breadth on a larger number of apps. We now describe each of these suites in detail.

Utility Test Suite – Google Play Dynamic Sample: To evaluate utility, we obtained a cross-section of real-world apps, we built a test suite by randomly selecting top apps, each with at least a million downloads, from the Google Play store. To ensure that app splitting did not cause any errors or changes in the functionality of the app, we executed the two app variants (original and split) on the same sequence of user interactions, and then manually inspected the resulting user interface (UI) states. This allowed us to observe any differences in functionality caused by the AppSaw transformation.

Previous work has noted that testing Android apps is challenging [5,28]. Apps are interactive and have significant functionality triggered via GUI. In the absence of a practical, comprehensive app testing approach, one must employ either human-generated or semi-random event sequences. Both of these options have disadvantages. Scalability is a problem for human users, while semi-random input sequences can be shallow in the functionality explored [21]. As the purpose of this test suite is to determine whether the user experiences the same behavior from an app in both its original and split versions, we chose human-generated inputs. This necessarily limited the number of apps in the Utility Test Suite.

We manually interacted with the original apps and recorded all interactions using [30]. We then used this tool to faithfully replay these interactions on the rewritten apps. For each app in the Utility Test Suite we collected two interaction traces, each sufficiently long to perform a logical task in the corresponding app. On average a logical task took 5 s to complete.

Security Test Suite – DroidBench: This consists of 119 applications from DroidBench 2, a testing suite originally developed as part of FlowDroid [7], for the purpose of evaluating static analyses for information-flow tracking. As such, apps in DroidBench are crafted by authors from a variety of institutions to provide challenging data flows. In our experiments, we use the information flows statically reported by FlowDroid as input to AppSaw, with the goal of splitting the DroidBench apps such that all of the FlowDroid-discovered flows are mediated by a cross-minion IPC.

Performance Test Suite – IPC Microbenchmarks: The primary overhead introduced by AppSaw is due to the cost of each IPC call when data is transferred back and worth among minions. While the cumulative cost of AppSaw IPC over the lifetime of an app execution is low enough to be invisible to the user, mostly because apps typically do not cross cut boundaries frequently, this does not give a precise estimate of the overhead. To isolate the overhead, we crafted a number of apps that only create permission-to-permission flows and do nothing else. These apps do not represent the behavior and performance of a useful app, but provide a worst-case analysis and thus an upper bound on the performance impact of app splitting.

The apps forming the Performance Test Suite are fully deterministic, do not depend on user input or any environment settings, and behave as follows.

- *Direct Flow:* In this app, we measure the performance penalty of splitting the most common form of permission leak on Android, a flow of a device-specific identifier (IMEI) to the network. This microbenchmark measures the cost of a single IPC call to a minion. Our measurements are averaged over 12 runs and compare the original app versus the split app.
- *Loop Flow:* Here, we modify the direct flow experiment such that source data is repeatedly queried in a loop. Once the loop is finished, the results of the final query leaked to the network. The purpose of this microbenchmark is to determine if the mechanism can properly identify good candidate regions for including in a single minion: AppSaw should include the entire loop in the

minion and perform a single transfer, rather than performing a per-iteration transfer.
- *Large Flow:* This app tests the overhead of moving a large amount of data into the minion. While a typical app will only include source and the small amount of data that touches it, in this app we ensure that a large, user-defined class is tainted with source data. We measured the overhead of transferring progressively larger classes.

We performed all app rewriting and evaluation on an Intel Core 2 Quad CPU at 3.00 GHz. This machine used the Android emulator configured to emulate an Intel x86 device with 1 GB RAM running Android 4.1.2 with host-GPU acceleration. Our implementation of AppSaw is based on Soot 2.5.0 and consists of 20K lines of new or modified code.

6.2 Experimental Results

Utility Findings: Our experiments with the Utility Test Suite did not show any change in behavior in the split apps compared to their original variants. A number of statistics about each app are shown in Fig. 1.

Security Findings: For each of the 119 apps in DroidBench, AppSaw successfully exposed each flow (as discovered by FlowDroid) to OS-level mediation. As shown in Fig. 1, the number of minions varied between apps, with some apps having no unwanted information flows (and thus no minions in the split version), while others having two or more.

Performance Findings: Our findings are summarized in Table 3. The performance measurements for the *Direct Flow* microbenchmark are shown in Fig. 7. The transfer of IMEI, a small string, across minions is inexpensive and is dominated by the cost of IPC itself, thus incurring only 3% overhead. For the *Loop Flow* microbenchmark, we observed that the loop is rightly placed in a single minion and so the overhead is unsurprisingly similar to that of the *Direct Flow* microbenchmark. Finally, the *Large Flow* microbenchmark showed that the runtime overhead scaled with the size of the data being transferred to the minions, as captured by Fig. 8.

Table 3. Results of the Performance Test Suite. Split apps incur small overhead when transferring low to moderate amounts of data, but can experience slowdowns with larger data transfers.

Microbenchmark	Overhead	Number of permissions	Number of minions	Instructions per minion
Direct flow	3%	2	1	18
Loop flow	3%	2	1	27
Large flow	21%	2	1	12

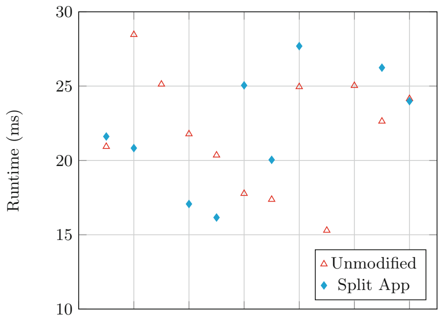


Fig. 7. Runtime measurements of the *Direct Flow* microbenchmark. This scatterplot shows the runtime of minion transfer on 12 runs of the split app compared to the runtime of the same code on 12 runs of the unmodified version of the same app.

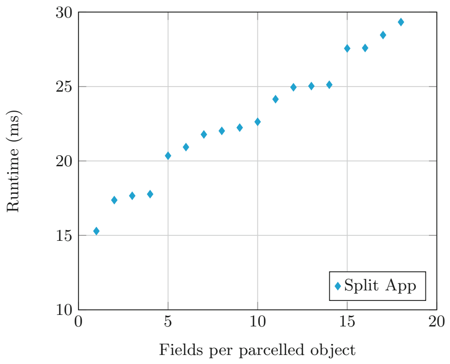


Fig. 8. Runtime measurements of the *Large Flow* microbenchmark. The minion-IPC overhead increases with the amount of data transferred.

7 Related Work

Program Partitioning: Several systems exist to partition applications. We note that in general, the Android permissions model allows our system to bootstrap simple policies without the cooperation of the developer which is a benefit of our domain that much previous work did not have available. Chong *et al.* propose a system for splitting web applications [17]. Unlike this work AppSaw does not require the placement of annotations, nor does it require source code, or any effort on the part of the app's developer. However, granting such conditions could potentially improve the performance of AppSaw, though it would require a different threat model. Zheng *et al.* propose a system to partition applications across multiple, mutually distrusting hosts [31]. This scheme also requires annotations to the program source. Program partitioning has also been studied for web apps. Akhawe et al. [4] propose non-invasive techniques to partition real-world web apps but their partitioning is manual. Calzavara et al. [10] detect privilege escalation vulnerabilities in web apps; they do not provide means for privilege separation though. Luo and Rezk [18] automatically partition web mashups to provide greater security. While their work is similar to ours, our work fits cleanly in the domain of Android apps and is backed by different formalisms.

Advertising Isolation: There has been a line of work in isolating advertising from the rest of an application, such as [25,26]. Special-case operation of AppSaw [20,25,26]. The most closely related work to our own is AdSplit, which automatically rewrites an app to use an isolated advertising library [25]. Unlike AppSaw, AdSplit uses Quire [11], which requires modifications to Android itself. AppSaw runs on an unmodified Android device, and thus has no presence on the actual device. Although the approach of AppSaw is similar to AdSplit, the goals of the systems are different and both users may benefit from using both tools in parallel.

Android Rewriting: Aurasium [29] rewrites apps in order to specify policies by hooking system calls, and employing a runtime security monitor. Unlike AppSaw, Aurasium does not separate apps into multiple pieces, and does not give the user the chance to control permissions in any way.

Android Isolation: Previous work has explored advantages of application level isolation. In particular, Roesner *et al.* developed a modified version of the Android OS, called LayerCake, that allows entities of different trust levels to be embedded into a single app [23]. At a high level, the goal of this work is similar to that of app splitting in that it sharpens the boundaries of security principals. However, the approach taken by Roesner *et al.* differs from our own in that it requires action on the part of the developers to employ new programming practices to comply with a new version of Android. In constrast, our work focuses on enabling existing security mechanisms to work within the current Android security model. Furthermore, the goal of LayerCake is to enable trusted UI components, whereas the goal of our work is to isolate fine-grained functionality of apps.

8 Limitations

AppSaw is effective at splitting apps to expose intra-app flows to OS mediation. However, it has some limitations. Importantly, AppSaw inherits all the limitations of static analysis. It may not work correctly in the presence of reflection, dynamic code loading, and code obfuscation. While lexical obfuscation (renaming identifiers, most Android apps are lexically obfuscated only) is not a problem for AppSaw, commercial Android packers [2] thwart static analysis. However, in the context of enterprise deployment with MAM, it is reasonable for developers to agree on not using such obfuscations in return for the enterprise deploying their apps on a large scale.

Certain Android permissions work on content providers, which are often specified as URL strings. AppSaw will work correctly if our strings analysis can decode these strings (which may not be possible in case of obfuscation). Moreover, since we perform Java-only static analysis, native code is not yet supported. Handling implicit flows efficiently is also an open area of research and we do not currently handle implicit permission flows during splitting. Finally, handling side channel flows is beyond the scope of this work.

9 Conclusion and Future Work

Modern operating systems, such as Android, provide mechanisms for fine-grained control using permissions how apps can use resources. In this paper we used app splitting to extend this control to permission flows. Specifically, we developed app rewriting techniques to split an app into multiple collaborative apps to expose its internal data flows for OS-level mediation. Our evaluation shows that our tool, AppSaw, is practical: it works on real-world apps, fulfills its security purpose, and the rewritten apps have low runtime overhead.

There are several avenues for future for in app splitting and AppSaw. Currently, we replicate all fields of an object when it is passed across minions. We can aggressively optimize this by not replicating fields which have not been initialized or will not be used after transfer. Another level of optimization may be achieved by solving the weighted version of PSP to identify split points with low data transfers. Another avenue is to introduce more fine-grained policies. Currently, our policies are defined in terms of Android permissions, which are known to be coarse-grained [22]. We can in the future introduce syntax and mechanisms for policies that are specified at the finer granularity of API functions. Another direction worth looking at is providing app developers with an SDK to enable easily developing collaborative, split apps that provide permission flow guarantees. By involving the developers we can overcome some of the limitations inherent with retrofitting such as those mentioned in Sect. 8.

References

1. Magic Quadrant for Enterprise Mobility Management Suites. https://www.gartner.com/doc/reprints?id=1-390IMNG&ct=160608&st=sb. Accessed 14 Apr 2015
2. We can still crack you! https://www.blackhat.com/docs/asia-15/materials/asia-15-Park-We-Can-Still-Crack-You-General-Unpacking-Method-For-Android-Packer-No-Root.pdf. Accessed 21 June 2017
3. Agarwal, A., Alon, N., Charikar, M.: Improved approximation for directed cut problems. In: STOC (2007)
4. Akhawe, D., Saxena, P., Song, D.: Privilege separation in HTML5 applications. In: Proceedings of the 21st USENIX Conference on Security Symposium, p. 23. USENIX Association (2012)
5. Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M.: Using GUI ripping for automated testing of android applications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pp. 258–261. ACM, New York (2012). https://doi.org/10.1145/2351676.2351717. ISBN 978-1-4503-1204-2
6. APKTool. Android apktool: a tool for Reengineering Android APK files. code.google.com/p/android-apktool/. Accessed 11 Nov 2015
7. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E. Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, June 2014
8. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, pp. 217–228. ACM, New York (2012). https://doi.org/10.1145/2382196.2382222. ISBN 978-1-4503-1651-4

9. Bartel, A., Klein, J., Monperrus, M., Le Traon, Y.: Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In: Proceedings of the International Workshop on the State Of the Art in Java Program Analysis (SOAP 2012) (2012). https://doi.org/10.1145/2259051.2259056. http://hal.archives-ouvertes.fr/hal-00697421/PDF/article.pdf
10. Calzavara, S., Bugliesi, M., Crafa, S., Steffinlongo, E.: Fine-grained detection of privilege escalation attacks on browser extensions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 510–534. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_21
11. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: QUIRE: lightweight provenance for smart phone operating systems. In: 20th USENIX Security Symposium, San Francisco, CA, August 2011
12. Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI 2010, pp. 1–6. USENIX Association, Berkeley (2010). http://dl.acm.org/citation.cfm?id=1924943.1924971
13. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 627–638. ACM, New York (2011). https://doi.org/10.1145/2046707.2046779. ISBN 978-1-4503-0948-6
14. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing android's permission system. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 1–18. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33167-1_1
15. Gupta, A.: Improved results for directed multicut. In: SODA (2003)
16. Lengauer, T., Tarjan, R.: A fast algorithm for finding dominators in a flowgraph. ACM TOPLAS **1**(1), 121–141 (1997)
17. Livshits, B., Chong, S.: Towards fully automatic placement of security sanitizers and declassifiers. In: Proceedings of the Symposium on Principles of Programming Languages (POPL), January 2013
18. Luo, Z., Rezk, T.: Mashic compiler: mashup sandboxing based on inter-frame communication. In: 2012 IEEE 25th Computer Security Foundations Symposium (CSF), pp. 157–170. IEEE (2012)
19. Muchnick, S.S.: Advanced Compiler Design and Implementation. Academic Press, Cambridge (1997)
20. Pearce, P., Felt, A.P., Nunez, G., Wagner, D.: AdDroid: privilege separation for applications and advertisers in android. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2012, pp. 71–72. ACM, New York (2012). https://doi.org/10.1145/2414456.2414498. ISBN 978-1-4503-1648-4
21. Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M.: Benefits and limitations of automated software testing: systematic literature review and practitioner survey. In: 7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, 2–3 June 2012, pp. 36–42 (2012). https://doi.org/10.1109/IWAST.2012.6228988
22. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing android sources and sinks. In: NDSS (2014)

23. Roesner, F., Kohno, T.: Securing embedded user interfaces: android and beyond. In: Proceedings of the 22nd USENIX Conference on Security, SEC 2013, pp. 97–112. USENIX Association, Berkeley (2013). http://dl.acm.org/citation.cfm?id=2534766.2534776. ISBN 978-1-931971-03-4
24. Sayed, S., Hashim, Y., Komatineni, S., MacLean, D.: Pro Android 2. Apress, New York (2010)
25. Shekhar, S., Dietz, M., Wallach, D.S.: AdSplit: separating smartphone advertising from applications. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security 2012, p. 28. USENIX Association, Berkeley (2012). http://dl.acm.org/citation.cfm?id=2362793.2362821
26. Toubiana, V., Narayanan, A., Boneh, D., Nissenbaum, H., Barocas, S.: Adnostic: privacy preserving targeted advertising. In: NDSS (2010)
27. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot-a Java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, p. 13. IBM Press (1999)
28. Wasserman, A.I.: Software engineering issues for mobile application development. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010, pp. 397–400. ACM, New York (2010). https://doi.org/10.1145/1882362.1882443. ISBN 978-1-4503-0427-6
29. Xu, R., Saïdi, H., Anderson, R.: Aurasium: practical policy enforcement for android applications. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security 2012, p. 27. USENIX Association, Berkeley (2012). http://dl.acm.org/citation.cfm?id=2362793.2362820
30. Zadgaonkar, H.: Robotium Automated Testing for Android. Packt Publishing, Birmingham (2013). ISBN 178216801X, 9781782168010
31. Zheng, L., Chong, S., Myers, A.C., Zdancewic, S.: Using replication and partitioning to build secure distributed systems. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy, SP 2003, p. 236. IEEE Computer Society, Washington (2003). http://dl.acm.org/citation.cfm?id=829515.830549. ISBN 0-7695-1940-7