ShamDroid: Gracefully Degrading Functionality in the Presence of Limited Resource Access

Lucas Brutschy
Department of Computer Science
ETH Zurich, Switzerland
lucas.brutschy@inf.ethz.ch

Pietro Ferrara
IBM Thomas J. Watson Research Center, U.S.A
pietroferrara@us.ibm.com

Omer Tripp
IBM Thomas J. Watson Research Center, U.S.A
otripp@us.ibm.com

Marco Pistoia
IBM Thomas J. Watson Research Center, U.S.A
pistoia@us.ibm.com

Abstract

Given a program whose functionality depends on access to certain external resources, we investigate the question of how to gracefully degrade functionality when a subset of those resources is unavailable.

The concrete setting motivating this problem statement is mobile applications, which rely on contextual data (e.g., device identifiers, user location and contacts, etc.) to fulfill their functionality. In particular, we focus on the Android platform, which mediates access to resources via an installation-time permission model. On the one hand, granting an app the permission to access a resource (e.g., the device ID) entails privacy threats (e.g., releasing the device ID to advertising servers). On the other hand, denying access to a resource could render the app useless (e.g., if inability to read the device ID is treated as an error state). Our goal is to specialize an existing Android app in such a way that it is disabled from accessing certain sensitive resources (or contextual data) as specified by the user, while still being able to execute functionality that does not depend on those resources.

We present SHAMDROID, a program transformation algorithm, based on specialized forms of program slicing, backwards static analysis and constraint solving, that enables the use of Android apps with partial permissions. We rigorously state the guarantees provided by SHAMDROID w.r.t. functionality maximization. We provide an evaluation over the top 500 Google Play apps and report on an extensive comparative evaluation of SHAMDROID against three other state-of-the-art solutions (APM, XPrivacy, and Google App Ops) that mediate resource access at the system (rather than app) level. SHAMDROID performs better than all of these tools by a significant margin, leading to abnormal behavior in only 1 out of 27 apps we manually investigated, compared to the other solutions, which cause crashes and abnormalities in 9 or more of the apps. This demonstrates the importance of performing app-sensitive mocking.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Program analysis

General Terms Security, Verification

Keywords Static Program Analysis, Android Applications, Privacy

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814296

1. Introduction

Software systems often rely on external resources to achieve their functionality. Though access-control policies normally govern access to sensitive resources [26], once an application is granted access to a given resource, it may utilize that resource in unintended ways. A notable example is mobile applications, which are often found to release sensitive information about the user (e.g., the user's location or date of birth) or the mobile device (e.g., the device ID) to advertising and analytics servers without the user's awareness [15, 21].

Problem statement We assume that the behavior (or execution) of an application is parameterized by the resources (or inputs) it depends on. If two runs of the program read exactly the same input values, then they are identical. We also assume that executions are comparable, forming an order according to the degree to which they exercise the application's intended functionality. As an example, a run that aborts due to an error condition exercises less functionality than a normal run through the business logic of the application. Given these assumptions, our goal is to simultaneously (i) disable access by the application to sensitive resources, as specified by the user, and (ii) retain, as best as possible, the application's functionality that depends on non-sensitive resources. Together, these two requirements amount to substituting the sensitive inputs with mock values that exhibit *maximal utility*, meaning that they can drive execution along a maximal path.

A concrete instantiation of this problem statement, serving as the motivation for this paper as well as the focus of its experimental part, is Android applications. Android is the most widespread mobile platform. Access by an Android app to sensitive resources, such as the GPS sensor or device state, is mediated by a permission system [7]. Permissions are requested, and granted, at install time. The user either grants the app all the permissions it asks for, or installation fails. This forces the user into the choice between (i) not using the app or (ii) being exposed to privacy threats.

Existing solutions One approach to the problem of ensuring user privacy is to monitor the behavior of the app and prompt the user in the event of a suspicious event. Research along this direction has led to privacy enforcement solutions based on information-flow-security analysis [15, 33], and more recently also to a classification-based approach [32] that compares between sensitive values and values arising at release points. In both cases, the monitoring system can easily be bypassed [28]. Approaches based on information-flow-security analysis also demand extensive engineering to constrain run-time overhead, which limits their accuracy and applicability.

An alternative approach, different from monitoring the behavior of the app, consists of replacing sensitive values with mock values. This overcomes both the overhead and soundness problems that online monitoring suffers from. Current realizations of this approach operate at the system level, rather than the app level [1–3, 8]. That is, the mock value is decided globally, without considering how the app depends on the given resource and what it assumes about it, which may cause the app to crash or to continue to run with unnecessarily reduced functionality.

Tapjoy As an illustration, we refer the reader to Figure 1. This code snippet is part of the Tapjoy library,[1] a widely used mobile advertising and publishing platform. When initializing its core connection component, Tapjoy reads the device identifier and performs some checks to determine if the identifier is genuine. If not, then Tapjoy creates an ad-hoc random identifier, stores it permanently, and tracks the device through it for the future executions as well.

The device identifier is important to Tapjoy, as it constitutes a persistent identity of the device/user across different apps and sessions. For this reason, Tapjoy explicitly checks whether an invalid value — in particular, `null`, the empty string, `"0"` or `"000000000000000"` — has been read. A system-wide mocking approach that is not aware of these checks, is likely to return a well-formed yet invalid device ID that would be rejected by Tapjoy. For instance, XPrivacy returns `"000000000000000"`, and this is rejected

```
String deviceID = null;
if (telephonyManager != null)
        deviceID = telephonyManager.getDeviceId();
boolean invalidDeviceID = false;
if (deviceID == null ||
        deviceID.length() == 0 ||
        deviceID.equals("000000000000000") ||
        deviceID.equals("0")) {
        TapjoyLog.e(TAPJOY_CONNECT,
                "Device id is null, empty or an emulator.");
        invalidDeviceID = true; }
else // Valid device id
        deviceID = deviceID.toLowerCase();
if (invalidDeviceID) {
        //Creates, stores and reuses a random id
}
```

Figure 1: Code snippet from method `init()` of class `com.tapjoy.TapjoyConnectCore` in the Tapjoy library

by Tapjoy. In that case, the device is tracked via a random ID, which is not persistent through software updates (e.g., version upgrades, cache reset, etc) and thus suboptimal.

Our approach We pursue a mocking approach to ensure user privacy. Unlike existing mocking solutions, however, we propose an app sensitive mocking algorithm, such that mock synthesis is governed by the particular behaviors of the subject app as well as the assumptions made by the app itself.

We focus on benign rather than malicious application. We assume the developer of the app to be largely oblivious to privacy concerns, and reflect convenient (rather than intentionally malicious) coding practices. However, even if the developer is mindful of privacy issues, developing applications such that any subset of permissions can be revoked by any user, demands prohibitive engineering effort. Our transformation algorithm proposed in this paper addresses precisely this need.

Adopting an application-specific rather than system-wide view allows us to explore the notion of maximal input utility outlined above. This translates into several challenges. First, we need to select a desirable execution path (e.g., the one skipping the `invalidDeviceId` branch in Figure 1). Second, we need to extract path conditions that depend on the input (e.g., the length and equality checks in the example). Last, we need to synthesize a mock value that meets the respective path constraints (e.g., the value `"1"` in the example). These challenges are, of course, interconnected. For example, a desirable path is tractable only if it yields path conditions that can be modeled and solved.

We present SHAMDROID, a transformation algorithm that rewrites an Android app to eliminate dependencies on sensitive resources. If successful, SHAMDROID guarantees normal execution with a built-in bias toward functionality maximiza-

[1] `http://www.tapjoy.com`

tion. Otherwise, the transformation fails. SHAMDROID is based on (i) a combined forward/backward slicing algorithm, (ii) a constraint inference based on the weakest-precondition calculus, (iii) a specialized iterative algorithm for constraint solving, and (iv) a rewriting module equipped with parametric mock libraries.

We report on extensive evaluation of SHAMDROID over the top 500 free Google Play apps in the United States. In our experiments, we compare SHAMDROID against three other approaches (APM, XPrivacy and Google APP OPS) that mediate access to permission-guarded functionality at the system level, and report on manual inspection of a systematically selected subset of 27 applications. SHAMDROID compares favorably to the three other approaches: It causes an abnormal behavior only in one out of 27 apps we manually investigated, while APM crashed in 17 cases, XPrivacy exposes abnormal behaviors in 14 cases, and App Ops in 9 cases.

Contributions This paper makes the following principal contributions:

- **Functionality maximization.** We pose the problem of maximizing residual functionality while preventing a program from accessing certain external resources it depends on (Sections 4.1 and 4.2). A concrete challenge that informs this problem definition is privacy enforcement on a mobile device. An app having the permission to access sensitive user, device or context information may manipulate or release that information in unauthorized ways. This calls for a solution whereby functionality is gracefully degraded as the app is blocked from accessing sensitive resources.
- **Algorithmic framework.** We provide a general formalization of the problem above, as well as a general algorithmic framework to address it (Sections 4.3 and 5). We state and prove properties of our framework, and discuss its theoretical guarantees and limitations.
- **Implementation and evaluation.** We describe the implementation and evaluation of SHAMDROID, a transformation algorithm that replaces sensitive-resource accesses in Android apps with mock values according to a specification (Section 6). The results of our experiments, comparing both quantitatively and qualitatively between SHAMDROID and three state-of-the-art permission mediation solutions, are highly encouraging.

2. Application-Specific Constraints

We proceed by showing a series of examples which show how applications make specific and potentially mutually exclusive assumptions about restricted resources. These examples motivate the need for an app-sensitive solution instead of existing system-level mocking approaches.

Consistency of observations As a first example, consider the component `com.paypal.android.lib.riskcomponent`, which is included in various very popular applications (such as the eBay mobile app). The purpose of the component is to generate a "Risk Blob", i.e. a collection of user data (IP address, phone numbers, location etc.) to be sent to the server to evaluate fraud risk. It is thereby a big threat to the users privacy and at the same time not essential to the functionality of the application.

```
switch (telephonyManager.getPhoneType()) {
case 1:
        riskBlob.location = (GsmCellLocation)
                telephonyManager.getCellLocation();
case 2:
        riskBlob.location = (CdmaCellLocation)
                telephonyManager.getCellLocation();
}
```

Figure 2: Simplified, deobfuscated code snipped from PayPals `RiskComponent`

Figure 2 displays a simplified and deobfuscated piece of code from this component. The application uses the harmless and unrestricted API `TelephonyManager.getPhoneType()` to determine the type of the phone first. Then, it uses the restricted API `TelephonyManager.getCellLocation()` to extract privacy-critical information from the phone. A mocking approach oblivious to the internal behavior of the application will fail to provide suitable mock values here, as it will crash when the return value is being cast to either `CdmaCellLocation` or `GsmCellLocation` (which are incomparable subtypes).

This problem is a simple representative of a variety of often complex interactions between observable properties and properties established by the mock, which may not be inconsistent to ensure correct program behavior. System-level mocking fails in these situations, while app-level mocking can adapt to the expectations of the application.

```
String str = telephonyManager.getLine1Number();
if (str != null) {
        if (str.startsWith("1") && str.length() == 11) {
                this.thisPhonesNumber = str.substring(1);
        }
}
```

Figure 3: `com.PrankRiot` requires an international format phone number

Region and formatting properties Apps are often targeted at a specific region or language and therefore rely on assumptions that are specific to that region. Consider for instance the method `TelephonyManager.getLine1Number()`, which returns the phone number of the device as a string and is guarded by the `READ_PHONE_STATE` permission. The format

```
String str1 = telephonyManager.getLine1Number();
if (str1 != null && str1.length() == 10) {
        String prefix = str1.substring(0, 3);
        // initialize the DataManager class
}
```

Figure 4: Code snippet from the method that initializes `DataManager` objects

of the returned string depends on (1) the format of phone numbers in the target region of the app and (2) the exact representation that the device choses convert that number into a string (e.g. double-zero-prefix, plus-prefix etc.). Consider now the snippet of code in Figure 3 from app `com.PrankRiot` in class `com.TapFury.Activities.CreatePrank`. The app checks if the phone number string consists of 11 characters and starts with a 1 (the international prefix for U.S. numbers). In contrast, the app `com.webascender.callerid` assumes that the phone number is made by 10 digits, and it assumes that the first three digits represent the local prefix (see Figure 4) to enable the main functionality of the app. Both these apps assumes an U.S. phone number, but they expect different formats. Therefore, when SHAMDROID revokes the `READ_PHONE_STATE` permission, we need to create app specific values, and a system solution might enable the functionality of only one of the two apps.

Cross-application interaction Using the `GET_TASKS` permission, and a call to `ActivityManager.getRunningTasks` an application reveals all activities currently running on the users device - which is a common and well-known security and privacy threat [10]. However, some applications make specific requirements about the return value of this API, making system-level mocking hard. For example, `com.oovoo` checks whether a certain activity of its video chat application is the top activity (currently displayed activity) through the above method call. A mocking approach that preserves the functionality of the application, must include the required activities class name as the top activity of the currently running tasks. For this, the an app-level mocking solution must analyze the application and observer this specific requirement.

Persistent storage Using the permissions `GET_ACCOUNTS` and `AUTHENTICATE_ACCOUNTS`, applications can list accounts and add their own accounts. Several applications require the mocking approach to present an existing "mock" account to the application to work correctly. A mock account object provided by a system-level mocking approach will not satisfy the invariants established by the account initialization code of the application. For example, the application `com.forshared` requires the account to contain two fields `totalSpace` and `freeSpace`, which are string values which must be parsable as a `Long`, for the application not to crash, as seen in Figure 5. SHAMDROID will infer these requirements and create corresponding mocks.

```
this.mTotalSpace =
        Long.valueOf(this.mAccountManager.getUserData
                (this.mAccount, "totalSpace")).longValue();
this.mFreeSpace =
        Long.valueOf(this.mAccountManager.getUserData
                (this.mAccount, "freeSpace")).longValue();
```

Figure 5: Code snippet from the method that initializes `DataManager` objects

External resources When restricting the connection to the network, apps shall not be allowed to create connections using `java.net.URL.openConnection(...)`. However, many applications require the execution to simple HTTP protocols to work correctly, even if these are not essential to the functionality. Following the REST-principle, applications often verify the correctness of their operations by observing the status code returned by the connection. While most operations expect the 200 HTTP status code, other operations require other status codes - For example, whatsapp requires the result of certain calls to be 204 (successful operation, no content). Similarly, `de.gmx.mobile.android.mail` requires the response of a HTTP request to be 201 (successful operation, content created). This also applies to other properties of HTTP responses. For example, `com.evernote.android` requires the response of an HTTP call to be produced by Apache Thrift - It checks that the content type of the response is "`application/x − thrift`".

Postcondition strengthening In addition to the previous examples, many apps make assumptions on the values returned by APIs methods which are not guaranteed by the postcondition of the method. While such assumptions are not necessarily mutually exclusive, they show that system-level solutions require a lot of manual investigation, as well as maintenance, to provide values that work with most applications. Instead, SHAMDROID investigates assumptions and generates mock values fully automatically.

We investigated various bug reports in the GitHub repository of the system-level mocking approach XPrivacy,[2] and found various reports that exposed crashes because of such assumptions on Android APIs. For example, WhatsApp assumes that a nonempty list of email accounts is returned by `AccountManager.getAccounts`, and it crashes when XPrivacy returns an empty list.[3] Analogously IM+ crashes if `AccountManager.getAccountsByType(String)` (guarded by `GET_ACCOUNTS`) returns an empty list of accounts.[4] Finally, certain value formats change across Android versions.

[2] https://github.com/M66B/XPrivacy

[3] https://github.com/M66B/XPrivacy/issues/164

[4] https://github.com/M66B/XPrivacy/issues/1604

For instance, the format of `WifiSsid` has changed across versions 4.1 and 4.2. Therefore, XPrivacy caused the crash of many apps, because, when `TelephonyManager.getNetworkOperator()` was called, it returned a mocked value that was not consistent with this new format.[5]

3. Technical Overview

In this section, we walk the reader through the main steps of our approach.

3.1 Running Examples

In addition to the Tapjoy example of Figure 1, we describe here another real-world running example. We shall refer to these two running examples throughout this paper to illustrate technical discussion points.

White Pages Current Caller ID & Block Consider now the code in Figure 4 from the Current Caller ID & Block application (`com.webascender.callerid`), one of the most popular free applications in the United States with 5M to 10M installs. This application uses the `READ_PHONE_STATE` permission to acquire the telephone number of the user and validates that it has exactly ten digits (which is true only for US numbers). In this case, it extracts the prefix of the phone number (that is, the first three digits). Only then the `DataManager` class is initialized and the application provides its functionality, blocking and identifying incoming calls.

Though it accesses private data, there is no strict need for that data for the core functionality of Current Caller ID & Block. In fact, even with a mock phone number it would be possible to identify and block incoming calls. However, the mock number has to satisfy the constraints set by the application. Also note that the 10-digit constraint on the phone number is specific to the US market. An application from a different country may enforce different restrictions on the phone number. This comes to highlight the need for an app-sensitive mocking solution.

3.2 Step I: Constraints Inference

The first step is to characterize how the app accesses resources. In theory, the app may perform full validation of the values it obtains, thereby complicating attempts to mock the actual value. In practice, however, the check is limited to certain specific local tests as illustrated in Section 3.1 on real-world examples. It should also be observed that validation tests often vary across applications. As an example, one app may verify that it is not running atop an emulated environment by ascertaining that the IMEI is not a string of `0`'s, as in the Tapjoy example, while another app may use the IMEI to validate manufacturer information, which is encoded in the IMEI prefix. In light of this observation, which we confirm experimentally in Section 6, there is the need for application-specific mock synthesis.

[5] `https://github.com/M66B/XPrivacy/issues/116`

The first goal is to collect sufficient constraints to avoid *bad executions*, wherein the application crashes or error-handling code (such as writing to the error log) is executed in place of the core functionality. Concretely, SHAMDROID considers a code path as being *bad* if it leads to a *bad program point*: a program point where (i) an exception is thrown, or (ii) a well known error-handling method is invoked. In Figure 1, `TapjoyLog.e(...)` is called, which in turn invokes the built-in `android.util.Log.e(...)` error-logging method, thereby rendering the constraints listed above — of synthesizing a mock value that is neither `null`, nor empty, nor a `0`, nor a sequence of fifteen `0`'s — necessary. This leads to infer the boxed constraints in the left part of Figure 6.

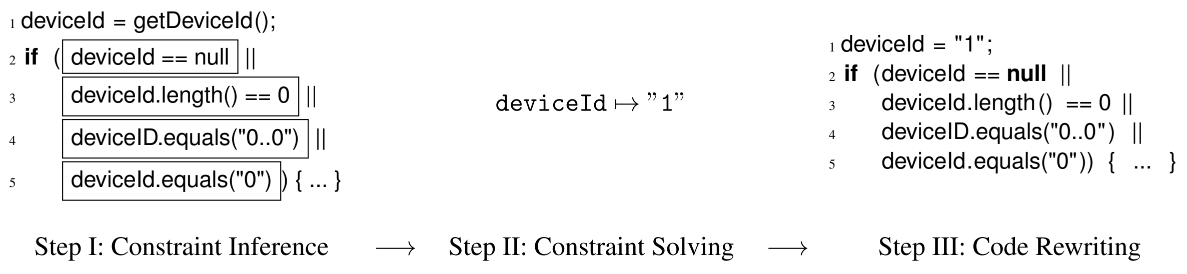
Beyond avoiding bad constraints, which is essential, SHAMDROID further attempts to synthesize a mock value that *maximizes* the core functionality; that is, while there are different possible values that guide execution away from error handling or crashing, in practice these values may have different grades of utility. Some lead to a more complete execution of the application's business logic than others.

As an illustration, we refer to Figure 4 where the phone number is retrieved via the `getLine1Number()` call. This code is solid, and there is no bad constraint since it checks if `str1` is not `null` before accessing it. However, executing lines 3-5 is noticeably preferable as discussed in Section 3.1. A concrete way of capturing this, which SHAMDROID applies, is to consider *uses* of the resource value along different code paths. Intuitively, given two paths that both execute normally, if one of the paths makes more use of the obtained value, then it is more likely to perform the actual logic that depends on that value. In Section 6.3, we demonstrate that this heuristic is highly effective. In Figure 4, `str1` is used at line 3, and so executing this branch of the `if` statement at line 2 is preferable, which implies that `str1` should not be `null` and made by ten digits.

3.3 Step II: Constraint Solving

The goal of our constraint solving is to satisfy as many as possible of the usage constraints (which capture the conditions that drive execution along a desirable path), while simultaneously refraining from violating any of the bad constraints (which ensure that bad execution paths are avoided). SHAMDROID partitions the constraints into clusters according to a relation detecting possible conflicts between constraints. A given cluster ideally contains constraints that are mutually consistent. In practice, SHAMDROID deliberately underapproximates the conflict relation to improve the performance, making it possible for a given cluster to contain contradictory constraints.

However, since the conflict relation is an underapproximation, if it states that two constraints are conflicting, then they cannot belong to the same solvable cluster. On the other hand, this relation might fail to prove conflict in some cases. Therefore, it may yield a cluster that is larger than the maximal cluster of satisfiable constraints. However, if the constraints



```
deviceId = getDeviceId();
if ( deviceId == null ||
     deviceId.length() == 0 ||
     deviceID.equals("0..0") ||
     deviceId.equals("0") ) { ... }
```

$\texttt{deviceId} \mapsto \texttt{"1"}$

```
deviceId = "1";
if (deviceId == null ||
     deviceId.length() == 0 ||
     deviceID.equals("0..0") ||
     deviceId.equals("0")) { ... }
```

Figure 6: High-level flow of the SHAMDROID System

in the cluster are consistent and a solution is obtained, this means that a maximal set of mutually consistent constraints has been discovered and solved. In this case, the mocking solution computed by the constraint solver simultaneously avoids all the bad constraints while maximizing functionality per the usage constraints (see Theorems 1 and 2). Our experimental findings, listed in Section 6, show that in over 90% of the cases SHAMDROID is indeed able to converge on a fully optimal mock implementation.

In the minority of other cases, the second-largest cluster is tried, and so on, until we obtain suboptimal (yet anomaly-avoiding) mock data.

The concrete values obtained as a solution to the constraint system usually avoids the bad program points (i.e., exception-raising and error-handling code). Back to our examples, a solution for Tapjoy that avoids the bad constraints is `"1"` as device ID (as shown in the central part of Figure 6), while for the example in Figure 4 a possible solution is a phone number made by ten 0 digits.

3.4 Step III: Code Rewriting

After collecting the constraints and finding a solution that is consistent with at least the constraints necessary to avoid bad program points, the third and final step is for SHAMDROID to impose the mock implementation on the original code. This is achieved via app-level code rewriting in the form of bytecode editing.

SHAMDROID replaces the source expression (i.e., the resource read) with another expression evaluating to a mock object based on the type of the actual value returned by the original expression. Hereby our solution is not restricted to primitive data but can handle objects and complex data structures as well. For instance, the call to `telephonyManager.getDeviceId()` at line 3 of the Tapjoy example is replaced with `"1"` (see right side of Figure 6), while `telephonyManager.getLine1Number()` at line 2 of Figure 4 is replaced by the string constant "0000000000". Example sof complex data structures SHAMDROID needs to mock are (i) the list of `LocationManager` objects returned by `TelephonyManager.getProviders()`, (ii) the `CellLocation` subtype (either `GsmCellLocation` or `CdmaCellLocation`) returned by `TelephonyManager.getCellLocation`, and (iii) the list of `NeighboringCellInfo` objects returned by `TelephonyManager.getNeighboringCellInfo()`.

4. Formal Setting

The concrete domain Σ is made out of an environment, $\mathsf{Env} : \mathsf{Var} \to \mathsf{Val}$, which relates variables to their values, and an input valuation function, $\mathsf{InpEnv}\colon \mathsf{Inp} \to \mathsf{Val}$, which relates each input identifier to a value. We thus have: $\Sigma = \mathsf{Env} \times \mathsf{InpEnv}$. Note that certain inputs (or resources) change their value over time. For example, the GPS sensor potentially returns different location reads when sampled at different points. This could potentially complicate our formal setting. Pleasingly, however, we can soundly assume that, as long as the program has not sampled the input, the value of that input has not changed. If, for instance, the program processes a location read at a given state, then a subsequent location change will have no effect on the processing done by the program, until/unless the location is read again. This observation yields a persistent notion of InpEnv.

Since in our model the only source of nondeterminism is the external input environment, our concrete semantics is a function from programs $\mathsf{p} \in \mathsf{St}$ and input valuations $\mathsf{in} \in \mathsf{InpEnv}$ to (single) traces (rather than sets of traces). A trace, as is standard, is a sequence of states $\tau \in \vec{\Sigma^{+}}$, where $\vec{\Sigma^{+}}$ denotes the set of all concrete traces. Formally, $\mathbb{S}^{\mathsf{St}}[\![\mathsf{p}, \mathsf{in}]\!] = \tau$.

Note that our theoretical model only requires deterministic per-input execution, which enables modeling nondeterminism as an external input. Further, the underlying call-graph representation is sound, accounting for all possible orders of asynchronous/concurrent execution.

4.1 Bad Executions and Functionality

The purpose of our analysis is to produce input valuations such that we (i) avoid bad executions (e.g., executions with erroneous states), and (ii) maximize the program functionality (e.g., the execution exercises the core business logic).

First of all, we suppose that an $isBad(\tau)$ predicate is provided. Given an execution trace $\tau \in \vec{\Sigma^{+}}$, it holds iff the execution is bad. We lift this predicate to input valuations by defining $isBad_{\mathsf{p}}(\mathsf{in}) \Leftrightarrow isBad(\mathbb{S}^{\mathsf{St}}[\![\mathsf{p}, \mathsf{in}]\!])$.

Then, we need to represent the *utility* of a given execution. For instance, we consider an execution that immediately quits the program (e.g., because it discovered that the device identifier it received is bogus) less useful than one going through all the main components. Given a program p, $ut_{\mathtt{p}}$: $\mathsf{InpEnv} \to \mathtt{Ut}$ returns the utility level of the input valuation in. We suppose that Ut is equipped with a total order $\leq_{\mathtt{Ut}}$. In our model, $\mathsf{u}_1 \leq_{\mathtt{Ut}} \mathsf{u}_2$ represents that level u_2 is considered more *useful* than level u_1.

Note that our approach is parametric in both $isBad$ and $ut_{\mathtt{p}}$. Section 4.3 will present the specific instances we adopt in SHAMDROID.

4.2 Property of Interest

Input-dependent bad executions We define a bad execution to be dependent on a particular input if there is another execution of the same program with a different input that is not bad. For a program p, this is modeled by predicate $badInDep_{\mathtt{p}}$, that, given an input identifier, holds iff the bad execution depends on the given input identifier, that is, if there exists a different value for the given input identifier that produces an execution that is not bad. Formally,

$$\begin{gathered} badInDep_{\mathtt{p}}(\mathtt{id}) \\ \Updownarrow \\ \exists \mathsf{in} \in \mathsf{InpEnv} : isBad_{\mathtt{p}}(\mathsf{in}) \\ \exists \mathsf{v} \in \mathsf{Val} : \neg isBad_{\mathtt{p}}(\mathsf{in}[\mathtt{id} \mapsto \mathsf{v}]) \end{gathered}$$

Input-dependent utility Different input valuations may enjoy different utility. Our approach is aimed at maximizing the functionality of the app; that is, we aim at an input valuation that maximizes utility.

First of all, we define a function that returns the maximal utility level of a program p w.r.t. the total order $\leq_{\mathtt{Ut}}$. Formally,

$$maxUtil(\mathtt{p}) = \max_{\mathsf{in} \in \mathsf{InpEnv}} ut_{\mathtt{p}}(\mathsf{in})$$

We then define how much the utility depends on a given input identifier id. Since $\leq_{\mathtt{Ut}}$ is a total order, we suppose that function $diff : (\mathtt{Ut} \times \mathtt{Ut}) \to \mathbb{Q}$ returns the difference in terms of utility between the two levels. In particular, $diff(\mathsf{u}_1, \mathsf{u}_2)$ returns how many elements of Ut are between u_1 and u_2 if u_1 is above u_2. If u_1 is equal to or below u_2, it returns zero. Relying on $diff$, we define $utilDep_{\mathtt{p}} : \mathsf{Inp} \to \mathbb{Q}$ that, given an input identifier, returns how many utility levels can be gained by choosing the *right* value for the given input identifier. Formally,

$$utilDep_{\mathtt{p}}(\mathtt{id}) = \max_{\substack{\mathsf{in} \in \mathsf{InpEnv} \\ \mathsf{v}_1, \mathsf{v}_2 \in \mathsf{Val}}} diff \begin{pmatrix} ut_{\mathtt{p}}(\mathsf{in}[\mathtt{id} \mapsto \mathsf{v}_1]), \\ ut_{\mathtt{p}}(\mathsf{in}[\mathtt{id} \mapsto \mathsf{v}_2]) \end{pmatrix}$$

Since we suppose that inputs are independent, an input valuation that maximizes the utility gain for each input identifier is also a valuation that maximizes the utility of the program. This is proved in the following lemma.

Lemma 1. *Let* p *be a program and* $\mathsf{in} \in \mathsf{InpEnv}$ *an input valuation, such that* $\forall \mathtt{id} \in \mathsf{Inp}$ *:*

$$utilDep_{\mathtt{p}}(\mathtt{id}) = \max_{\mathsf{v} \in \mathsf{Val}} diff(ut_{\mathtt{p}}(\mathsf{in}), ut_{\mathtt{p}}(\mathsf{in}[\mathtt{id} \mapsto \mathsf{v}])) \quad (1)$$

Then $ut_{\mathtt{p}}(\mathsf{in}) = maxUtil(\mathtt{p})$

Goal To summarize in terms of our formal notation, given a program p, our approach aims at finding an input valuation $\mathsf{in} \in \mathsf{InpEnv}$ such that (i) it does not produce a bad execution if it is input dependent ($\forall \mathtt{id} \in dom(\mathsf{in}) : badInDep_{\mathtt{p}}(\mathtt{id}) \Rightarrow \neg isBad_{\mathtt{p}}(\mathsf{in})$), and (ii) it maximizes functionality; that is, it produces an execution with a maximal utility level ($ut_{\mathtt{p}}(\mathsf{in}) = maxUtil(\mathtt{p})$).

4.3 Model

Until this point, we have not yet specified how bad executions and functionality are decided. Our framework may be instantiated in different ways, enabling different choices how to fix these judgments.

Bad executions In the SHAMDROID instantiation, executions are considered bad if they contain a bad program statement. That is, an execution is considered bad if it either (i) throws an exception or an error (i.e., `throws` a `Throwable`); or (ii) invokes standard error-logging APIs (`Log.w(...)`, `Log.e(...)` or `Log.wtf(...)`). Therefore, given a set of bad program points badPP, we have that:

$$isBad(\tau) \Leftrightarrow \exists \mathtt{l} \in \mathsf{badPP} : \mathtt{l} \in \tau \quad (2)$$

We consider executions such that $\neg isBad(\tau)$. In our model, this means that $\forall \mathtt{l} \in \mathsf{badPP} : \mathtt{l} \notin \tau$ by negating Equation 2.

As defined in Section 4.2, the foremost goal of SHAMDROID is to build up an input valuation such that input-dependent bad executions are avoided. Therefore, we want to compute an $\mathsf{in} \in \mathsf{InpEnv}$ such that $\forall \mathtt{id} \in dom(\mathsf{in})$:

$$badInDep_{\mathtt{p}}(\mathtt{id}) \Rightarrow \forall \mathtt{l} \in \mathsf{badPP} : \mathtt{l} \notin \mathbb{S}^{\mathsf{St}}[\![\mathtt{p}, \mathsf{in}]\!] \quad (3)$$

Running Example: In the Tapjoy code (Figure 1), the bad program point is the statement at line 9 that invokes `Log.e(...)`. SHAMDROID seeks to avoid it by finding an input that directs execution toward the `else` branch.

Functionality To quantify (or compare) functionality across different executions (or due to different inputs), SHAMDROID adopts the heuristic of statically counting *usage points*. A usage point is a statement that consumes — either directly or transitively — the value of a resource. The utility we associate with an input is then proportional to the number of usage points that the input's respective execution trace goes through. Intuitively, this means that the application has maximized its usage of the resource value (along a normal execution path), and so the likelihood that our choice of mock value has blocked any dependent functionality is minimal.

We represent by utPP the set of usage points, and a utility level by a set of usage points ($\mathtt{Ut} = \wp(\mathsf{utPP})$). The utility level of an input valuation is defined by the usage points its execution contains. Formally, $ut_{\mathtt{p}}(\mathsf{in}) = \{\mathtt{l} : \mathtt{l} \in$

$\mathbb{S}^{\mathsf{St}}[\![\mathsf{p}, \mathsf{in}]\!] \wedge \mathtt{l} \in \mathsf{utPP}\}$. A weak total order is then formed by comparing cardinalities: $\mathsf{u}_1 \leq_{\mathtt{Ut}} \mathsf{u}_2 \Leftrightarrow |\mathsf{u}_1| \leq |\mathsf{u}_2|$.

As defined in Section 4.2, the goal of SHAMDROID is to build up an input valuation that produces an execution that maximizes functionality w.r.t. all input identifiers. Therefore, we want to compute a $\mathsf{in} \in \mathsf{InpEnv}$ such that $\forall \mathtt{id} \in dom(\mathsf{in})$ we have that $ut_{\mathtt{p}}(\mathsf{in})$ is equal to $utilDep_{\mathtt{p}}(\mathtt{id})$.

Running Example: The usage points in the White Pages example (Figure 4) are lines 2 (twice), and 3, since these program points (transitively) access the input returned by `getLine1Number()`. Indeed, the execution switching into the `then` branch is more desirable, as it involves all three of these usage points.

5. Constraint Inference and Solving

In this Section, we present how our system computes an input value that (1) does not expose any input-dependent bad executions, and (2) exposes as many usage points as possible.

5.1 Parameters

To compute the mock value, the SHAMDROID algorithm has to collect constraints along different execution paths. This necessitates (i) a model of the heap, such that aliasing queries can be answered to track data flow through object fields and arrays, as well as (ii) a model of the software system's lifecycle.

SHAMDROID is parametric in the choice of heap and lifecycle models. Both are factored into the call-graph representation of the target program [19]. As such, SHAMDROID is parametric in the choice of supporting call graph.

5.2 Slicing

Since in Android inputs are read via designated method calls (e.g., `getDeviceId()`), we have well-defined slicing criteria. We compute a forward slice starting from each program point requiring a permission. From the resulting slice, we collect all program points that are marked as either *bad* or *usage*.

We then compute a backward slice starting from these points. In this way, we obtain a slice of the program that represents how a specific input may influence the execution of a bad/usage point. Given a program $\mathtt{p}$, an input identifier $\mathtt{id}$ and a program point $\mathtt{pp}$, we represent by $slice(\mathtt{p}, \mathtt{id}, \mathtt{pp})$ the function that returns this slice.

5.3 Constraint Inference

Our analysis aims at inferring constraints that are strong enough to avoid a bad program point or reach a given usage point. Therefore, given a program $\mathtt{p}$ and a program point $\mathtt{l}$, we apply a standard weakest-precondition calculus to infer the constraint c that has to be satisfied by the input of $\mathtt{p}$ to reach $\mathtt{l}$. We denote this by $wp(\mathtt{p}, \mathtt{l}) = \mathsf{c}$.

Given an input identifier $\mathtt{id} \in \mathsf{Inp}$, a program $\mathtt{p}$ and a program point $\mathtt{l}$, SHAMDROID computes

$$\mathsf{c} = wp(slice(\mathtt{p}, \mathtt{id}, \mathtt{l}), \mathtt{l}) \tag{4}$$

Since the weakest-precondition calculus [12, 14, 20] infers the weakest constraint that is sufficient in order to reach $\mathtt{l}$, we have that $\forall \mathsf{in} \in \mathsf{InpEnv}$:

$$evalC(\mathsf{c}, \mathsf{in}) \Leftrightarrow \mathtt{l} \in \mathbb{S}^{\mathsf{St}}[\![\mathsf{p}, \mathsf{in}]\!] \tag{5}$$

where $evalC$ is a function that, given a constraint over the input and an input valuation, returns `true` iff the given input valuation satisfies the given constraint.

Note that the input flowing into the program might be not only a primitive (e.g., numerical or string) value, but also a structured object (e.g., a `Location` containing latitude, longitude and altitude values). The values stored in the object might be retrieved by field accesses and method calls. Therefore, SHAMDROID adopts a symbolic approximation of the object that is later used to mock it.

Further, observe that constraints need not be collected directly for heap locations, which simplifies our weakest-precondition reasoning. The reason is that inputs stored into the heap must be read into the local state of the executing process (or thread) before being accessed (tested or used). As such, the only requirement with regard to the heap is to track flow of input values via the supporting pointer analysis, where constraints refer to environment values (i.e., local variables).

The weakest-precondition calculus is necessarily incomplete to ensure the convergence of the analysis, since the domain is possibly infinite. Therefore, when we impose bounds on the domain, we infer a constraint c', such that

$$evalC(\mathsf{c}', \mathsf{in}) \Rightarrow \mathtt{l} \in \mathbb{S}^{\mathsf{St}}[\![\mathsf{p}, \mathsf{in}]\!] \tag{6}$$

That is, a constraint c' computed for the bounded domain is stronger than the weakest constraint c, and thus strong enough to reach the given program point. On the other hand, there could exist an input valuation that leads to the given program point, but it is not covered by the inferred constraint [11].

For a bad program point, we are interested in constraints that ensure that, if satisfied, the bad program point is never executed. Dually, for a usage point, we are interested in proving that, given a constraint, we expose executions that contain that program point.

Therefore, given a program $\mathtt{p}$ and a program point $\mathtt{l}$, through our weakest-precondition calculus we obtain a constraint c that satisfies the soundness requirement of reaching usage program points as stated below.

Lemma 2. *Given program $\mathtt{p}$ and constraint c' computed by SHAMDROID for arriving at a program point $\mathtt{l}$ through an input $\mathtt{id}$, c' guarantees that $\mathtt{l}$ will always be executed.*

Beyond the soundness requirement asserted above, we state below a stronger result, guaranteeing that a bad point is not visited under the assumption that the constraint produced by the weakest-precondition calculus is complete.

Lemma 3. *Given program $\mathtt{p}$ and constraint c computed by SHAMDROID for arriving at a program point $\mathtt{l}$ through an input $\mathtt{id}$, $\neg\mathsf{c}$ guarantees that $\mathtt{l}$ will not be executed.*

According to our experiments, described in Section 6, the weakest-precondition calculus we formulated is rarely incomplete in practice for bad program points. The reason is that typically sanity checks on inputs are performed within `if` statements rather than in loops or in recursive methods (cf. Figure 1), and the same applies to implicit exceptions. In this common scenario, the weakest-precondition calculus achieves completeness.

Running Example: Consider the Tapjoy example in Figure 1. The bad program point at line 9 is reached if $\texttt{id} = \texttt{null} \vee \texttt{id.length} = 0 \vee \texttt{id} = \text{"}0\cdots0\text{"} \vee \texttt{id} = \text{"}0\text{"}$, where `id` represents the device identifier retrieved through `getDeviceId` at line 3. Negation of this constraint leads to $\texttt{id} \neq \texttt{null} \wedge \texttt{id.length} \neq 0 \wedge \texttt{id} \neq \text{"}0\cdots0\text{"} \wedge \texttt{id} \neq \text{"}0\text{"}$

Switching to Figure 4, the first usage point at line 2 ($\texttt{str1}! = \texttt{null}$) is unconstrained. Instead, the second usage ($\texttt{str1.length}() == 10$) is guarded by $\texttt{ph} \neq \texttt{null}$ (that is, the first part of the conjunct), where `ph` represents the phone number retrieved by `getLine1Number()` at line 1. Finally, the usage point at line 3 is guarded by $\texttt{ph} \neq \texttt{null} \wedge \texttt{ph.length} = 10$ (that is, the Boolean condition of the `if` statement).

5.4 Iterative Constraint Solving

Constraint solving simultaneously addresses the two goals of avoiding bad points and maximizing usage points.

Bad constraints Given an input identifier `id`, SHAMDROID computes the conjunction of all the constraints obtained to avoid bad program points that are input dependent on `id`. Formally, $\mathsf{badc} = \bigwedge_{\texttt{l}\in\mathsf{badPP}} \neg wp(\mathit{slice}(\texttt{p}, \texttt{id}, \texttt{l}), \texttt{l})$. Then, if we find an $\mathsf{in} \in \mathsf{Inp}$ such that $evalC(\mathsf{badc}, \mathsf{in})$ holds, and if the weakest precondition calculus is complete on all the slices of bad program points, we have that $\forall \texttt{l} \in \mathsf{badPP} : \texttt{l} \notin \mathbb{S}^{\mathsf{St}}[\![\texttt{p}, \mathsf{in}]\!]$ by Lemma 3, that meets the goal defined in Equation 3.

Usage constraints In order to achieve the second goal, given an input identifier `id`, for each usage point $\texttt{l} \in \mathsf{utPP}$ in our model defined in Section 4.3 we collect the constraint $\mathtt{c}^{\texttt{l}}_{\texttt{id}} = wp(\mathit{slice}(\texttt{p}, \texttt{id}, \texttt{l}), \texttt{l})$. For each $\mathtt{c}^{\texttt{l}}_{\texttt{id}}$ satisfied by an input valuation in, we know that a usage point will be executed by Lemma 2. This means that the execution is *higher* relatively to $\leq_{\texttt{Ut}}$ than an execution that does not satisfy it.

However, the conjunction $\bigwedge_{\texttt{l}\in\mathsf{utPP}} \mathtt{c}^{\texttt{l}}_{\texttt{id}}$ may not be satisfiable, since it potentially contains contradictory clauses. Following our goal of reaching as many usage points as possible, we try to find an assignment to the variables in our constraint systems, such that badc is satisfied and as many as possible of the constraints $\mathtt{c}^{\texttt{l}}_{\texttt{id}}$ are satisfied.

Though this problem is reducible to a Max-SMT instance [25], in our practical experience (and as we will discuss in Section 7), this solution does not scale up to the number of constraints we obtain from real-world apps. Therefore, in our solution, we leverage the particular structure of the inferred constraints to derive a faster, though approximate, solution

```
LastClusters = {$\emptyset$}
for ($c_1 \in$ UsageConstraints):
  NextClusters = $\emptyset$
  for ($cl \in$ LastClusters):
    NextClusters = NextClusters
      $\cup \{\{c_2 \in cl | c_1 \bowtie c_2\}, \{c_2 \in cl | c_1 \not\bowtie c_2\} \cup \{c_1\}\}$
  LastClusters = NextClusters
return LastClusters
```

Figure 7: Constraint clustering algorithm

that is often also optimal in practice. The guiding intuition is that in many cases contradictions between usage constraints can be detected straightforwardly at the syntactic level. For example, given the program

$$\texttt{if } (c) \texttt{ then } [...]^{g_1} \texttt{ else } [...]^{g_2}$$

and usage points g_1, g_2, we obtain the constraints c and $\neg c$, which are visibly contradictory.

Our algorithm, which exploits this property, is based on the idea of computing a syntactic underapproximation, $\bowtie$, of the conflict relation over constraints:

$$c_1 \bowtie c_2 \Longrightarrow \nexists \sigma . \sigma \models c_1 \wedge c_2$$

That is, the constraints are derived directly from the syntactic checks in the program. Using $\bowtie$, we define an algorithm that clusters a set of constraints accordingly, as specified in Figure 7. For instance, suppose that $\mathsf{UsageConstraints} = \{\mathtt{c}, \neg\mathtt{c}, \mathtt{c}'\}$, and that the `for` loop at line 2 iterates the constraint in this order. Then, at the first iteration we obtain $\{\emptyset, \{\mathtt{c}\}\}$, at the second $\{\emptyset, \{\mathtt{c}\}, \{\neg\mathtt{c}\}\}$, and at the third $\{\emptyset, \{\mathtt{c}\}, \{\neg\mathtt{c}\}, \{\mathtt{c}'\}, \{\mathtt{c}, \mathtt{c}'\}, \{\neg\mathtt{c}, \mathtt{c}'\}\}$.

Iterative solving Our iterative constraint solving algorithm starts from the conjunction of badc with the biggest cluster returned by the algorithm in Figure 7. If it succeeds, it returns an input valuation $\mathsf{in} \in \mathsf{InpEnv}$ that satisfies this constraint. Otherwise, it tries with a smaller cluster, and so on.

Theorem 1. *If our iterative constraint solving algorithm returns a solution* in*, and the weakest-precondition calculus is complete on all the slices of bad points, then input-dependent bad executions are avoided.*

Theorem 2. *If our iterative constraint solving algorithm returns a solution* in *at the first iteration, then* in *produces an execution with maximal functional level.*

Running Example: Given the constraints we described in Section 5.3 for the example in Figure 4, our syntactic conflict relation $\bowtie$ soundly detects that there is no conflict between the inferred constraints. Therefore, we obtain a cluster containing all the usage constraints. These are solved by SHAMDROID as the constant string "`0000000000`".

6. Experimental Evaluation

In this section, we present and discuss first a quantitative and then a qualitative evaluation of SHAMDROID. We focus in particular on the `READ_DEVICE_STATE` permission (that mediates access to device/user identifiers, in particular the IMEI) and the two location permissions `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` (that mediate access to the GPS sensor), which figure most prominently (by a far margin) in privacy violations [15, 21]. We compare SHAMDROID against the three main implementations of permission mediation in use in the Android community.

First of all, as a baseline for our evaluation, we perform simple revocation of permissions by rewriting the manifest file of an application. In our experiments, we used APM [2] for this purpose. At runtime, the Android system will emit a `SecurityException` whenever a restricted resource is used. To resume functionality, an application must handle this exception properly.

Then we compare against Google's APP OPS [3], which allows the revocation of certain permissions by returning fixed mock values. APP OPS was introduced as an hidden feature in Android 4.3, but then removed starting from Android 4.4 KitKat. In addition, APP OPS allows to revoke only few permissions, and in particular it does not allow the user to restrict accesses to the IMEI. Therefore, we can only partially compare it with SHAMDROID.

Finally, we study XPrivacy [1], which — to the best of our knowledge — is the most popular and mature Android fixed mocking solution. In fact, the free version of XPrivacy in the Android marketplace has between 100K and 500K installs and more than 4.6K evaluations, even though it requires rooting of the device. In addition, XPrivacy won the BlackDuck "2013 Open Source Rookie of the Year" award.

Unfortunately, further comparison of SHAMDROID against recently published solutions [21] could not be performed because we were unable to retrieve the corresponding experimental data[6].

In Section 6.2, we show that SHAMDROID finds solutions to both bad and usage constraints in most cases, and that XPrivacy fails to satisfy these constraints in many cases. In Section 6.3 we show that these properties result in a better experience for the user through more functional applications in a permission restricted setting.

6.1 Implementation

Figure 8 summarizes the key implementation details of the SHAMDROID system. SHAMDROID receives as input (i) an Android application as Dalvik bytecode and (ii) a permission to be revoked. The output is a transformed version of the input application that does not require the input permission. Our system consists of four pieces of functionality.

[6] [21] support only apps compatible with Android 2.1, while the apps we analyzed are compatible with Android 4.3. In addition, we were not successful in communicating with the authors to obtain the benchmarks they used.

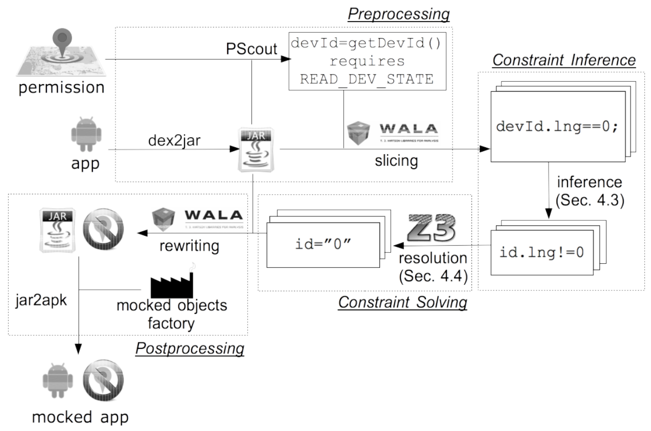


Figure 8: The architecture of SHAMDROID

In our code implementation, we have instantiated the parameters described in Section 5.1 as follows: For the call graph and pointer analysis, we utilize the type-based construction algorithm [31], which is conservative and scalable. For lifecycle modeling, we utilize the same lifecycle model as [16], which essentially treats the different event handlers (e.g., `onStart`, `onResume`, etc) as control-flow entry points.

1. Preprocessing Given the compiled Android application, SHAMDROID retargets the Dalvik bytecode to JVM bytecode with the help of the `dex2jar` tool. The resulting JAR file is then loaded into WALA [4] together with an unimplemented version of the Android API. This JAR is used during Android Development. SHAMDROID generates intra-procedural control flow graphs, and atop these a global call graph. We then use built-in WALA facilities to generate an interprocedural program dependence graph [22], which serves for slicing. Relying on the permission/API mapping provided by PScout [7], we identify the program points requiring a given permission.

2. Slicing and inference As described in Section 5.2, SHAMDROID applies WALA context-sensitive forward thin [29] and backward slicing algorithms. Each of the *usage* and *bad* program points ℓ serves as the seed of a backward analysis that computes the set of constraints that must be satisfied by the values returned by the method requiring the permission in order to reach ℓ. This is mostly an application of the assignment rule of weakest-precondition calculus, as described in Section 5.3. The backward analysis is path sensitive, and is defined as an IFDS problem [27]. In practice, SHAMDROID computes an interprocedural slice up to a fixpoint over a 0-CFA call-graph representation without any depth bound [31]. SHAMDROID records when the backward analysis has to give up in terms of completeness to ensure convergence. In this case, the user might be notified that the solution of the bad constraints might not ensure that a bad point is avoided as discussed in Section 5.3.

For improved accuracy, our backward analysis utilizes built-in models for certain Android APIs. This approach is commonly adopted when analyzing Android applications [6] since the direct analysis of these APIs is infeasible given the state of the art in static analysis, because of the intricacies of the Android implementation, in particular the frequent use of native method calls.

3. Constraint solving SHAMDROID applies the iterative constraint solving approach described in Section 5.4. The constraints are all encoded into SMT-LIB form and fed into a constraint solver. Specifically, SHAMDROID makes use of the Z3 string theory [13, 35]. It can handle (i) numerical constraints, (ii) string constraints (string equality, numerical constraints over the string length and the predicates `StartsWith` and `EndsWith`), and (iii) subtyping constraints (and in particular, checking if a given object is of a given type).

4. Postprocessing The solution of these constraints provides us with the values that can be used to replace the method calls requiring the permission. Therefore, we modify the bytecode (through WALA's SHRIKE) and consequently obtain a new JAR file. To simplify synthesis of mock values, SHAMDROID has available factory methods for different types of private values, including for instance `Location`. The factory methods expose parameters per the constraints the solver may compute (e.g., setting a particular value for the `Location` object's `longitude` or `latitude` fields). Finally, the instrumented Java bytecode is converted back to Dalvik (again, using `jar2dex`), and then injected into the original APK. We then overwrite the APK's manifest file, removing the revoked permission from the list of requested permissions. Last, SHAMDROID signs the resulting APK to enable its installation and deployment.

6.2 Quantitative Comparison

We conducted our experiments on a computer with 4 Octacore Intel Xeon E5-4627 3.30 GHz CPUs and 256GB RAM atop version 14.04 of Ubuntu Linux and the Oracle 1.7.0_59 JDK. In the experiments, we applied SHAMDROID to the top 500 free Android apps in the United States as of June 2014. We ran 32 instances of SHAMDROID in parallel.

The whole process took roughly 46 minutes in total, and 71.43 seconds per app on average. This measurement reflects the complete workflow (that is, from the initial `dex2jar` translation to the final repacking of the mocked app). We set a 30-minute timeout that was triggered only by 2 apps.

Table 1 reports the results of the analysis. Out of 500 apps, 310 apps (row apps perm.) require permission to access the location or the IMEI. We infer 126 constraints about bad program points (row bad constr., 0.4 per app). SHAMDROID was always complete in inferring them. This ensures that if we are able to solve the constraints, the bad point is not executed as proved in Lemma 3.

The experimental data confirms our hypothesis that in practice, bad points are often governed by a small, finite

Apps	500
Apps perm.	310
Bad constr.	126
Use constr.	2170
Sat. use constr.	1146
Apps w. bad constr.	60
Apps w. use constr.	156

Sol. bad. apps	58
Sol. use apps	141
Sol. bad. constr.	122
Sol. use const.	1117
XPrivacy sol. bad. apps	45
XPrivacy sol. use constr.	41

Table 1: Quantitative analysis, reporting the number of analyzed apps, bad and usage constraints, and solved constraints by SHAMDROID and XPrivacy.

set of integrity checks expressed as a (nested) conditional structure, and not inside loops or recursive calls. Therefore, applying a bounded weakest precondition calculus should have little overall effect on the precision of our approach in practice.

In addition, we infer 2170 usage constraints (row use constr., 7.0 per app), out of which 1146 constraints (row sat. use constr., 3.6 per app) form a part of a maximal cluster of satisfiable usage constraints. Many usages of the resources protected by a permission were unconstrained (e.g., they do not return a value), and therefore we obtained 60 apps with bad constraints (row apps w. bad. constr.), and 156 with usage constraints (row apps w. use constr.). SHAMDROID was able to successfully find a solution for 58 apps producing bad constraints (96.7%) solving 122 (row sol. bad. constr.) out of 126 bad constraints (96.8%), and an optimal mocked value for 141 apps producing usage constraints (90.4%) solving 1117 (row sol. use constr.) out of 1146 usage constraints (97.4%).

On average, SHAMDROID produces 4.54 different mock return values for each method guarded by a permission. For example, it generates 11 different mock device driver IDs, 10 mock phone numbers, and 7 different results for `getLastKnownLocation()`.These all point to the need for a specialized mocking approach (as opposed to system-wide mocking). In some other cases, fewer mock values are needed. For example, the `getCellLocation()` method call yields only two mock return values. These satisfy the mutually exclusive constraints shown in Figure 2. Even in this case, where only two mock values are generated, a system-wide approach is insufficient as is clear from Figure 2.

In order to compare SHAMDROID and XPrivacy, we extracted the mock values by inspecting the XPrivacy implementation [7].

We check if these values satisfy the bad constraints as well as the maximal cluster of usage constraints. As we demonstrate below, in the qualitative discussion (Section 6.3), violations of bad constraints often translate into abnormal

[7] `https://github.com/M66B/XPrivacy/blob/master/src/biz/bokhorst/xprivacy/PrivacyManager.java` Note that we did not rely on the values reported in the homepage of this GitHub repository, since these are incomplete, and sometimes not consistent with the standard values produced by the implementation

runtime behavior, if not a crash, and so this comparison is instructive. The fixed mock values produced by XPrivacy satisfy the bad constraints posed by 45 apps (row XPrivacy sol. bad. aps., *75.0% vs 96.7%* by SHAMDROID) and the maximal cluster of usage constraints posed by 41 apps (row XPrivacy sol. use constr., *26.3% vs 90.4%* by SHAMDROID). Finally, based on the performance results, we conclude that SHAMDROID is able to scale up to industrial apps.

6.3 In-depth Analysis

In our second set of experiments, we validate whether solving more constraints indeed leads to preservation of more functionality in practice. Therefore, we manually exercised a subset of the applications to detect what runtime behaviors SHAMDROID and the other permission management approaches lead to. We ran this experiment on an Asus Transformer Prime TF201 tablet.

Table 2 reports the results of this qualitative investigation. We have divided the table into two sections. The 13 apps in the upper section are those out of the top 500 Android apps in the United States for which (i) the values produced by XPrivacy do not satisfy the bad constraints inferred by SHAMDROID, and (ii) the solution produced by SHAMDROID satisfies all the bad constraints. The lower section consists of the top 100 Android applications selected with the same criteria for usage constraints, of which there are 14 in total.

Columns **Loc** and **IMEI** denote whether the application makes use of a location or the phone state permission, respectively. Here, ✓denotes that the application uses the permission in a way that triggered the inclusion of the application in this table according to the rules in the previous paragraph, while (✓) denotes all other uses of a permission. Cells marked with † use a permission in the code that is not declared in the manifest. In these cases, the permission is usually used in a library that is included in the application, and the code is either robust with respect to the unavailability of the permission or the corresponding portion of the application is dead code.

Columns **B**, **A**, **X** and **S** report the observations made with APM, APP OPS, XPrivacy, and SHAMDROID, respectively. In particular, we detect if an app does not expose any (visible) abnormal behavior (✓), it crashes (C), it is blocked (B), it slows down (S), or it leaks private information (L).

Altogether, we inspected 27 apps. In 6 of these cases (marked with N), however, we were not able to test the app comprehensively as it requires a paying account (e.g., Yahoo or AT&T) to access certain services.

Among these apps, `com.ijinshan.kbatterydoctor_en` checks the signature of the app at startup. Since both APM and SHAMDROID unpack and repack the app, it utilizes a different key to sign the final app. This app notices the signature mismatch and refuses to proceed. This issue is orthogonal to our mocking strategy. It could be fixed if we had owned the original key, or the check may be automatically removed by means of static analysis. We manually modified the app removing this check, and marked this app in Table 2 with an asterisk.

Since neither APM (which is not a mocking approach) nor AppOps (which is not able to mock the IMEI) are fully comparable with SHAMDROID, we only briefly comment on the results due to these two tools while focusing our main comparison on XPrivacy.

Comparison with APM and App Ops Almost all applications that declare one of the relevant permissions in its manifest crash after being modified by APM. Only three applications implemented correct code to handle the unavailability of a declared permission. This shows the correctness of initial claim, that most applications do not implement such code and motivates mocking approaches.

APPOPS can perform mocking of only the location. Over the 23 applications that use the location permission, 5 applications leak the location during execution, even when access to the location is restricted by APP OPS. In 4 different cases, execution slowed down significantly due to the application being stuck for a while in a loop, waiting for a location fix.

Comparison with XPrivacy The only tools in this evaluation that may revoke both the location and the phone state permissions and allow mocking of corresponding values are SHAMDROID and XPrivacy, so the rest of this section will compare in detail these two approaches.

We have then 5 cases where XPrivacy crashes, 5 cases where it leaks the location, and 3 cases where it clearly slows down execution. In all of these cases, the app produced by SHAMDROID works correctly. The only case in which the app produced by SHAMDROID exposes limited functionalities is `com.groupon`. Under SHAMDROID, this app idles at startup, whereas XPrivacy merely slows it down. Since SHAMDROID enforces partial correctness, it may produce values that cause the application not to terminate. This is an orthogonal problem compared to our specific approach, and we plan to extend SHAMDROID to avoid these cases.

In summary, in 13 cases both SHAMDROID and XPrivacy work correctly; in 13 cases SHAMDROID achieves a better result than XPrivacy (that crashes, causes slowdown or leaks the location); and in one case both XPrivacy and SHAMDROID introduce abnormal behaviors (slowdown vs the app hanging, respectively). The overall conclusion is that SHAMDROID is clearly more effective than XPrivacy in practice on real apps leading to a better user experience.

We discuss three cases in detail: (i) one case where the value produced by XPrivacy did not satisfy the bad constraints, leading the app to crash (`com.kayak.android`), and (ii) two cases that involve usage points: one where XPrivacy creates a slowdown (`com.whatsapp`) and another one where the location is still leaked (`com.wf.wellsfargomobile`).

KAYAK Flights, Hotels & Cars When we revoke access to the location and the IMEI through XPrivacy by `com.kayak.android`, this app crashes when we look for flights. In partic-

Android application	**Loc**	**IMEI**	**B**	**A**	**X**	**S**	
`com.yelp.android`	✓	-	C	L	C	✓	
`com.ijinshan.kbatterydoctor_en`	-	✓	C*	-	✓	✓*	
`com.pof.android`	✓	(✓)	C	L	✓	✓	N
`com.paypal.android.p2pmobile`	✓	(✓)	C	✓	✓	✓	
`com.myyearbook.m`	✓	(✓)	C	✓	✓	✓	
`com.mapquest.android.ace`	✓	(✓)	C	L	L	✓	
`com.qihoo.security`	(✓)	✓	C	S	✓	✓	N
`com.xfinity.playnow`	-	✓	C	-	✓	✓	N
`com.google.android.stardroid`	✓	-	✓	L	C	✓	
`com.kayak.android`	✓	(✓)	C	S	C	✓	
`com.att.android.uverse`	✓	(✓)	✓	✓	✓	✓	N
`com.activision.callofduty.mobile`	✓	-	✓	✓	✓	✓	N
`com.disney.blankvinyl.goo`	✓†	-	✓	✓	S	✓	
`com.whatsapp`	(✓)	✓	C	✓	S	✓	N
`net.zedge.android`	✓†	-	✓	✓	✓	✓	
`com.clearchannel.iheartradio.controller`	(✓)	✓	C	S	L	✓	
`com.yahoo.mobile.client.android.mail`	✓	(✓)	C	✓	✓	✓	N
`com.sgiggle.production`	(✓)	✓	C	✓	C	✓	N
`com.groupon`	✓	(✓)	C	✓	S	B	
`com.viber.voip`	(✓)	✓	C	✓	C	✓	
`com.jb.gokeyboard`	-	✓†	✓	-	✓	✓	
`com.jb.gosms`	(✓)	✓	✓	✓	✓	✓	
`com.wf.wellsfargomobile`	✓	-	✓	✓	L	✓	
`com.dianxinos.dxbs`	-	✓	✓	-	S	✓	
`net.flixster.android`	✓	(✓)	C	L	L	✓	
`com.utorrent.client`	✓†	-	✓	✓	✓	✓	
`com.fandango`	(✓)	✓	C	S	L	✓	

Table 2: Manual inspection and comparison

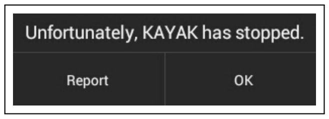
ular, when the user attempts to retrieve nearby airports (therefore accessing the location), the mock value produced by XPrivacy triggers a crash (Figure 9a). In fact, this app extracts the best location provider by calling `getBestProvider`, and XPrivacy returns `null` since it mocks available providers with an empty list. Then the app dereference the provider to check if it is equal to "`passive`", and this causes the app to crash. Instead, the value injected by SHAMDROID allows the app first to look for flights near the mock location, and then it displays an alert saying that it was unable to start the search (Figure 9b). This degradation of the functionality is unavoidable since the location is not available for contextual airport search. Still, the app does not crash, and all other functionality is retained.

WhatsApp Messenger Another behavior we noticed is a slowdown in the execution of some apps when using XPrivacy. `com.whatsapp` is one of these cases. When logging in for the first time, `whatsapp` authenticates the user as the device owner. This can be done through an SMS or — if SMS identification fails — via a phone call. When we revoke the READ_PHONE_STATE permission with XPrivacy, `whatsapp`'s server sends the SMS to the phone number used to log in, but then it waits for 5 minutes before giving up on SMS authentication (Figure 10a). In particular, this app asks for the subscriber ID, and XPrivacy returns `null`. This causes the app to manage an exceptional situation, making other attempts to send the SMS and idling until the time limit is reached. On the other hand, the app produced by SHAMDROID prevents `whatsapp` from sending the SMS repeatedly, since it creates a mock subscriber identifier, which immediately directs `whatsapp` to phone authentication (Figure 10b).

Wells Fargo Mobile Finally, in some cases we have noticed that XPrivacy leaks the exact location, while SHAMDROID does not. One of these cases is `com.wf.wellsfargomobile`. When looking for ATMs near us, XPrivacy leaks our location, which enables the result in Figure 11a. Instead, the transformed app due to SHAMDROID pops up a dialog asking to switch on the GPS capabilities. Therefore, we cannot see the ATMs that are near our position, but this is unavoidable when we revoke the location permission.

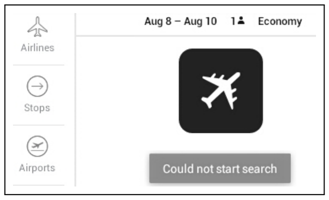
7. Related Work

A large body of work in the context of *Test Case Generation* has adopted solutions that are similar to ours (that is, constraint inference and solving) to synthesize an input value that exercises a code path that was not covered by previous test runs. These approaches usually adopt symbolic execution [24] statically [34] or dynamically [17, 30]. Similarly, Snugglebug [11] introduced a demand-driven backward symbolic analysis to find a precondition such that a goal state



Unfortunately, KAYAK has stopped.

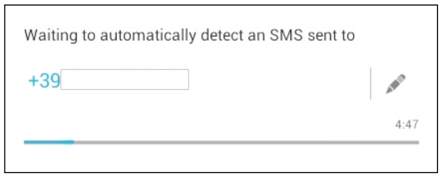
(a) XPrivacy



Could not start search

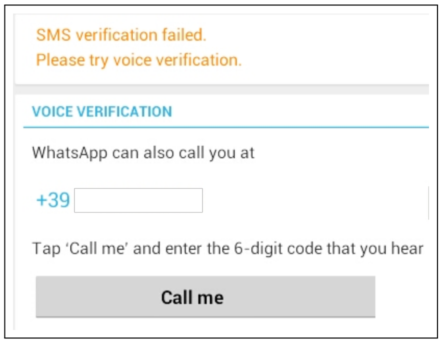
(b) SHAMDROID

Figure 9: Kayak



Waiting to automatically detect an SMS sent to

(a) XPrivacy



SMS verification failed.
Please try voice verification.

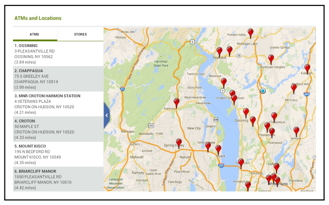
VOICE VERIFICATION

WhatsApp can also call you at

Tap 'Call me' and enter the 6-digit code that you hear

(b) SHAMDROID

Figure 10: WhatsApp



ATMs and Locations

1. OSSINING
3 PLEASANTVILLE RD
OSSINING, NY 10562
(3.84 miles)

2. CHAPPAQUA
75 S GREELEY AVE
CHAPPAQUA, NY 10514
(3.99 miles)

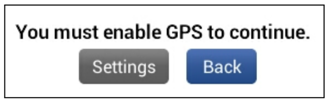
3. MNR CROTON HARMON STATION
4 VETERANS PLAZA
CROTON ON HUDSON, NY 10520
(4.21 miles)

4. CROTON
50 MAPLE ST
CROTON ON HUDSON, NY 10520
(4.33 miles)

5. MOUNT KISCO
195 N BEDFORD RD
MOUNT KISCO, NY 10549
(4.35 miles)

6. BRIARCLIFF MANOR
1050 PLEASANTVILLE RD
BRIARCLIFF MANOR, NY 10510
(4.42 miles)

(a) XPrivacy



You must enable GPS to continue.

(b) SHAMDROID

Figure 11: Wells Fargo

is reached. While our analysis can be seen as an instance of bounded static symbolic execution, our main contribution is in maximizing the functionality of the program.

Groce *et al.* [18], given a counterexample trace exposing an error in a program, produced an execution as similar as possible to the counterexample. Similarly to our approach, it applied constraint solving to produce an optimal input.

Our iterative constraint solving is reducible to a Max-SMT problem [25], where the constraints about bad locations have the weight ∞, and each usage constraints has weight 1. Max-SMT then encodes this system into SMT, computing an upper bound on the sum of weights of non-satisfied clauses. This upper bound is iteratively strengthened until an optimal solution is determined. However, our practical experience shows that this approach does not scale up to the number of constraints inferred by SHAMDROID.

As discussed in Section 1, many approaches based on information-flow ensure user privacy statically [6, 33] or dynamically [15, 32]. Instead of monitoring the flow of sensitive data inside the program, we replace sensitive values with mock values. In this context, many recent works proposed various solutions for Android apps. Like SHAMDROID, they usually replace the data coming from some APIs requiring permissions (e.g., `Location`) with some mock values. However, these tools adopt a system level mocking strategy like XPrivacy, rather than an app specific approach like SHAMDROID. Our experimental results in Section 6 showed that an app specific solution is more effective in practice than system level solutions.

MockDroid [9] was the first tool introduced in this area. When a denied resource is accessed, this tool returns a fixed value (e.g., a constant value for the device identifier), or simulates that the resource is not available (e.g., by always time outing Internet sockets). Similarly, AppFence [21] applies a fixed mocking strategy and blocks network communications, while TISSA [5] supports various mocking strategies that can be manually chosen by the user. The authors observed that it is not possible to define an effective fixed mocking strategy for any app. This motivates the need of the app specific mocking strategy of SHAMDROID.

Finally, Dr. Android and Mr. Hide [23] and AppGuard [8] are two tools that rewrite the app to produce fixed mock values instead of modifying the operating system like SHAMDROID. For this reason, they do not require to jailbreak the device.

8. Conclusion

In this paper, we introduced a novel approach to create mock data when a resource is denied such that (i) bad executions (e.g., runtime errors) are avoided, and (ii) functionality is preserved as much as possible. We first infer the constraints aimed at avoiding bad executions and maximizing functionality, and then we apply an iterative constraint solving algorithm producing an input valuation satisfying the constraints.

This theoretical approach has been instantiated to Android applications to revoke the access to some sensitive resources (e.g., device identifier and location). This app-sensitive mocking strategy has been implemented in SHAMDROID. Our experimental results show that it substantially improves the app's functionality with respect to state-of-the-art tools that apply a fixed mock strategy.

References

[1] XPrivacy. http://www.xprivacy.eu/.

[2] Advanced permission manager. https://play.google.com/store/apps/details?id=com.gmail.heagoo.pmaster.

[3] App ops brings granular permissions control to android 4.3. http://www.xda-developers.com/app-ops-brings-granular-permissions-control-to-android-4-3.

[4] Watson libraries for analysis (wala). https://github.com/wala/WALA.

[5] Taming information-stealing smartphone applications (on android). In J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, editors, *Proceedings of TRUST '11*, pages 93–107. Springer, 2011. .

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of PLDI '14*, pages 259–269, 2014. .

[7] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the android permission specification. In *Proceedings of CCS '12*, pages 217–228. ACM, 2012. .

[8] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In *Proceedings of TACAS '13*, pages 543–548. Springer, 2013. .

[9] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of HotMobile '11*, pages 49–54. ACM, 2011. .

[10] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[11] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of PLDI '09*, pages 363–374. ACM, 2009. .

[12] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002. .

[13] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of TACAS '08*, pages 337–340. Springer, 2008. .

[14] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. .

[15] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI '10*, pages 393–407. USENIX, 2010.

[16] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, CS-TR-4991, Department of Computer Science, University of Maryland, 20o9.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of PLDI '05*, pages 213–223. ACM, 2005. .

[18] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, 2006. .

[19] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, (6), Nov. 2001. ISSN 0164-0925.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. .

[21] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of CCS '11*, pages 639–652. ACM, 2011. .

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of PLDI '88*, pages 35–46. ACM, 1988. .

[23] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained permissions in android applications. In *Proceedings of SPSM '12*, pages 3–14. ACM, 2012. .

[24] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. .

[25] R. Nieuwenhuis and A. Oliveras. On SAT modulo theories and optimization problems. In *Proceedings of SAT '06*, pages 156–169. Springer, 2006. .

[26] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *Proceedings of the Symposium on Security and Privacy '07*, pages 149–163. IEEE, 2007.

[27] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of POPL '95*, pages 49–61. ACM, 1995. .

[28] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kâafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *Proceedings of SECRYPT '13*, pages 461–468, 2013.

[29] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. *SIGPLAN Not.*, 42(6):112–122, June 2007. .

[30] N. Tillmann and J. De Halleux. Pex–white box test generation for .net. In *Proceedings of TAP '08*, pages 134–153. Springer, 2008. .

[31] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of OOPSLA*, pages 281–293, 2000.

[32] O. Tripp and J. Rubin. A bayesian approach to privacy enforcement in smartphones. In *Proceedings of USENIX Security '14*, pages 175–190. USENIX, 2014.

[33] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of PLDI '09*, pages 87–97. ACM, 2009. .

[34] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of ISSTA '04*, pages 97–107. ACM, 2004. .

[35] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *Proceedings of FSE '13*, pages 114–124. ACM, 2013. .

A. Formal Proofs

Lemma 4. *Let* $\mathtt{p}$ *be a program and* $\mathsf{in} \in \mathsf{InpEnv}$ *an input valuation, such that* $\forall \mathtt{id} \in \mathsf{Inp}$ *:*

$$utilDep_{\mathtt{p}}(\mathtt{id}) = \max_{\mathsf{v} \in \mathsf{Val}} diff(ut_{\mathtt{p}}(\mathsf{in}), ut_{\mathtt{p}}(\mathsf{in}[\mathtt{id} \mapsto \mathsf{v}])) \quad (7)$$

Then $ut_{\mathtt{p}}(\mathsf{in}) = maxUtil(\mathtt{p})$

Proof. This is proved by induction on the input identifiers contained in Inp.

Base Case We assume that $|\mathsf{Inp}| = 1$. So Inp is a singleton, that is, $\mathsf{Inp} = \{\mathtt{id}\}$. By hypothesis (Equation 7), we have that in is such that

$$utilDep_{\mathtt{p}}(\mathtt{id}) = \max_{\mathsf{v} \in \mathsf{Val}} diff(ut_{\mathtt{p}}(\mathsf{in}), ut_{\mathtt{p}}(\mathsf{in}[\mathtt{id} \mapsto \mathsf{v}]))$$

By definition of $diff$, this means that in maximizes the utility w.r.t. $\mathtt{id}$. Since $\mathtt{id}$ is the only input identifier, this means that in maximizes the overall utility of program $\mathtt{p}$, that is, $ut_{\mathtt{p}}(\mathsf{in}) = \max_{\mathsf{in}' \in \mathsf{InpEnv}} ut_{\mathtt{p}}(\mathsf{in}')$. Then, by definition of $maxUtil$, we have that $ut_{\mathtt{p}}(\mathsf{in}) = maxUtil(\mathtt{p})$.

Inductive Step We assume that $|\mathsf{Inp}| = n$, and that the lemma holds for $|\mathsf{Inp}'| = n - 1$ where $\mathsf{Inp}' = \mathsf{Inp} \setminus \{\mathtt{id}_1\}$. Therefore, given a in such that $\forall \mathtt{id} \in \mathsf{Inp} : utilDep_{\mathtt{p}}(\mathtt{id}) = \max_{\mathsf{v} \in \mathsf{Val}} diff(ut_{\mathtt{p}}(\mathsf{in}), ut_{\mathtt{p}}(\mathsf{in}[\mathtt{id} \mapsto \mathsf{v}]))$ by inductive hypothesis we have that

$$ut_{\mathtt{p}}(\mathsf{in}) = maxUtil(\mathtt{p}) \quad (8)$$

when considering all the input identifiers in Inp', that is, all the input identifiers in Inp except $\mathtt{id}_1$. By Equation 7 we have then that

$$utilDep_{\mathtt{p}}(\mathtt{id}_1) = \max_{\mathsf{v} \in \mathsf{Val}} diff(ut_{\mathtt{p}}(\mathsf{in}), ut_{\mathtt{p}}(\mathsf{in}[\mathtt{id}_1 \mapsto \mathsf{v}])) \quad (9)$$

By definition of $diff$, this means that in maximizes the utility w.r.t. $\mathtt{id}_1$ as well. Since by inductive hypothesis (Equation 8) in maximizes the functionality for all the identifiers except $\mathtt{id}_1$, and by Equation 9 in maximizes the functionality w.r.t. $\mathtt{id}_1$, this means that in maximizes the overall utility of program $\mathtt{p}$, that is, $ut_{\mathtt{p}}(\mathsf{in}) = \max_{\mathsf{in}' \in \mathsf{InpEnv}} ut_{\mathtt{p}}(\mathsf{in}')$. Then, by definition of $maxUtil$, we have that $ut_{\mathtt{p}}(\mathsf{in}) = maxUtil(\mathtt{p})$. □

Lemma 5. *Given program* $\mathtt{p}$ *and constraint* c' *computed by* SHAMDROID *for arriving at a program point* $\mathtt{l}$ *through an input* $\mathtt{id}$*,* c' *guarantees that* $\mathtt{l}$ *will always be executed.*

Proof. Since SHAMDROID computes what is described by Equation 4, then by Equation 6 we have that

$$evalC(\mathsf{c}', \mathsf{in}) \Leftarrow \mathtt{l} \in \mathbb{S}^{\mathsf{St}}[\![\mathtt{p}, \mathsf{in}]\!] \quad (10)$$

By contraposition and Equation 10, we obtain that

$$\neg evalC(\mathsf{c}', \mathsf{in}) \Rightarrow \mathtt{l} \notin \mathbb{S}^{\mathsf{St}}[\![\mathtt{p}, \mathsf{in}]\!] \quad (11)$$

By definition of $evalC$, we have that

$$\neg evalC(\mathsf{c}', \mathsf{in}) \Leftrightarrow evalC(\neg \mathsf{c}', \mathsf{in}) \quad (12)$$

By combining Equations 11 and 12, we obtain

$$evalC(\neg \mathsf{c}', \mathsf{in}) \Rightarrow \mathtt{l} \notin \mathbb{S}^{\mathsf{St}}[\![\mathtt{p}, \mathsf{in}]\!] \quad (13)$$

Therefore, the satisfaction of $\neg \mathsf{c}'$ guarantees that $\mathtt{l}$ is not executed. □

Lemma 6. *Given program* $\mathtt{p}$ *and constraint* c *computed by* SHAMDROID *for arriving at a program point* $\mathtt{l}$ *through an input* $\mathtt{id}$*,* $\neg\mathsf{c}$ *guarantees that* $\mathtt{l}$ *will not be executed.*

Proof. By negation of Equation 5, we have that

$$\neg evalC(\mathsf{c}, \mathsf{in}) \Leftrightarrow \mathtt{l} \notin \mathbb{S}^{\mathsf{St}}[\![\mathtt{p}, \mathsf{in}]\!] \quad (14)$$

By definition of $evalC$, we have that

$$\neg evalC(\mathsf{c}, \mathsf{in}) \Leftrightarrow evalC(\neg \mathsf{c}, \mathsf{in}) \quad (15)$$

By Equations 14 and 15 we have that

$$evalC(\neg \mathsf{c}, \mathsf{in}) \Leftrightarrow \mathtt{l} \notin \mathbb{S}^{\mathsf{St}}[\![\mathtt{p}, \mathsf{in}]\!] \quad (16)$$

That is, $\neg\mathsf{c}$ guarantees that $\mathtt{l}$ is not executed. □

Theorem 3. *If our iterative constraint solving algorithm returns a solution* in*, and the weakest-precondition calculus is complete on all the slices of bad points, then input-dependent bad executions are avoided.*

Proof. Our iterative constraint solving algorithm always tries to solve a constraint c that is the conjunction of $\mathtt{badc}$ with some other constraints produced by the algorithm in Figure 7. Since at each iteration of the constraint solving algorithm the constraint c is the conjunction of $\mathtt{badc}$ with something else, then by Lemma 3 we know that any bad program point is not reached if the weakest-precondition calculus is complete on all the slices. Since these constraints cover all bad program points, if a solution in is found, this will avoid input-dependent bad execution. □

Theorem 4. *If our iterative constraint solving algorithm returns a solution* in *at the first iteration, then* in *produces an execution with maximal functional level.*

Proof. If our iterative constraint solving algorithm succeeds during the first iteration, the solution in satisfies for any usage points $\mathtt{l}$ all the usage constraints $\mathsf{c}^{\mathtt{l}}_{\mathtt{id}}$ that are part of the biggest cluster that does not contain contradictions. By Lemma 2 this implies that all the usage points are executed when starting with input valuation in . Therefore, by Lemma 1 the utility level of the produced execution is maximal. The fact that $\bowtie$ is a syntactic underapproximation of the conflict relations guarantees that the biggest cluster is an overapproximation of the maximal cluster of satisfiable usage constraints. Therefore, the solution in produces an execution with maximal functional level. □