Technical Report 73

Understanding the Implemented Access Control Policy of Android System Services with Slicing and Extended Static Checking

Tanveer Mustafa
Karsten Sohr

TZI, Universität Bremen

TZI-Bericht Nr. 73
2014

Universität Bremen

TZI-Berichte

Herausgeber:
Technologie-Zentrum Informatik und Informationstechnik
Universität Bremen
Am Fallturm 1
28359 Bremen
Telefon: +49 421 218 94090
Fax: +49 421 218 94095
E-Mail: hq@tzi.de
http://www.tzi.de

ISSN 1613-3773

Understanding the Implemented Access Control Policy of Android System Services with Slicing and Extended Static Checking

***Abstract*—Android is one of the major smartphone platforms today. One reason for this success is that many interesting applications are made available through Google Play. The increasing functionality, however, entails new risks. To defend against attacks, Android provides a sophisticated security architecture based on permissions which must be granted to applications at installation time. Since the Android source code is publicly available, the security community has the chance to assess the security mechanisms of Android. Due to its large code body, a completely manual code review is tedious and hence tool support for this task is desirable. As a first step in this direction, we propose to extract the implemented access control policy from the code for Android system services with the help of program slicing. After this abstraction phase, we analyze the extracted policy against the documentation. For this purpose, we use the Java Modeling Language (JML) in conjunction with extended static checking. We applied this approach to core system services of Android 4.0.3, although we support all Android versions. In our experiments, we identified inconsistencies between the documentation and the implementation.**

I. Introduction

Smartphones become more and more popular, with a steadily growing market share among mobile phones. One reason for the success of smartphones lies in the possibility to conveniently download small programs (called "apps") from application repositories, such as Apple's App Store or Google Play. In particular, the Android platform has become one of the major smartphone operating systems. The fact that smartphones store and process sensitive information, such as location data, passwords for applications, or device IDs, makes them attractive for attackers. Specifically, malicious apps can harm the end user's phone by trying to exploit security holes on the phone or tricking the user to grant excessive permissions.

Due to the fact that Android is open source and supports rich inter-process communication means, it has attracted much attention in the security research community [10], [22], [13], [4], [33], [8], [11], [19], [18], [28]. Most of the current works focus on enhancing the security functionality of the Android platform or statically/dynamically analyzing Android applications w.r.t. security and privacy aspects. Interestingly, not many approaches address the security analysis of the Android platform itself, although its source code has been published. Due to the complexity of the source code, however, approaches are desirable that help one better understand the implemented security architecture of Android. This is even more important as no sufficiently detailed documents on the security architecture are publicly available.

As a first step to better understand the implemented security architecture of Android, we propose an approach to statically extracting and analyzing the access control policy of Android system services. The Android platform makes available about 30 system services, which run under a privileged user ID and provide critical functionality, such as managing installed applications, providing access to location information, offering Bluetooth connectivity, and managing user accounts.

The access control policy of the system services is largely implemented by means of specific access control checks, which are also called "service hooks" [12]. These service hooks use predefined Android APIs, such as the `enforceCallingPermission()` method of the `Context` class and several variants [21]. More than 350 service hook calls exist within the system services, for example, the statement

```
mContext.enforceCallingPermission(BLUETOOTH);
```

enforces the `BLUETOOTH` permission. If necessary, these checks are augmented with additional access conditions.

Due to the complexity of the source code, we propose a *two-step* analysis technique. In the first step, we extract the access control checks as well as their data and control dependences automatically from the source code. This step allows an analyst to better comprehend the implemented access control policy because she can focus on the implemented policy and does not have to consider unrelated code. We employ interprocedural program slicing for this purpose, which is a well-studied technique for program comprehension [23]. Due to the fact that program slicing respects data as well as control dependences, we can consider those program statements which influence the access control decision. In particular, through control dependences, we can capture conditional enforcement. Here, the access control checks depend on certain additional conditions, e.g.,

```
if(callingUid != Process.SYSTEM_UID)
            mContext.enforceCallingPermission(perm);
```

We have implemented our slicing tool based on the WALA analysis tool-suite, a mature source and bytecode analysis framework for Java, which makes available sophisticated slicing algorithms [9]. The main motivation behind this extraction process is to simplify the subsequent analysis step, although

program slicing can be utilized on its own to better understand the access control policy.

The analysis step aims to detect flaws and inconsistencies in the implemented access control policy of the system services, e.g., allowing access to functionality without the required permissions. For this purpose, we employ the extended static checker ESC/Java2 [16], [2], which is based on the principle of design by contract (DBC) and allows an analyst to automatically check method specifications, which are in the form of pre- and post conditions, against the implementation. Extended static checking is a kind of light-weight verification, which allows one to analyze real-world applications [16]. ESC/Java2 uses the Java Modeling Language (JML) [25] for the specification of pre- and postconditions. DBC-based specification languages like JML let one conveniently define access control rules as logical formulae, which is more powerful and declarative than rule languages of commercial static code analyzers, such as Fortify SCA [17]. Code annotations are a useful format to redocument the security architecture of a software system unambiguously.

We examined core system services with the help of this approach and detected some inconsistencies. In addition, our technique helped us extract the implemented access control policy of system services and is independent of the Android version, although we concentrated on Android 4.0.3. We employed a static analysis approach as we intend to utilize our analysis infrastructure for future security analyses of the Android Framework. In this way, our work differs from the approach by Felt et al., who generated a permission map for Android 2.2 dynamically [33], [34]. Specifically, we give concrete examples in which the aforementioned map is incomplete due to limited test coverage.

In the end, we believe that our analysis technique can be generalized to show that applications use security APIs to meet their security requirements. Our contribution lies in the *combination* of slicing and extended static checking and their application to the problem of separating and analyzing access control code of real-world software. In particular, we utilize specific knowledge on the software framework to initialize the analysis tools. Also, our proposed technique can serve as a basis for the redocumentation and comprehension of security-relevant code that is distributed over a larger code base.

The remainder of this paper is organized as follows. Section II gives a detailed overview of Android security, JML and extended static checking. We describe our approach in Section III, whereas we report on implementation aspects and our results in Section IV. Section V discusses limitations as well as further prospects of our approach. After a section on related work, we conclude with a summary and an outlook.

II. Background

In the following, we briefly describe JML and extended static checking. Thereafter, we present the Android security concepts relevant to this paper in more detail.

A. The Java Modeling Language and Extended Static Checking

JML is a formal behavioral interface specification language, specifically designed for specifying the functional behavior of Java programs [25]. Due to the fact that JML specifications are written by the Java programmers themselves at the source code level, JML uses a Java-like syntax and is relatively easy to understand by an average programmer. JML provides a rich set of language constructs that are necessary to precisely specify the functional behavior of Java programs, mostly, in the form of class invariants, and methods' pre- and postconditions. This way, JML is based on the DBC principle introduced by Eiffel [29]. JML specifications are written in special annotation comments in the form of `/*@...@*/` or using `//@...` if a single line specification is intended. JML uses `requires` and `ensures` clauses to specify method's pre- and postconditions, respectively. The preconditions enforce the client's obligations, whereas postconditions enforce the implementer's obligations. JML provides a logical variable `\result`, which represents the value returned by a method; the variable `\old` refers to the pre-state of the current method and is used in `ensures` clauses. The fact that exceptions can be thrown is expressed by the `signals` clause and by `exceptional_behavior` statements. The keyword `pure` indicates that a method is free of side effects.

There is a variety of tools available that allow one to check the JML constraints at run-time or (in part) statically [2]. These tools usually check that the code corresponds to the JML specifications. One such tool is ESC/Java2, which can statically detect inconsistencies between the code and the specification using the built-in theorem prover Simplify [16]. However, due to the fact that such conformance checking in general is undecidable, false positives and negatives may be produced. ESC/Java2 employs **modular reasoning**, which is regarded as an effective technique when used in combination with static checking. Code sections, e.g. Java methods, can be analyzed one at a time and their JML-based specifications can be proved by inspecting the specification contracts (and not the code) of the methods they call within their bodies [16].

B. Android Security Concepts

We briefly describe the Android programming model and thereafter discuss Android's security concepts in more detail as far as they are related to this paper's topic.

1) The Android Programming Model: Android provides a specific programming model for application development. Android applications consist of components, which are activities, services, content providers, and broadcast receivers. Activities represent an application's user interface, whereas services implement the application logic. Content providers allow an application to share data (e.g., calendar data, phone lists) with other applications. Broadcast receivers can subscribe to broadcast messages from other applications. Android components/applications interact with other applications via Inter Process Communication (IPC), which is internally implemented by the Binder protocol.

2) Android Security: Android applications are separated from each other by assigning a different Linux user ID to each application [12]. Due to the fact that such a strict isolation prevents one from developing many useful applications, the

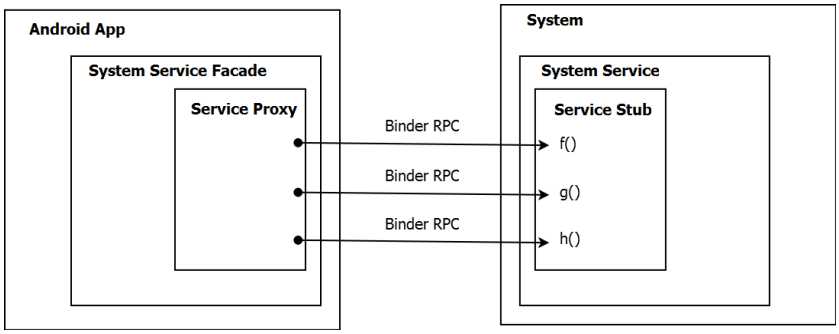

Fig. 1. RPCs with Android system services.

IPC concept has been introduced. However, this IPC must be adequately secured. In Android, an application's components can only be accessed if the caller has sufficient permissions. Permissions are usually granted by a user at installation time via a specific user interface; other permissions can be gained by an application depending on code-signing certificates. Android's permission-based security model has several refinements, such as delegation (e.g., pending intents, URI permissions), shared User IDs, and service hooks [12].

The concept of *service hooks* is relevant to our further discussion. Service hooks allow a developer to secure single methods of a service with permissions rather than the whole service component. This way, more fine-grained access control policies for a service are possible. The Android Framework Classes provide several APIs in the `Context` class for this purpose. `checkCallingPermission(perm)` checks if the permission `perm` has been granted to the calling application. `enforceCallingPermission()` enforces this access decision and throws a security exception if the calling application does not hold the appropriate permission. `checkCallingOrSelfPermission(perm)` is less restrictive in that it additionally returns `PERMISSION_GRANTED` if the callee ("self") holds `perm` and *no IPC* is currently being processed. This point is interesting as the Android Framework also uses the `clearCallingIdentity()` and `restoreCallingIdentity()` API calls for privilege management. `clearCallingIdentity()` sets the user ID of an incoming IPC to the current process' user ID, whereas `restoreCallingIdentity()` restores the original user ID. If this privilege management is used wrongly, `checkCallingOrSelfPermission()` will grant the requested permission, although this was not intended. In case of a hole in privilege management, all code locations can be attacked that use this service-hook variant. Consequently, it is important to know where `checkCallingOrSelfPermission()` is called. A tool for the comprehension of the access control policy of the Android Framework should provide enough information to distinguish between the different kinds of service hook calls.

3) Securing System Services: Service hooks are heavily used by Android system services, although third-party applications can also employ this security feature to secure their exported APIs. Android makes available about 30 system services, which run under the system user ID and export a rich set of APIs. Typical examples are the `PackageManagerService`, the `BluetoothService`, the `TelephonyRegistry`, and the `LocationManagerService`.

A system service is started as a single instance at system start-up. Android applications can access these service instances through APIs, which are implemented as remote procedure calls (RPCs). The Android Framework Classes provide a facade for each system service, e.g., the `PackageManager` class. Each facade contains a proxy object which actually makes the RPCs to the remote system service instance via the Binder device. From a security point of view, this facade is not relevant because it only implements the proxy for the remote system service. In fact, the proxy is under the control of the calling applications and hence of a possible attacker, who can access the proxy via Java reflection.

From the security viewpoint, the exported interface of a system service is of importance. It contains those methods of the system service for which a Binder transaction is defined, i.e., which are remotely accessible. The exported methods are defined in a Java interface, which is a subinterface of `android.os.IInterface` and is implemented by the system service. Summarizing, these exported methods belong to the attack surface of the system services and consequently of the system user ID. Fig. 1 gives an overview of how the access to system services in Android works.

The Android platform secures the exported methods of system services by placing service hooks before security-critical code. A typical example is the `getBluetoothState()` method of the `BluetoothService`:

```
public int getBluetoothState(){
  mContext.enforceCallingOrSelfPermission(BLUETOOTH);
  return mBluetoothState;
}
```

The Android documentation describes for each API method (of a system service's facade) if and which permissions are needed. This information gives one a hint which permissions the corresponding system service must actually enforce by mapping the API methods to remote calls of a system service. Please note that there is not much documentation available in the source code of the system services, so we can deduce this permission map only indirectly. Also, there are further

High-level algorithm: Abstraction, annotation, and verification of a system service

Input: The source file of an Android system service

Step 1: Construct an SDG and generate instructions in SSA form from the source, with the exported methods as the entry points.

Step 2a: Search for all service hook statements (e.g., *enforceCallingOrSelfPermission()*, and *checkCallingPermission()*) with the help of the SDG's call graph part as well as the SSA representation and add them to the slicing criterion.

Step 2b: Tabulate the enforced permissions for each entry point using the SDG's call graph part.

Step 3: Do context-sensitive backward slicing on the SDG with respect to the slicing criterion.

Step 4: Create a new source file from the SDG/SSA representation.

Step 5: Insert JML annotations for each method (automatically inferred or via a user interface).

Step 6: Call the extended static checker on the annotated code.

Result: A static checker's report on specification violations of the access control policy

Fig. 2. Algorithm for extracting and analyzing the implemented access control policy of a system service.

methods exported by a system service which are only called by hidden and not the official APIs. For example, this is the case for methods of the `DevicePolicyManager`.

Having extracted the policy, one interesting task is to verify if this expected policy has been implemented correctly with the help of service hooks. For example, in an earlier Android version, a bug was introduced which allowed applications to access the Android camera without any permission. The Android developer responsible for fixing this problem noted:

Some debugging code was added to camera service. Later it was # ifdef'd out, but this change also removed the camera permission check. [35]

Certainly, the problem occurred at the C++ level as the camera service is implemented by native code. However, permission checks exist at the C++ level which are similar to the Java-based service hooks. In general, extracting and verifying the implemented access control policy of the system services is a main topic of the rest of this paper.

III. Our Approach

Our approach aims to better understand the access control policy of Android system services. We are not the first ones dealing with this topic. In fact, our work was inspired by Felt et al. who generated a permission map for the Android Framework Classes dynamically by means of code instrumentation [33]. One difference is that we use static analysis. In addition, we take a more long-term perspective in providing an analysis infrastructure for the Android Framework, which can be employed for other security analyses as well (beyond checking the service hooks). In contrast, Felt et al. used their map for their Stowaway tool, which aims to detect overprivileged Android applications [33]. There are also approaches to generate a permission map statically that are based on call graphs (e.g., [19]). We also compare our technique with these approaches in the course of the paper.

Our analysis technique comprises two steps. First, we extract the implemented access control policy from the system services' source code by program slicing. The second step performs the lightweight verification with the help of extended static checking on the sliced version of the system service. The innovation lies in the combination of both software engineering techniques and the fact that we exploit Android Framework know-how (about service hooks) to adequately initialize the slicer. Both steps are now explained in more detail. Fig. 2 depicts the overall algorithm used for the abstraction and verification tasks—this figure is described later.

A. Extracting the Access Control Policy with Slicing

For extracting the access control policy, we use a technique called "slicing", first introduced by Weiser [38]. A backward slicing algorithm starts from a statement, the "slicing criterion", and calculates all statements which (transitively) influence the slicing criterion. Slicing is often used for program comprehension and debugging tasks in order to focus only on those code parts that are relevant to the analysis. Technically, slicing is usually implemented based on system dependence graphs (SDGs) [23]. SDGs contain the statements in static single assignment form (SSA), an intermediate representation well-suited to data and control flow analyses, as well as call graph information. An SDG represents methods via special nodes. Context-sensitive slicing only allows accessible execution paths, i.e., a method must return to the site where the method has been called and not to other call sites of the method.

We now discuss the slicing approach in the context of the Android system services. Our task is to extract the access control policy implemented by the service hooks, i.e., we, for example, use `checkCallingPermission()`, `checkCallingOrSelfPermission()`, and `enforceCallingOrSelfPermission()` statements as slicing criteria. We employ the SDG for **automatically** finding these criteria statements. Starting from the entry points of a system service (i.e., the remote interface), we visit all reachable

```
public void removeActiveAdmin(ComponentName adminReceiver) {
    ActiveAdmin admin = getActiveAdminUncheckedLocked(adminReceiver);
    if(admin == null) return;
    if(admin.getUid() != Binder.getCallingUid()) {
        mContext.enforceCallingOrSelfPermission(BIND_DEVICE_ADMIN, null);
            }
    long ident = Binder.clearCallingIdentity();
    try{
        removeActiveAdminLocked(adminReceiver);
    }finally {
            Binder.restoreCallingIdentity(ident);
    }
}

ActiveAdmin getActiveAdminUncheckedLocked(ComponentName who) {
    ActiveAdmin admin = mAdminMap.get(who);
    if(admin!= null && who.getPackageName().equals(admin.info.getActivityInfo().packageName)&&
        who.getClassName().equals(admin.info.getActivityInfo().name)){

        return admin;
    }
    return null;
}
```

Fig. 3. An example of a slice taken from the `DevicePolicyManagerService`.

methods in the call graph. Within each visited method, we loop through all statements in SSA form until we find service hook calls. We collect all these statements from different methods into *one* slicing criterion.

Starting from this slicing criterion (the access checks), we compute all influencing statements via backward slicing, i.e., the statements which influence the access decision. Fig. 3 depicts an example taken from the `DevicePolicyManagerService`, a system service providing device management functionality. In our example, the slicing criterion is

```
mContext.enforceCallingOrSelfPermission
                                (BIND_DEVICE_ADMIN);
```

The underlined statements in Fig. 3 belong to the slice. Please observe that we also descend into methods called on the slicing path, such as the `getActiveAdminUncheckedLocked()` method because we use interprocedural slicing. In the appendix, we present a more complete example to give the reader a feel how the sliced and annotated code looks like.

B. Verification of the Access Control Policy

Our approach to the light-weight verification of system services works as follows. For each exported method of a system service, we provide a JML specification (which is discussed shortly). We use the sliced method rather than the original method for verification as we would like to focus on the implementation of the access control policy.

In principle, we could have carried out this task without the aforementioned abstraction step, i.e., directly on the original source code. However, as pointed out by Lloyd and Jürjens, for example, this can lead to performance problems. In particular, the verification conditions, which are generated by ESC/Java2 for each method, will become large or even cannot be generated at all [27], [16]. The background is that ESC/Java2 is based on the Simplify theorem prover, which needs the verification conditions representing each analyzed method as input [16]. Having a smaller source code at hand makes the problem more tractable if methods are large, which is often the case in the implementation of the Android Framework (there are methods with more than 800 lines of code, which show dependencies with many other methods and classes).

Fig. 4 displays a sliced method, which is taken from the `BluetoothService` class, together with its JML annotation. The annotation expresses that the normal behavior of the method requires that `checkCallingOrSelfPermission(BLUETOOTH)` returns `PERMISSION_GRANTED`, otherwise a security exception must be thrown because the caller does not possess the permission to execute the method. This kind of specification pattern can be used for most of the system services' methods, only with the appropriate permissions inserted. Additional patterns, for example, are that two permissions or one of two permissions are required. One can also see that the annotation distinguishes between the different versions of service hook calls; in this case, `checkCallingOrSelfPermission()` is used and not the more secure variant `checkCallingPermission()` (see Section II-B).

In Fig. 4, we also give the annotation for the `enforceCallingOrSelfPermission()` method, which guarantees that a security exception is thrown if `checkCallingOrSelfPermission(perm)` does not return `PERMISSION_GRANTED`. The method `checkCallingOrSelfPermission()` has the postcondition `true`, which is the weakest postcondition and is trivially satisfied.

1) Static checking based on modular reasoning: We assume that `enforceCallingOrSelfPermission()` and `checkCallingOrSelfPermission()` have been implemented correctly. The verification of these methods is a different and complex task, e.g., RPCs with the system services `ActivityManagerService` and `PackageManagerService` must be handled. We abstract from these details and define stub methods (not displayed). The simplified stub methods must only satisfy the contract of the original methods. Our task is to show that they are *used correctly*

rather than proving the correctness of their implementation. We rely on modular reasoning, a central concept of extended static checking, as mentioned in Section II-A. The contract of a callee and not its implementation is important for static checking. Consequently, we have decomposed the analysis task into two parts.

Calling ESC/Java2, we can identify all exported API calls which do not satisfy their specification, i.e., we attempt to detect violations of the access control policy. ESC/Java2 then emits warnings which show possible violations (e.g., this is the case for the specification of the `getTrustState()` method shown in Fig. 4). Due to the fact that ESC/Java2 may produce false positives, these results must then be cross-checked. Please note that we here employ ESC/Java2 as an annotation checker rather than searching for null pointer dereferences and array bounds errors [16].

2) Permission inference: One problem while working with extended static checking is the annotation burden [16]. To give an analyst a *starting point*, we implemented the following feature. Since the SDG comprises the call graph, we can determine and tabulate the service hook calls (enforced permissions) for each entry point—this serves as a hypothesis for the enforced access control policy. We perform this step when we collect the service hook calls as slicing criteria (see Section III-A). If more than one service hook call is noted, we conservatively assume the "logical and" of the corresponding permissions. Please notice that approaches that generate a permission map statically normally use an approach based on the call graph [19]. Additionally, the SDG provides information on the statement level which allows us to perform analyses on the code level as well in contrast to techniques that only use the call graph.

If a permission check is not called in an entry point, but later in a method along a call path, then all methods of this path are also automatically annotated with a corresponding JML specification. In the special case that the permission parameter of the service hook call is a variable and not a string constant, we cannot directly deduce the called permission (as there is no concrete value). Using the SDG, we have implemented an analysis to determine the permission value. We walk back through the call graph until we find method calls with concrete permission values (of the namespace `Manifest.Permission`) as parameters. The discussed situation occurred in two of the analyzed system services where the security checks have been factored out in specific methods.

Certainly, the aforementioned permission inference produces imprecise results, but if ESC/Java2 gives a warning, then we have interesting special cases that can be further analyzed, e.g., by inspecting the code. Often such cases are not mentioned in available permission maps as we discuss below. Furthermore, we later describe a user interface where an analyst can enter additional specifications or select specification templates to improve these results.

C. Putting It All Together

Fig. 2 depicts the overall algorithm. Input of this algorithm is the Java source code of the system services. From the Java interface, which each system service exports, we obtain all methods which are remotely accessible. These are the entry points for the slicer. The slicer then builds the intermediate representation (SDG) and performs the slicing. Afterwards, the sliced (intermediate) code is translated back to the source code. For each method, the tool automatically inserts the inferred annotations by means of the tool or manually via a graphical user interface. Thereafter, we call the static checker which (possibly) emits warnings that are interpreted by an analyst.

```
/*@
 public normal_behavior
 requires checkCallingOrSelfPermission(BLUETOOTH)
    == PERMISSION_GRANTED;
 also
 public exceptional_behavior
 requires !(checkCallingOrSelfPermission(BLUETOOTH)
    == PERMISSION_GRANTED);
 signals_only SecurityException;
@*/
public synchronized boolean getTrustState(String addr){
  if(!BluetoothAdapter.checkBluetoothAddress(addr)){
    mContext.enforceCallingOrSelfPermission(BLUETOOTH);
  }
  return true;
}

/*@
 public normal_behavior
 requires checkCallingOrSelfPermission(perm)
   == PERMISSION_GRANTED;
 also
 public exceptional_behavior
 requires !(checkCallingOrSelfPermission(perm)
   == PERMISSION_GRANTED);
 signals_only SecurityException;
@*/
public/*@ pure*/void enforceCallingOrSelfPermission(
     String perm){
// ... code omitted
}
```

Fig. 4. Annotated sliced source code.

IV. Implementation and Results

We now describe implementation aspects in more detail and discuss some of the results we achieved by our project.

A. Implementation

WALA: We implemented the concepts described in the previous section in a tool. For the slicing part, we used WALA, a tool-suite developed by IBM [9]. WALA supports slicing in several variants (e.g., forward and backward slicing, thin slicing). Furthermore, WALA is mature and support is still available. Last but not least, WALA lets one implement new kinds of analyses and can serve as the basis for other security analysis tasks (see Section V).

Generating sliced source files: We implemented the analysis concept illustrated in Fig. 2 with WALA and ESC/Java2 as follows. First, we compile the Java source code of a system service into bytecode and load it into WALA, which generates the SDG (including the statements in SSA form) as an intermediate representation (IR). Please notice that WALA lets one define an analysis scope such that we can restrict our

analyses to the system services and do not need to consider the whole Android Framework.

In the next step, we collect all the statements of the slicing criterion as described in Section III-A. For this purpose, we implemented a depth-first search (DFS) on the call graph part of the SDG. We then traverse the call graph with the help of the DFS and for each visited graph node, we loop through its statements. Proceeding this way, we find all service hook calls, which form the slicing criterion. We finally conduct a backward slicing w.r.t. this criterion.

We chose a slicing option which ignores data dependence edges to/from heap locations because otherwise too large SDGs are generated [36]. The slicer's control-flow option, however, is fully enabled, which is needed for considering, e.g., if-statements (see Section IV-B1).

Since WALA is an analysis tool-suite rather than a transformation and code generating tool, we could not translate the generated slice (which is in the form of WALA's IR) back to source code or at least to bytecode. To address this problem, we use a mapping between the statements belonging to the slice and the source code's line numbers. This mapping is provided by WALA.

Based on this mapping, we build a new source file containing the sliced code. For this task, we employ WALA's IR *as well as* the original source code of the system service under investigation. All the information of the class (name, parent class, declared fields, and method information, such as the signature and the name) can be retrieved from the IR via WALA's API and is written to the new source file. We then walk through all the statements of the slice (which are represented by the IR) and obtain the line number for each statement. Using this line number, we can select the corresponding line in the original source file and copy it to the newly created source file. This step is quite simple because the statements appear in increasing order w.r.t. the line number in the IR.

On employing this basic approach, one technical issue still remains. Not all lines of the slice represent *complete* Java statements, which leads to syntactically incorrect code. For example, consider the following if-statement

```
if(checkPermission(...)==PERMISSION_GRANTED){
 ... some code
}
```

If we slice the code w.r.t. the criterion `checkPermission()` and employ the aforementioned approach, only the first line will be added to the slice—the body does not appear in the slice as it does not influence the criterion. To address this issue, we decided to transform the source code with a Java parser [31] into an abstract syntax tree rather than using the original source file. This additional step allowed us to determine and emit the minimum complete Java statement which contains the line in question.

Graphical User Interface: To improve the results of the automated inference process (see Section III-B2), we display a dialog window. Here, an analyst can select methods of a system service to be analyzed. For a method, the user can choose between one of three predefined patterns (one permission, logical and of two permissions, logical or), which must be filled in with the concrete permissions to be checked. This way, in most cases, the user is not confronted with JML annotations. Furthermore, she can apply a pattern to a *collection of methods* rather than a single method, which leads to a higher degree of automation, as there are often many methods requiring the same permission. The user can also insert JML specifications manually because in some situations specific security checks exist in addition to the service hooks (see also Section IV-B2). An analyst can can obtain such information from consulting the Framework documentation or available permission maps.

B. Experiments and Results

In this section, we present the results and experience we gained from our approach. Beyond the analysis of service hooks, we report on three other experiments conducted with the help of our slicing infrastructure, i.e., extracting the permission enforcement for whole Android components, identifying locations in which security exceptions are thrown, and identifying privileged regions. We conclude with observations that we made during the analysis of the system services.

1) Service Hooks: We analyzed ten system services, among them the two most complex ones, `ActivityManagerService.java` and `PackageManagerService.java`, with about 50k LoC in total (see Table I). From Table I, one can conclude that the sliced files become considerably smaller. In total, as an approximation, they are about 11% of the original code. Due to WALA's `NO_HEAP` option and the possibility to restrict the search space, each file was sliced on an ordinary laptop within a few seconds. Also, the verification step took only a few seconds, which shows that slicing enabled us to apply extended static checking to a real-world application.

We now describe some of our findings. Almost all system-service checks behaved as expected, but we found a few inconsistencies. We also detected conditional security checks, i.e., the enforcement is only carried out under certain conditions. The last point is interesting from the viewpoint of program comprehension to understand the implemented policy in detail.

a) Conditional enforcement: One example of conditional enforcement is given in Fig. 3 where the access control check is only executed if the condition

```
admin.getUid() != Binder.getCallingUid()
```

holds, i.e., if the current app runs under a user ID different from the administrator app. ESC/Java2 helped us detect such cases during the automated annotation and validation process (see Section III-B2). If there is a condition, ESC/Java2 displays a warning because there are paths through the method which do not satisfy the postcondition. Then, one can look into the source code to understand the situation better and provide more precise annotations (see also Appendix, `removeActiveAdmin()`) via the user interface. As a result, we found that in most of the analyzed services, such specific cases exist, although they are not mentioned in the API documentation nor in Java comments.

In the services listed in Table I, 65 hooks depend on additional conditions. This is about 27% of the total number of hooks in these services.

In addition, the aforementioned example also reveals some weaknesses of the permission map provided by Felt et al.[1] The `removeActiveAdmin()` method is not listed[2], i.e., the map assumes that no permission is required. It is likely that testing has difficulty in dealing with conditions. The dynamic approach only considered a test case in which the aforementioned condition is false. Similarly, several APIs of the `WindowManagerService`, such as `addWindowToken()`, enforce the permission `MANAGE_APP_TOKENS` only if the caller's process ID is different from the system process ID. This condition again is not noted in the map.

b) Inconsistencies: On analyzing system services, we detected some inconsistencies w.r.t. the Android Framework documentation. As an example, we subsequently show a code extract of the `setTrust()` API of the `BluetoothService`:

```
public  boolean setTrust(String addr, boolean val){
  if(!BluetoothAdapter.checkBluetoothAddress(addr)){
     mContext.enforceCallingOrSelfPermission(
          BLUETOOTH_ADMIN);
     return false;
  }
  if(!isEnabledInternal()) return false;
  return setDevicePropertyBooleanNative(...);
}.
```

The permission enforcement is called within an if-statement and after this check, the method returns. `checkBluetoothAddress()` validates a Bluetooth address. If `addr` is syntactically correct, then the required `BLUETOOTH_ADMIN` permission is not enforced, whereas only if the address is wrong, there is a permission check. Our tool also flagged this as well as the `getTrustState()` and `getRemoteClass()` methods, which show the same behavior. ESC/Java2 issues a warning here because not all paths through these methods enforce the permission. Conversely, we found through manual inspection that `getRemoteUuids()` implemented the expected enforcement (the access control check is before the if-statement) and hence was not flagged by our tool. We reported the aforementioned inconsistencies to Google; Google will fix them in the future. Interestingly, the permission map produced by Felt et al. also did not cover this issue.

Another inconsistency stems from the fact that the Android documentation explicitly notes that one needs the `BLUETOOTH` permission to exercise `BLUETOOTH_ADMIN` [20]. This statement is not supported by the current implementation of the `BluetoothService`. We found about ten methods secured by `BLUETOOTH_ADMIN`, not enforcing `BLUETOOTH`, e.g., `startDiscovery()`, `cancelDiscovery()`, and `enable()`. We also verified via the Android manifest that there is no additional protection of the whole `BluetoothService` with the `BLUETOOTH` permission.

[1]Certainly, in other cases, the dynamic approach works better than a static one, e.g., it considers the `INTERNET` permission, which is enforced by Unix groups rather than service hooks.

[2]Please note that the map only indirectly lists the interface of the system services by enumerating the proxy methods.

Source Code File [.java]	Lines [k]	Lines per Slice [k]
AccountManagerService	2.5	0.3
ActivityManagerService	15	1.1
BluetoothService	2.8	0.7
PackageManagerService	9	0.3
DevicePolicyManagerService	1	0.1
TelephonyRegistry	0.5	0.1
LocationManagerService	2.5	0.4
BackupManagerService	5.5	1.3
PhoneInterfaceManager	0.7	0.2
WindowManagerService	10.1	0.6
In Total	*49.6*	*5.1*

TABLE I
LOC FOR SYSTEM SERVICES AND THE SLICED FILES.

```
public void revokePermission(String packageName,
                                    String permissionName){
  final PackageParser.Package pkg =
                                mPackages.get(packageName);
  if (pkg.applicationInfo.uid != Binder.getCallingUid())
    mContext.enforceCallingOrSelfPermission
                          (GRANT_REVOKE_PERMISSIONS, null);
}
```

Fig. 5. Sliced source code of the `PackageManagerService`, version 4.1.1.

c) Different Android versions and regression testing: Our technique is also valuable for regression testing. Here, an analyst can start from an already analyzed version of the Android platform. Then only the difference between both version needs to be analyzed with the help of our approach. Typical changes in the policy may occur when new APIs are added or new, more fine-grained permissions are introduced.

We carried out such a task for the `PackageManagerService` and followed the versions from 4.0.3 to 4.4.2. Fig. 5 displays sliced code from the `PackageManagerService`. `revokePermision()` was only a *local* method before Android 4.1.1 (similar remarks hold for the corresponding `grantPermision()` API). It is now exported to allow for the revocation of permissions at runtime. It still seems to be an undocumented feature that is exported as a hidden API and can only be accessed via Java reflection. Our approach contributes to a better understanding of such "hidden features" and the corresponding access control policy.

2) Security Program Comprehension Tasks: In this and the following section, we show how to employ our slicing infrastructure for the task of security program comprehension. This task allows an analyst to better understand additional security checks and mechanisms which are implemented in the Android platform. For example, many additional security checks exist in the Android Framework, which are not of the form of service hooks and are mostly undocumented. They range from ensuring that certain security-critical operations can only be performed by the system process to checking code-

signing certificates for code instrumentation. We found about 50 additional checks within the examined system services.

Slicing helps one understand and redocument these security mechanisms. We can locate these checks within the code by searching for `throw new SecurityException();` statements with the help of WALA and carry out backward slicing, i.e., we use these throw statements as slicing criteria. Technically, WALA represents throw statements as `SSAAbstractThrowInstruction` and and hence we can search for them in a similar way as we have done for the security hooks. In the following, we discuss three different slicing tasks, which we performed on the system services.

a) Protected broadcasts: We traced one additional concept, which is called "protected system broadcasts" and which is also mentioned in the security literature [10], [33]. The Android Framework prevents specific system broadcasts, such as "battery low", from being sent from third-party applications in order to prevent denial-of-service attacks. Specific security checks are inserted into the code for implementing this feature. We identified the code location in the `ActivityManagerService` class, which realizes this concept, and attempted to track its dependences by means of backward slicing.

Starting from a `throw` statement with a `SecurityException`, which occurs in the `broadcastIntentLocked()` method, we located the following security check via the backward slice:

```
ActivityThread.getPackageManager().
              isProtectedBroadcast(intent.getAction()).
```

We further followed the slice to the public method `broadcastIntent()` of the `ActivityManagerService` class. The method `broadcastIntent()` is then called by the `Context.send*Broadcast()` methods, the public API calls for sending broadcasts, via RPCs. Since our slicing algorithm currently does not support RPCs, we manually followed this last call. Similarly, the slicing algorithm cannot conclude that the aforementioned `isProtectedBroadcast()` call is an RPC with the `PackageManagerService`.

In the future, we will enhance our approach to support RPCs by inserting additional edges into the call graph. For example, we can replace the corresponding `invokespecial` instructions of the `IPackageManager` interface with the `invokevirtual` instructions of the `PackageManagerService`. These modifications allow us to directly connect different system services in case of RPC calls.

b) Account authenticators: We examined the `AccountManagerService`, which provides centralized access to a user's accounts. To interface with this service, specific authenticator apps must be implemented, e.g., for Google and Facebook accounts. Authenticators store user passwords as well as account information locally and handle credential validation. This way, an app can request reusable authentication tokens to access a server without sending passwords repeatedly. The `AccountManagerService` provides methods, which should only be called from authenticators. For example, to call `AccountManagerService.setPassword()`, an app needs the permission `AUTHENTICATE_ACCOUNTS` *and* must have the same user ID as the account's authenticator app. Only the authenticator is allowed to store passwords for its account locally; authenticators with other user IDs may not access the account's password.

The permission check can be extracted by our slicing approach using the criterion `checkCallingOrSelfPermission(perm)`. The second condition (the user ID check), however, does not influence the permission check. For this reason, we performed a second slicing task with the corresponding throw statement as slicing seed. With this slicing task, we detected that an app does not need to have the same user ID as the authenticator. It suffices that the authenticator and the calling app are signed by the same developer. The reason for this more liberal access policy is unclear. Probably, it allows two or more apps of the same developer to share authentication tokens without additional user interaction. However, the incomplete documentation may lead to confusion. For example, although Facebook uses the account manager, the Facebook app and the Facebook Messenger app share tokens via a content provider rather than using the account manager as we found out by manual inspection of the decompiled code.

c) Enabling components: In the `PackageManagerService`, we also found conditional access checks, which we identified via a slice w.r.t. a throw statement. For example, the `setApplicationEnabledSetting()` and `setComponentEnabledSetting()` methods can be called without a permission if the caller has the same ID as the application/component to be changed. The following code fragment depicts the corresponding slice:

```
final int uid = Binder.getCallingUid();
final int permission = mContext.checkCallingPermission
        (CHANGE_COMPONENT_ENABLED_STATE);
final boolean allowedByPermission=
      (permission==PERMISSION_GRANTED);
if(!allowedByPermission && (uid!=pkgSetting.userId)){
              throw new SecurityException(...);
```

This example again shows that the combination of service hooks and throw statements as slicing criteria allows an analyst to extract undocumented security checks from the code. Certainly, most of these checks are reasonable and pose no security problem, but documenting them gives an analyst a clearer and more comprehensive picture of Android's access control policy.

This last point becomes apparent when applying Fortify SCA on Android apps. The latest release of Fortify SCA supports some specific analyses for Android. For example, it implements an analysis to detect underprivileged applications and uses a permission map for this purpose. On analyzing the K9Mail client, for example, Fortify SCA reports that this app calls `setComponentEnabledSetting()` without the permission `CHANGE_COMPONENT_ENABLED_STATE`. However, this is a false positive. The permission is not needed because the condition `uid!=pkgSetting.userId` is false in this case as code inspection revealed. Fortify's warning is misleading because it is classified with severity "high". Fortify SCA also

erroneously reports a missing API permission for the method `openInputStream()` of the class `ContentResolver` (although not in a system service).

3) Identification of Privileged Code: Our analysis infrastructure allows us to detect mistakes in privilege management. This topic has been discussed in the security community for a long time, e.g., in the context of Unix systems [5]. As mentioned in Section II-B2, Android makes available the method pair `clearCallingIdentity()` and `restoreCallingIdentity()` for privilege management. We search for privileged regions in the system services. Then, we instruct ESC/Java2 as a protocol checker [26] to make sure that for each `clearCallingIdentity()` call, a corresponding `restoreCallingIdentity()` call exists on each execution path. For this purpose, we define the following specification:

```
//@ ensures Binder.callingID==\old(Binder.callingID);
```

This annotation ensures that the calling ID is reset on method return. We only examined a few services to demonstrate the feasibility of this analysis. For instance, in the `unbindFinishedService()` and `publishService()` methods of the `ActivityManagerService`, `restoreCallingIdentity()` is erroneously called within the body of an if-statement. If the condition of this statement is false, the user ID is not reset. We could not find an exploit of this vulnerability because the method returns immediately and control goes back to the calling app. However, since `clearCallingIdentity()` is called in many places all over the Android Framework (ca. 140 locations in the examined system services), it is important to handle privilege management carefully. `getCallingUid()` and `checkCallingOrSelfPermission()` depend on correct privilege management.

4) Observations: We made the following observations on analyzing core system services:

1) The implementers of the Android OS heavily used their own framework to secure system services, e.g., the `check*Permission()` and `enforce*Permission()` methods as well as the `clearCallingIdentity()` or `getCallingUid()` APIs. The fact that framework functionality is used simplifies the comprehension and analysis of the implemented access control checks. In particular, this specific structure allows one to *automatically select* slicing criteria. Plus, one can employ the semantics of these APIs to check whether these API calls are used correctly to enforce the intended policy.
2) The consideration of the service hooks *as well as* throw statements gives one a comprehensive picture of the in-code access checks of the system services as the conducted slicing tasks indicate. These results complement already available permission maps [34].
3) Many security checks in the system services depend on determining the calling user or process ID (e.g., via `Binder.getCallingUid()`). Hence, the Binder component, which determines the user ID, is an interesting target for further security analyses; flaws in this component may allow an attacker to compromise all installed apps, including system apps.
4) Each system service must guarantee specific security requirements, which are often not explicitly or only vaguely described in the developer documentation. The comprehension of system services helps a developer better understand the security implications of using Android APIs. Software security comprehension is a first step beyond current static code analyzers and a completely manual code review.

V. Discussion of the Approach

In this section, we discuss some problems related to the currently available tool support. We also describe how and to which extent our approach can be extended.

a) Limitations of the current tool support: We have already mentioned that the static checking approach can produce false positives and negatives. At least, the questionable code locations mentioned in Section IV-B1 were found this way. In general, one must thoroughly select the annotations which can be used by ESC/Java2. Another restriction was that only Java versions up to 1.4 are supported, i.e., Java generics cannot adequately be handled. For this reason, we had to manually replace Android code, which uses these generic types (however, only at a few places because slicing eliminated most dependences).

As the OpenJML initiative is currently under way, we have great hope that our approach can be applied to larger case studies in industrial contexts in the near future. This newer extended static checker is expected to leverage more powerful backend SMT solvers such as Yices and Z3, but the current implementation of heavyweight specifications (as used for exceptional behavior statements) is incomplete.

In addition, WALA's `NO_HEAP` option can also be a source of false negatives as data dependences related to heap locations might be lost. We manually checked that such situations did not occur in the services depicted in Table I. Furthermore, due to the control-dependence option all locations with conditional enforcement could be identified. Even if one does not consider the analysis with ESC/Java2, program comprehension w.r.t. security is eased due to slicing as our experiments with slicing demonstrate.

b) Support of native system services: Our current prototype does not cover native system services, which are implemented in C++, such as the `CameraService` or `AudioFlinger` (see Section II-B3), and which use the C version of `checkCallingPermission()`. In case of native code, we can resort to tools supporting C/C++, such as the commercial CodeSurfer slicer [1] and the static checker Eau Claire [7]. We do not see any fundamental obstacles which prevent one from applying our technique to native services.

c) Improved inference of JML annotations: In Section III-B2, we have described how to automate the annotation process, but the result is a coarse estimation. As a further

```
1 public boolean addAccount(Account account, String password, Bundle extras){
2  // checkAuthenticateAccountsPermission(account);
3  long identityToken = clearCallingIdentity();
4  try{
5    return insertAccountIntoDatabase(account, password, extras);
6  }
7  finally{
8    restoreCallingIdentity(identityToken);
9   }
10 }
11
12 /*@ public normal_behavior
13  requires mAccessInsertAccountIntoDatabase == true ==>
14                     checkCallingOrSelfPermission(AUTHENTICATE_ACCOUNTS) == PERMISSION_GRANTED;
15  also
16  public exceptional_behavior
17  requires !(mAccessInsertAccountIntoDatabase == true ==>
18                     checkCallingOrSelfPermission(AUTHENTICATE_ACCOUNTS) == PERMISSION_GRANTED);
19  signals_only SecurityException;
20 @*/
21 public boolean addAccount(Account account, String password,Bundle extras){
22   mAccessInsertAccountIntoDatabase = false;
23   try{
24     mAccessInsertAccountIntoDatabase = true;
25     return true; // dummy
26   }
27   finally{}
28 }
```

Fig. 6. Code transformation with respect to a critical access.

improvement, we can, for example, collect all the if-conditions on which the slicing criterion (e.g., service hook call, throw statements) depends since the SDG contains all program statements. Then we can insert the logical "and" of all collected conditions into normal or exceptional behavior statements. Alternatively, we can employ tools, such as Daikon [14], that automatically infer likely JML annotations and insert them at the corresponding position in the code. Daikon infers the annotations dynamically by code instrumentation. Proceeding this way, we obtain a fully automated abstraction, annotation, and verification process. Due to the fact that exceptional-behavior statements are currently not supported by Daikon, we will need to extend it accordingly.

d) Missing security checks: The approach described so far has the drawback that it cannot handle situations in which security checks are missing because slicing would remove methods without checks from search space. Subsequently, we outline an approach that allows us to address this situation.

If background knowledge on the implementation and architecture of a system service is available (e.g., through an architect of the service), one can identify the security-critical private data and methods of a system service and map these security-critical resources to the required permissions. We can then carry out the following abstraction step. Introduce for each security-critical resource X a newly-created Boolean member variable `mAccessX`, which indicates access to X. Initialize this flag with `false` at the beginning of a method and replace each access of the corresponding resource with the statement `mAccessX=true;`. This abstraction step simplifies the code while still keeping information on critical resources. Thereafter, the two-phase analysis described in this paper can be applied.

The example depicted in Fig. 6 illustrates the result of our technique. Only for the discussion let us assume that the security check `checkAuthenticateAccountsPermission()` has been commented out (see line 2 of Fig. 6). At the bottom of this figure, we display the transformed and annotated code. The annotation in lines 12-20 states that the critical access can only be done if the permission `AUTHENTICATE_ACCOUNTS` has been granted, otherwise a `SecurityException` is thrown. This specification is violated by the code.

e) Other authorization APIs: Our approach is not restricted to Android. For example, we can apply this technique to Java-based Web applications which use programmatic access control. For example, the Spring software framework makes available certain authorization APIs such as `hasRole()` [32]. Fig. 7 depicts code that uses Spring and shows the correspondence to Android access control. The access control check allows the method to be successfully completed only if the caller has activated the appropriate roles. The `hasRole()` calls correspond to the `check*Permission()` APIs and can be automatically extracted from the code by slicing. In general, all applications that use a similar security library can be dealt with by our approach. The general idea is to check whether security APIs are used correctly w.r.t. a given specification/policy by extracting the security checks from the code. The security APIs themselves do not need to be verified; we utilize modular reasoning to prove that the contract is respected by the client (preconditions) and the postconditions are strong enough to guarantee the security requirements of the framework client.

VI. Related Work

Static code analysis: There are several works on static checking for software security. An overview, for example, is given by Chess and West [6]. In more recent work,

```
/*@
public normal_behavior
requires secExpr.hasRole("Manager") ||
      secExpr.hasRole("Financial Officer");
also
public exceptional_behavior
requires !(secExpr.hasRole("Manager") ||
      secExpr.hasRole("Financial Officer"));
signals_only SecurityException;
@*/
public int getBalance(){
 if(!(secExpr.hasRole("Manager") ||
      secExpr.hasRole("Financial Officer")))
   throw new SecurityException("Access Denied");
 return balance;
}
```

Fig. 7. Annotated Spring code.

Felmetsger et al. employ Daikon to dynamically infer security specifications for web applications. Thereafter, they use a model checker to detect application logic vulnerabilities violating the specifications [15]. This approach resembles our technique in that it combines different analysis techniques and works on Java code. We, however, focus on the understanding of Android's access control policy rather than analyzing web applications.

Furthermore, work exists that deals with the verification of the correct authorization hook placement in Linux kernels, e.g., by Jaeger et al. [24]. They use runtime instrumentation as well as a supplementing static analysis to detect deviations from a normal access control behavior. Their approach utilizes the specifics of Linux kernels, whereas we target at a Java-based system. Furthermore, we focus on a program comprehension task employing interprocedural-slicing algorithms.

DBC and security analysis: Eau Claire allows the formulation of pre- and postconditions as code annotations for C code [7]. Similarly to ESC/Java2, it is based on an automatic theorem prover. Eau Claire can detect general security problems such as buffer overflows and race conditions. Although Eau Claire primarily focuses on common classes of security bugs in C applications, it demonstrates the benefit gained by employing extended static checking for security analysis.

Other works that employ JML in the application security context are presented by Lloyd et al. (a biometric authentication system) [27] and by Cataño et al. (an electronic purse implemented as a Java Card application) [3]. Both works use JML in conjunction with the static checker ESC/Java for an already implemented application, facing several problems concerning the limitations of extended static checking, specifically, too large verification conditions. We address this problem by automatically extracting the implemented access control policy out of the code and leaving out all other code not related to access control. In addition, our approach reduces the annotation overhead, which is usually a burden. This automation is possible because we employ a pattern-based mechanism for JML annotations. JML patterns for security have been introduced by Warnier, but they are not tailored to access control nor do they reflect the Android Framework semantics [37].

Android security: There are several works dealing with Android security; specifically, we have earlier commented on the permission map by Felt et al. Fragkaki et al. also analyze Android's permission system based on a formal model and show that Android's delegation concept contains flaws [18]. Enck et al. validate Android manifest files containing the access control policy of an application with the help of the Kirin tool [13]. Kirin works at the manifest level and does not consider the source code for the analysis. The Saint architecture extends Android's permission model to support context information for access decisions [30]. TaintDroid tracks information flows through the Android platform to detect privacy breaches [11].

There are also approaches to the static analysis of Android applications. Enck et al. developed a tool which allows one to decompile Android's custom bytecode (DEX code) [10]. Thereafter, they employ Fortify SCA and write custom Fortify rules to detect weaknesses or privacy violations in third-party applications, such as sending IMEI numbers or location data to the Internet. In contrast to their work, we use techniques, such as slicing and extended static checking, which are not supported by Fortify SCA [6]. The ComDroid tool leverages static analysis to also consider IPC [8], working on DEX code. This way, several weaknesses in applications have been detected, e.g., through unprotected IPC. Grace et al. analyze system applications, mostly made available by Google and smartphone manufacturers, for leaking system permissions and discovered several critical weaknesses [22]. Recently, Lu et al. tackled the problem of component hijacking in Android apps by a static analysis approach [28]. Gibler et al. presented AndroidLeaks which aims to detect potential leaks of sensitive information in Android applications by employing static analysis [19]. They also constructed a permission map via static analysis using the call graph, but did not consider conditional enforcement nor additional access checks because the call graph does not capture this information. In particular, the analysis of system services was not the goal of that work.

Most of the works mentioned so far focus on the analysis of apps rather than considering the Android Framework. Our approach differs from those works in that it focuses on separating out and analyzing permission-enforcement code from the Android platform (or more generally, from Java-based software). Android apps do not seem to use sophisticated access control features like service hooks, even not business apps such as those from SAP as we checked by decompilation. Since our goal is the comprehension of implemented access control mechanisms, we address a different topic as the aforementioned static analysis approaches for Android apps.

VII. Conclusion and Outlook

We showed in this paper how to extract the implemented access control policy from Android system services by means of slicing and thereafter verified the policy against security specifications with extended static checking. We performed our task with the original code and detected inconsistencies between the Android documentation and implementation. Also, this task contributes to a better understanding of Android's implemented security.

In the future, we will extend our work to cover further security-relevant mechanisms of Android, such as URI permissions and pending intents. Also, we will provide a more accurate inference mechanism for JML annotations based on available tools such as Daikon or an own solution. The extracted code and the inferred JML specifications help an analyst better comprehend the implemented security architecture of system services. The extracted architecture can then be subject to further analyses, e.g., the correctness of the inferred specifications can be analyzed.

References

[1] Anderson, P., Zarins, M.: The CodeSurfer software understanding platform. In: Proc. of the 13th International Workshop on Program Comprehension. pp. 147 – 148 (May 2005)

[2] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)

[3] Cataño, N., Huisman, M.: Formal specification of Gemplus's electronic purse case study. In: FME 2002. vol. LNCS 2391, pp. 272–289. Springer, Berlin (2002)

[4] Chaudhuri, A.: Language-Based Security on Android. In: Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'09). pp. 1–7. ACM (2009)

[5] Chen, H., Wagner, D., Dean, D.: Setuid demystified. In: Proceedings of the 11th USENIX Security Symposium. pp. 171–190 (2002)

[6] Chess, B., West, J.: Secure Programming with Static Analysis. Addison-Wesley (2007)

[7] Chess, B.: Improving computer security using extended static checking. In: IEEE Symposium on Security and Privacy. pp. 160–173. IEEE Computer Society (2002)

[8] Chin, E., Porter Felt, A., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proc. of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, USA. pp. 239–252. ACM (2011)

[9] Dolby, J., Sridharan, M.: Static and Dynamic Program Analysis Using WALA, PLDI Tutorial (2010), http://wala.sourceforge.net/files/PLDI_WALA_Tutorial.pdf

[10] Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proceedings of the 14th USENIX Security Symposium (Aug 2011)

[11] Enck, W., Gilbert, P., gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: 9th USENIX Symposium on Operating Systems Design and Implementation (2010)

[12] Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security & Privacy 7, 50–57 (2009)

[13] Enck, W., Ongtang, M., McDaniel, P.D.: On lightweight mobile phone application certification. In: Proc. of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009. pp. 235–245. ACM (2009)

[14] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69, 35–45 (December 2007)

[15] Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G.: Toward automated detection of logic vulnerabilities in web applications. In: USENIX Security Symp. pp. 143–160 (2010)

[16] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation. pp. 234–245. PLDI '02 (2002)

[17] Fortify Software: Fortify Source Code Analyser (2012), http://www.fortify.com/products

[18] Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android's permission system. In: 17th European Symposium on Research in Computer Security. LNCS, vol. 7459, pp. 1–18 (2012)

[19] Gibler, C., Crussell, J., Erickson, J., Chen, H.: AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In: Trust and Trustworthy Computing, LNCS, vol. 7344, pp. 291–307. Springer (2012)

[20] Google Inc.: Bluetooth (2012), http://developer.android.com/guide/topics/wireless/bluetooth.html

[21] Google Inc.: Permissions (2012), http://developer.android.com/guide/topics/security/permissions.html

[22] Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012). USENIX Association (Feb 2012)

[23] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, TOPLAS 12(1), 26–60 (Jan 1990)

[24] Jaeger, T., Edwards, A., Zhang, X.: Consistency analysis of authorization hook placement in the Linux security modules framework. ACM Trans. Inf. Syst. Secur. 7(2), 175–205 (May 2004)

[25] Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer Academic Publishers (1999)

[26] Leino, K.R.M.: Applications of extended static checking. In: Proc. of the 8th Int'l. Symposium on Static Analysis SAS. LNCS, vol. 2126, pp. 185–193. Springer (2001)

[27] Lloyd, J., Jürjens, J.: Security analysis of a biometric authentication system using UMLsec and JML. In: MoDELS. Lecture Notes in Computer Science, vol. 5795, pp. 77–91. Springer (2009)

[28] Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: statically vetting Android apps for component hijacking vulnerabilities. In: Proc. of the 2012 ACM conference on Computer and communications security. pp. 229–240. CCS '12 (2012)

[29] Meyer, B.: From structured programming to object-oriented design: The road to Eiffel. Structured Programming 10(1), 19–39 (1989)

[30] Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in Android. In: Proceedings of the 25th Annual Computer Security Applications Conference. pp. 340–349 (2009)

[31] Oracle Inc.: Compiler Tree API (2012), http://docs.oracle.com/javase/6/docs/jdk/api/javac/tree/com/sun/source/util/package-summary.html

[32] Pivotal, Inc.: Spring security 3.1.2 (2013), http://static.springsource.org/spring-security/site/index.html

[33] Porter Felt, A., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proc. of the 18th ACM Conf. on Computer and Communications Security, Chicago, USA. pp. 627–638. ACM (2011)

[34] Porter Felt, A., Chin, E., Hanna, S., Song, D., Wagner, D.: STOWAWAY (2011), http://www.android-permissions.org/permissionmap.html

[35] Spark, D.: Enable camera permission check (2009), bug ID = 1869264

[36] Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 112–122. PLDI '07 (2007)

[37] Warnier, M.: Language Based Security for Java and JML. Ph.D. thesis, Radboud University, Nijmegen, Netherlands (2006)

[38] Weiser, M.: Program slicing. In: Proceedings of the International Conference on Software Engineering. pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981)

Appendix

Subsequently, we give the sliced and annotated version of the `DevicePolicyManagerService` as an example. Due to the fact that generics are used, which are not supported by ESC/Java2, we had to replace the reference to the `HashMap<ComponentName, ActiveAdmin>` class. Some side effect-free methods had also to be declared `pure` to assist ESC/Java2 in proving the verification conditions.

```java
package com.android.server;

import java.util.HashMap;
import android.app.admin.DeviceAdminInfo;
import android.content.ComponentName;
import android.content.Context;
import android.content.pm.PackageManager;
import android.os.Binder;

public class DevicePolicyManagerService {

        // final HashMap<ComponentName, ActiveAdmin> mAdminMap = new HashMap<ComponentName, ActiveAdmin>(); commented
            out (Java 1.5)
        final /*@ non_null @*/ HashMap mAdminMap = new HashMap();       // Java 1.4; ESC/Java2
        final /*@ non_null @*/ Context mContext;

        public DevicePolicyManagerService(/*@ non_null @*/ Context context){
                mContext = context;
        }

        /*@ pure @*/ ActiveAdmin getActiveAdminUncheckedLocked(/*@ non_null @*/ ComponentName who) {
                //@ assume mAdminMap.get(who) instanceof ActiveAdmin;
                ActiveAdmin admin = (ActiveAdmin)mAdminMap.get(who);

                if (admin != null
                                && who.getPackageName().equals(
                                                admin.info.getActivityInfo().packageName)
                                && who.getClassName().equals(admin.info.getActivityInfo().name)) {
                        return admin;
                }

                if (admin != null) {
                        return admin;
                }
                return null;
        }

    /*@ public normal_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) ==
            PackageManager.PERMISSION_GRANTED;
        also
        public exceptional_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) !=
            PackageManager.PERMISSION_GRANTED;
        signals_only java.lang.SecurityException;
      @*/
        public void setActiveAdmin(ComponentName adminReceiver, boolean refreshing) {
                mContext.enforceCallingOrSelfPermission(
                                android.Manifest.permission.BIND_DEVICE_ADMIN, null);
        }

    /*@ public normal_behavior
        requires getActiveAdminUncheckedLocked(adminReceiver) == null || mContext.checkCallingOrSelfPermission(android
            .Manifest.permission.BIND_DEVICE_ADMIN) == PackageManager.PERMISSION_GRANTED || !(
            getActiveAdminUncheckedLocked(adminReceiver).getUid() != Binder.getCallingUid()) ;
        also
        public exceptional_behavior
        requires !(getActiveAdminUncheckedLocked(adminReceiver) == null || mContext.checkCallingOrSelfPermission(
            android.Manifest.permission.BIND_DEVICE_ADMIN) == PackageManager.PERMISSION_GRANTED || !(
            getActiveAdminUncheckedLocked(adminReceiver).getUid() != Binder.getCallingUid())) ;
        signals_only java.lang.SecurityException;
      @*/
        public void removeActiveAdmin(/*@ non_null @*/ ComponentName adminReceiver) {
                ActiveAdmin admin = getActiveAdminUncheckedLocked(adminReceiver);
                if (admin == null) {
                        return;
                }
                if (admin.getUid() != Binder.getCallingUid()) {
                        mContext.enforceCallingOrSelfPermission(
                                        android.Manifest.permission.BIND_DEVICE_ADMIN, null);
                }
        }
```

```
    /*@ public normal_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) ==
             PackageManager.PERMISSION_GRANTED;
        also
        public exceptional_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) !=
             PackageManager.PERMISSION_GRANTED;
        signals_only java.lang.SecurityException;
      @*/
        public void getRemoveWarning(ComponentName comp, final RemoteCallback result) {
                mContext.enforceCallingOrSelfPermission(
                                android.Manifest.permission.BIND_DEVICE_ADMIN, null);
        }

    /*@ public normal_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) ==
             PackageManager.PERMISSION_GRANTED;
        also
        public exceptional_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) !=
             PackageManager.PERMISSION_GRANTED;
        signals_only java.lang.SecurityException;
      @*/
        public void setActivePasswordState(int quality, int length, int letters,
                        int uppercase, int lowercase, int numbers, int symbols,
                        int nonletter) {
                mContext.enforceCallingOrSelfPermission(
                                android.Manifest.permission.BIND_DEVICE_ADMIN, null);
        }

    /*@ public normal_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) ==
             PackageManager.PERMISSION_GRANTED;
        also
        public exceptional_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) !=
             PackageManager.PERMISSION_GRANTED;
        signals_only java.lang.SecurityException;
      @*/
        public void reportFailedPasswordAttempt() {
                mContext.enforceCallingOrSelfPermission(
                                android.Manifest.permission.BIND_DEVICE_ADMIN, null);
        }

    /*@ public normal_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) ==
             PackageManager.PERMISSION_GRANTED;
        also
        public exceptional_behavior
        requires mContext.checkCallingOrSelfPermission(android.Manifest.permission.BIND_DEVICE_ADMIN) !=
             PackageManager.PERMISSION_GRANTED;
        signals_only java.lang.SecurityException;
      @*/
        public void reportSuccessfulPasswordAttempt() {
                mContext.enforceCallingOrSelfPermission(
                                android.Manifest.permission.BIND_DEVICE_ADMIN, null);
        }

    static class ActiveAdmin {
        final /*@ non_null @*/ DeviceAdminInfo info;

        ActiveAdmin(/*@ non_null @*/ DeviceAdminInfo _info) {
            info = _info;
        }

        /*@ pure @*/ int getUid() {
                return info.getActivityInfo().applicationInfo.uid;
        }
    }
}
```