Semi-Automated Discovery of Server-Based Information Oversharing Vulnerabilities in Android Applications

William Koch
Boston University
USA
wfkoch@bu.edu

Abdelberi Chaabane
Northeastern University
USA
3abdou@ccs.neu.edu

Manuel Egele
Boston University
USA
megele@bu.edu

William Robertson
Northeastern University
USA
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
USA
ek@ccs.neu.edu

ABSTRACT

Modern applications are often split into separate client and server tiers that communicate via message passing over the network. One well-understood threat to privacy for such applications is the leakage of sensitive user information either in transit or at the server. In response, an array of defensive techniques have been developed to identify or block unintended or malicious information leakage. However, prior work has primarily considered privacy leaks originating at the client directed at the server, while leakage in the reverse direction – *from the server to the client* – is comparatively under-studied. The question of whether and to what degree this leakage constitutes a threat remains an open question. We answer this question in the affirmative with Hush, a technique for semi-automatically identifying Server-based InFormation OvershariNg (SIFON) vulnerabilities in multi-tier applications. In particular, the technique detects SIFON vulnerabilities using a heuristic that *overshared* sensitive information from server-side APIs will not be displayed by the application's user interface. The technique first performs a scalable static program analysis to screen applications for potential vulnerabilities, and then attempts to confirm these candidates as true vulnerabilities with a partially-automated dynamic analysis. Our evaluation over a large corpus of Android applications demonstrates the effectiveness of the technique by discovering several previously-unknown SIFON vulnerabilities in eight applications.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**;

KEYWORDS

information leakage, software analysis, Android testing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA'17, July 2017, Santa Barbara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092708

ACM Reference format:
William Koch, Abdelberi Chaabane, Manuel Egele, William Robertson, and Engin Kirda. 2017. Semi-Automated Discovery of Server-Based Information Oversharing Vulnerabilities in Android Applications. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
https://doi.org/10.1145/3092703.3092708

1 INTRODUCTION

Modern mobile applications (apps) typically employ a multi-tier architecture. This has been largely due to the explosive growth of cloud computing platforms, such as Amazon AWS, Microsoft Azure and Heroku, allowing developers to conveniently manage and operate scalable web services [1]. As a result, apps can provide rich user experiences and are no longer limited by client-device hardware. In such settings, the cloud essentially provides an extension of the client's computation and data storage capabilities.

This application architecture often results in sensitive information flows from user devices to centralized server-side logic and storage tiers in the cloud. Users place trust in the app to securely transfer and store their sensitive information. Unfortunately, despite the benefits of the multi-tier architectures, the decoupled nature of the tiers opens up new security and privacy concerns for the app that would not have occurred if the app was self-contained.

Much research has investigated ways to detect and prevent leakage of sensitive user information. Prior work has identified advertising libraries that exfiltrate sensitive user data from mobile devices [2]. Malware has also been observed to send sensitive user data to command and control (C&C) servers [3–5]. Furthermore, client-side data leakages are exacerbated by access control and permission vulnerabilities in Android [6]. To address these issues, many extensions to Android have been proposed to enhance its security and prevent data leakages [7–9]. Additionally, side-channel information leaks such as timing and size of requests, can reveal insights about a user's online activities, even over an encrypted channel [10, 11]. However, existing research primarily considers information leakage from the client to the server.

In this work, we ask the question of whether sensitive information leakage can occur in the *reverse* direction, namely from the *server* to the *client*. To answer this question, we devise an approach that semi-automatically identifies instances of Server-based InFormation OvershariNg (SIFON) vulnerabilities. In particular, we focus on detecting SIFON vulnerabilities by identifying apps that perform client side access control using the heuristic that *overshared* information from server-side APIs will not be displayed by the app's user interface.

We built a prototype for this approach called Hush for the Android platform, and evaluated it over a large corpus of 31,559 Android apps drawn from the Google Play Store. Our evaluation demonstrates that SIFON vulnerabilities indeed exist in the wild, exposing sensitive user or corporate information to adversaries or market competitors. These vulnerabilities arise from missing or invalid access control policies on an app's cloud back-end. Essentially, SIFON vulnerabilities manifest if web services overshare data while access control is pushed to and enforced by the *client* instead of the server. We have reported our findings to the developers of eight apps we studied and have received confirmation from two of them.

In a culture of increasingly rapid development cycles [12], Hush would provide independent and enterprise development teams a tool to identify and address SIFON vulnerabilities before vulnerable services are deployed in the wild.

To summarize, the main contributions of this paper are the following.

- We introduce a novel class of Server-based InFormation OvershariNg (SIFON) vulnerabilities.
- We propose an approach called Hush that leverages both static and dynamic program analysis techniques to confirm the existence of SIFON vulnerabilities in Android apps.
- We develop a prototype implementation of Hush and evaluate it over a large corpus of Android apps drawn from the Google Play Store. Our evaluation demonstrates that SIFON vulnerabilities exist in real apps, manifesting as serious leakages of sensitive user or proprietary corporate information.

2 BACKGROUND

The rise of cloud computing has led to the predominance of a multi-tier application architecture for modern applications, especially in the web and mobile domains. In this architecture, applications are split into several tiers, where each tier is responsible for a clearly defined set of tasks and communicates with other tiers through message passing. One example of such a multi-tier application is a typical mobile app, where the app implements the user interface and some portion of the application logic, but also invokes cloud-based services and consumes the results. These services are often web-based – i.e., they are invoked via HTTP(S) requests, and the data is returned in the form of JSON, XML, Google Protocol Buffers, or similar format.

To illustrate this, we introduce a simplified running example of a mobile social networking app authored in Java for the Android platform. A central concept in this app is that of a user profile that has multiple representations within the cloud, on the wire, and within the app itself. To load a user profile, the app invokes a cloud-based web service using HTTP(S) to request a user profile with a given identifier. The cloud service returns a JSON message that represents this user, shown in Listing 1.

```
{"firstName": "Donald",
 "lastName": "Knuth",
 "email": "donald.k@spambox.us"}
```

Listing 1: JSON representation of a user profile.

To convert this representation into a form that can be computed on more easily, the mobile app *deserializes* the user profile into a Java object, referred to as the model. This typically happens with the help of a serialization library (e.g., GSON) specifically designed for this purpose. An example of this deserialization process is shown in Listings 2 and 3.

```
public class Profile {
  public String firstName;
  public String lastName;
  public String email; }
```

Listing 2: Java model of a user profile.

```
InputStream is = getProfileInputStream();
Reader r = new InputStreamReader(is);
Profile p = new Gson().fromJson(r, Profile.class);
```

Listing 3: Deserialization of a user profile using the GSON library.

After the JSON message has been deserialized into an instance of the `Profile` class, the app is ready to present the details to the user. However, the developer might have recognized that the email is sensitive information and should not be displayed. Instead of updating the cloud service endpoint to only return safe information, the developer chose to implement local access control by only displaying the first and last name of the user in the UI (Listing 4).

```
tv0.setText("First Name:");
tv1.setText(p.firstName);
tv2.setText("Last Name:");
tv3.setText(p.lastName);
```

Listing 4: User interface code to display the safe elements of a deserialized user profile.

This example contains the essential elements of a *server-based information oversharing*, or SIFON, vulnerability. The user that owns this profile has released this sensitive information to the social networking app's cloud back-end under the assumption that it would be properly handled. However, while the developers have made an effort to prevent the release of this information by sanitizing the mobile app's user interface, the cloud service API is nevertheless releasing this information to what should be considered an untrusted client device. We note that client-side defenses such as the Android permissions system, taint tracking, or information flow control would not prevent this vulnerability or its exploitation. As we show in Section 6, taint tracking can be used to *identify* the vulnerability, however the problem originates from improper implementation of access control on the server and therefore defense cannot be acheived client-side.

Unfortunately, this simplified example is not entirely fictitious. App A, is a dating app that contains a similar SIFON vulnerability as the simplified example in which we discovered in our analysis.

The app developers considered the first and last name, email address, and date of birth of a user's profile to be sensitive information and thus do not display it in the UI to other unrelated users. The SIFON vulnerability exists because the sensitive information is sent to all users who request a profile indiscriminately. If the server selectively provided sensitive information only to users who have a relationship (e.g., friends on the social network) with the data-owner, no vulnerability would exist. Our goal with this work is to develop an analysis that can identify instances of SIFON vulnerabilities. In the remainder of the paper, we present an analysis and a prototype implementation to that end.

3 THREAT MODEL

SIFON vulnerabilities are a consequence of implementation flaws in the (server-) application code that supports client applications. That is, properly implemented access controls on the server side would prevent SIFON vulnerabilities. Thus, our threat model does not consider attackers who have the capability to circumvent proper access controls on the server side.

Instead, we consider an attacker who is looking to collect sensitive information without targeting a specific victim, and operates under a budget in terms of money and time. Thus, our attacker model is opportunistic in the sense that the attacker can afford to spend some time to try to siphon sensitive information off an online service. However, the budget restriction incentivizes the attacker to shift focus to other targets if the currently analyzed service does not yield any sensitive information.

We assume targeted services and their accompanying client apps are benign but might be vulnerable. Furthermore, we assume that the attacker can create a regular (i.e., unprivileged) user account on the targeted service if necessary, but has not engaged in social engineering or otherwise duped any victim into disclosing or making available their private data. For instance, social network applications, such as Facebook or App A, provide users with the functionality to establish friendships and connected users can access each other's sensitive information. In this paper we only consider SIFON vulnerabilities where an entirely unrelated attacker can access sensitive information of his victims, because the backend server shares this information indiscriminately.

Our interpretation of what constitutes sensitive information is based on Trend Micro's analysis of the Privacy Rights Clearinghouse (PRC)'s Data Breaches database [13]. In this context, sensitive data encompasses personally identifiable information (PII), financial data, health data, education data, payment cards, and credentials.

4 SYSTEM OVERVIEW

We developed an approach for detecting SIFON vulnerabilities on the Android platform that are due to cloud service API oversharing called Hush. The goal of the approach is to detect SIFON vulnerabilities in a scalable manner, and is structured as a three-stage analysis pipeline:*(i)* preprocessing, *(ii)* static vulnerability candidate detection (S-Hush), and *(iii)* dynamic confirmation of vulnerabilities (D-Hush). An overview of Hush is shown in Figure 1.

Preprocessing The goal of the preprocessing stage is to triage applications submitted for analysis and gather initial information for input to the subsequent static and dynamic analysis stages.

S-Hush The static analysis stage serves as a scalable, automated triage phase to determine whether any sensitive data obtained from the invocation of a cloud service API is hidden from a user (i.e., not displayed in the user interface). The static analysis allows for the efficient identification of apps that should be forwarded to the subsequent dynamic analysis for confirmation.

This static analysis is implemented as a multi-stage data flow analysis in which the output of the first stage determines the configuration for the second. The first stage identifies data flows from program points that receive data from the network (i.e., sources) to points where that data is deserialized into a Java object (i.e., sinks). Then, the second stage identifies flows from these deserialization points (sources) to user interface elements (sinks). Note that the sinks from the first stage become sources for the second stage analysis. If particular deserialized object fields are hidden – i.e., they *do not* flow to an UI element – then this data is considered to be a potential instance of cloud service oversharing. These oversharing instances are categorized according to whether they are sensitive or not. If deserialized object fields are hidden and considered sensitive, then these fields are labeled as candidate SIFON vulnerabilities and the app is forwarded to D-Hush for confirmation.

The reason that the static analysis alone is not sufficient to directly declare hidden, sensitive deserialized object fields as SIFON vulnerabilities is three-fold. First, the static analysis only reports *possible* flows, but these flows might not necessarily occur at runtime. Second, it is impossible for the static analysis to determine whether the cloud service actually returns data to populate the potentially vulnerable object fields. If no data is actually returned, then the fields will be empty, and no vulnerability will exist in practice. Third, the static analysis is oblivious to the apps functionality and intent. It is possible the hidden data may be necessarily for the apps functionality and given the context in the app may not be considered sensitive. For example, hidden GPS data could be used to position restaurant locations on a map in one app, while unintentionally leaking a users location in another app resulting in a SIFON vulnerability.

D-Hush The dynamic analysis stage takes as input an app that contains candidate SIFON vulnerabilities and a set of methods to hook as identified by the preprocessing module. The app is instrumented at these methods to dynamically track information flows at runtime from deserialized messages to the user interface. The app is then executed and manually explored by an analyst. During this exploration, D-Hush captures how information flows into model object fields and how it is accessed there. If there is no access to sensitive model object fields, D-Hush reports a confirmed SIFON vulnerability.

We note that achieving high dynamic coverage of GUI-based applications with automated tools has proven to be a challenging task. While significant research progress has been made on this front, triggering advanced application functionality requires complex inputs that are currently beyond the reach of any automated tool [14]. However, future improvements in this fundamental enabling capability could be easily adopted by our approach. In the following sections, we describe the technical details of the preprocessing, S-Hush, and D-Hush analysis stages.

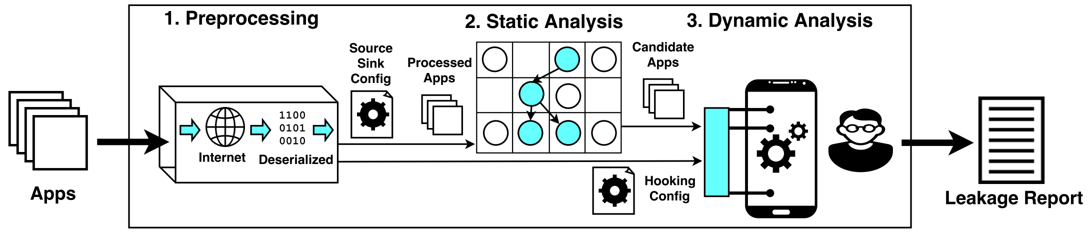


Figure 1: Overview of the Hush analysis pipeline. Android apps are preprocessed to handle program obfuscation and cull apps that cannot contain SIFON vulnerabilities. Candidate SIFON vulnerabilities are identified in a static analysis stage, and a subsequent dynamic analysis stage confirms or rejects candidate vulnerabilities.

5 PREPROCESSING

The first stage of the Hush analysis pipeline begins by extracting information about the app packages and method signatures. This functionality allows us to filter out apps that we consider not susceptible to SIFON vulnerabilities. Additionally it also generates inputs for both S-Hush and D-Hush. In particular, the triage step discards all apps that do not request the INTERNET permission (i.e., they cannot invoke cloud services over the network) or do not contain a (known) serialization library.

S-Hush and D-Hush require the identification of methods from three categories: those that invoke network communication APIs, those that deserialize network data, and those that manipulate the user interface. The preprocessing stage, thus, first disassembles the app and then performs a lightweight static analysis to extract this information.

One important challenge in this respect is that some apps are obfuscated which can prevent the automated identification of its methods. While generically deobfuscating apps can be challenging, Hush only needs to address the obfuscation of deserialization libraries. As network sources and UI sinks are provided and implemented by the Android framework, an app developer cannot easily obfuscate these methods.

As Android apps are easy to decompile, and reverse engineer, application repackaging represents a serious problem [15–17]. To counteract such techniques, the Google Android Team included the ProGuard [18] obfuscation tool in the Android Developer Tools (ADT). Moreover, in recent versions of the ADT, ProGuard is enabled by default. One of the obfuscation strategies employed by ProGuard transforms method names and rewrites call sites to use the transformed names. For example, ProGuard might convert `com.google.gson.Gson.fromJson` to `com.a.b.j.a`. As ProGuard needs to transform call sites and target methods synchronously, this approach cannot be used to obfuscate method calls to framework-provided APIs (e.g., network and UI APIs). Therefore, the preprocessor must incorporate a deobfuscation step to reverse transformations such as those performed by ProGuard. We report on our heuristics-based implementation in Section 8.1. The final output of this module is the set of sinks for S-Hush and the set of functions to hook and their category for D-Hush.

6 S-HUSH

S-Hush is a fully automated, scalable static analysis to identify data flows that represent potential SIFON vulnerabilities in Android apps. At its core, S-Hush is a two-stage data-flow analysis that *(i)* identifies flows from program points that receive data from the network to deserialization of that data into a model, and *(ii)* identifies flows from individual fields of a deserialized model into user interface elements. The output of the analysis is a list of deserialized model fields that do not appear in UI elements. These elements are subject to a classification step that heuristically infers whether individual fields are likely to contain sensitive information. Hidden and likely sensitive deserialized model fields are then forwarded to D-Hush as candidate SIFON vulnerabilities for confirmation. In the following, we elaborate on each of these steps of the static analysis.

6.1 Model Deserialization

The first step of the static analysis takes as input an app to analyze, a precomputed database of standard Android API methods that receive data from the network (*sources*), and the list of call sites for deserialization libraries found by the preprocessing stage (*sinks*). The goal of this step is to identify flows of data received from the network to deserialization points.

As a precursor, the app is disassembled to recover its Dalvik bytecode representation. A class hierarchy analysis (CHA), control flow analysis (CFA), and call graph extraction is then performed on the app bytecode to recover a super-control flow graph (sCFG) that superimposes control flow graphs (CFGs) for individual methods onto the program call graph. The analysis then recovers a directed acyclic graph (DAG) $G = (\mathcal{S}, \mathcal{F})$, where $\mathcal{S}$ is the set of program variables and $\mathcal{F}$ are edges that represent transfers of data between variables. Using the provided database of network APIs (i.e., sources), program variables are labeled as network sources $\mathcal{S}_{\text{source}} \subseteq \mathcal{S}$. Similarly, program variables that flow to input arguments of deserialization API methods are labeled as deserialization sinks $\mathcal{S}_{\text{sink}} \subseteq \mathcal{S}$ using input from the preprocessor.

A forward data flow analysis is then performed, beginning from each labeled network source in the app. This analysis iterates using a regular worklist algorithm until a fixpoint is reached. During the analysis, a standard operational semantics is used to model the propagation of data between variables, updating the DAG (G) in an incremental fashion.

Once a fixpoint has been found, a reachability analysis is performed from all network sources to deserialization sinks to obtain a relation $(\rightsquigarrow) \mapsto \mathcal{S} \times \mathcal{S}$ that indicates whether, given a pair of vertices, data flows from one to another. If a flow is ever found during this reachability analysis where

$$\exists f \text{ s.t. } s \overset{f}{\rightsquigarrow} t, s \in \mathcal{S}_{\text{source}}, t \in \mathcal{S}_{\text{sink}}, f \in \mathcal{F}^*$$

then the analysis records the sink as a potential source of a deserialized model $d \in \mathcal{D}$. This set ($\mathcal{D}$) of deserialization points serves as input to the next step of the analysis.

6.2 Hidden Model Field Identification

The second step of the static analysis takes as input the app to analyze, the set of deserialization points $\mathcal{D}$ (*sources*), and a precomputed database of APIs that render text in UI elements (*sinks*). The goal of this step is to enumerate flows from individual fields of a deserialized data model to user interface elements and, importantly, to identify model fields for which no such flow exists.

The first task of this step is to identify, for each deserialization point $d \in \mathcal{D}$, the type of the model that could be deserialized. The model type is required since the analysis needs to know the total set of fields comprising the object (as well as any nested member objects). This information can be recovered as the deserialization library also needs to know the model type to instantiate at runtime. In practice, the target model type typically appears as a `java.lang.Class` parameter to the deserialization method. The analysis uses this information to recover the deserialized model type.

Models often nest other models which the analysis handles by recursively identifying their types in order to enumerate nested fields. One challenge that arises in this context are sub-models that are contained in collection classes which use Java generics. Hush operates directly on Java bytecode and generics are subject to type erasure in accordance with the Java language specification. Thus, it is possible that type information is lost when generic containers are used. Fortunately, we found that for the Android platform the original type is helpfully preserved in the form of a Dalvik annotation (`dalvik.annotation.Signature`) that allows the analysis to recover this information from the bytecode. Although annotations are not mandatory in Dalvik code, the Signature annotation is required by most deserialization libraries, therefore Proguard and other obfuscators must be configured to keep these accordingly [19].

With the full tree of deserialized model classes in hand, the analysis then proceeds to perform a second round of forward data flow analysis. Here, individual data object fields are treated as sources $\mathcal{S}_{\text{source}}$, and API methods that set data to be displayed in user interface elements are considered sinks $\mathcal{S}_{\text{sink}}$. Similarly to the previous step, an iterative fixpoint computation is performed to construct a DAG $G = (\mathcal{S}, \mathcal{F})$ that encodes possible flows between program variables. Once a fixpoint is reached, a second reachability analysis is performed that identifies all data object fields whose data can flow to a UI element. In contrast to the previous step, however, this step of the analysis reports those fields for which *no* flows to UI elements exist. That is,

$$\nexists f \text{ s.t. } s \overset{f}{\rightsquigarrow} t, s \in \mathcal{S}_{\text{source}}, t \in \mathcal{S}_{\text{sink}}, f \in \mathcal{F}^*.$$

These hidden fields deserialized from network input are considered potential SIFON vulnerabilities, and serve as input to the final classification step of the static analysis.

6.3 Hidden Model Field Classification

The final step of the static analysis stage is a heuristic post-filter that classifies the hidden fields identified in the previous two steps as to whether they are likely to contain sensitive information. This filter is necessary to reduce the workload on the subsequent dynamic analysis stage by focusing effort on fields that are more likely to be considered sensitive personal or corporate information.

This classification step takes as input both the set of hidden fields as well as their corresponding variable names. These field names are easily recovered from the app bytecode. A simple heuristic is then applied that checks each field name against a database of sensitive data patterns that were manually compiled from keywords extracted from breach reports in the Chronology of Data Breaches database [20]. The breach report states the type of information compromised for a variety of organizations including businesses, government, military, medical providers and educational institutions. For example, we derive patterns from the keywords location, date of birth, and gender, manually extracted from a breach report on July 24, 2013 stating Tinder's mobile app leaked this user information. Furthermore, we extracted keywords from the Android API representing PII. For example SubscriberId, returns the IMSI for a GSM phone, and DeviceId, returns the IMEI for a GSM phone and the MEID or ESN for CDMA phones [21].

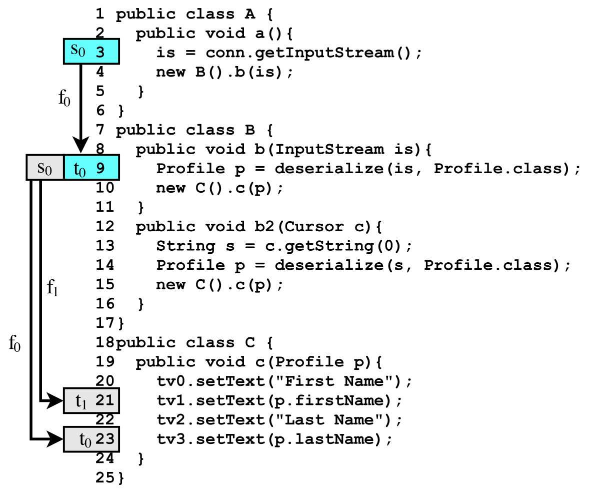
We note that while this technique is simple, it works well in practice as supported by the evaluation in Section 9. The pattern database can be adjusted using domain specific knowledge depending on the type of app being analyzed, and more advanced methods based on natural language processing or machine learning of sensitive keywords could also be leveraged to automatically infer this database if necessary.

Finally, we also note that, in principle, these data object field names could be obfuscated. However, in practice we found this not to be the case, as deserialization libraries typically match network message fields directly to data object fields based on their names. Thus, model field name obfuscation would require the developer to synchronize the obfuscated field names with the wire protocol – seemingly a fragile engineering exercise which we do not expect to take place in benign apps.

The output of this final step of the static analysis is a set of data object fields that are likely to be sensitive and hidden, i.e., not displayed, in the app's user interface. These fields are considered candidate SIFON vulnerabilities, and are forwarded to the next stage of the analysis for dynamic confirmation.

6.4 Example

To illustrate the static analysis, we apply it to the code depicted in Figure 2 which is based on the running example introduced in Section 2. The first analysis identifies a flow where a `Profile` data model is deserialized from data received over the network, $s_0 \overset{f_0}{\rightsquigarrow} t_0$. The second step of the analysis identifies two flows where the profile's fields are displayed to the user, $s_0 \overset{f_0}{\rightsquigarrow} t_0$ and $s_0 \overset{f_1}{\rightsquigarrow} t_1$. However, no flow is found from the profile's `email` field; this is flagged as potential SIFON vulnerabilities. Finally, the classification step identifies the email field as likely to be sensitive, and outputs this field as a candidate SIFON vulnerability.



```
public class A {
  public void a(){
    is = conn.getInputStream();
    new B().b(is);
  }
}
public class B {
  public void b(InputStream is){
    Profile p = deserialize(is, Profile.class);
    new C().c(p);
  }
  public void b2(Cursor c){
    String s = c.getString(0);
    Profile p = deserialize(s, Profile.class);
    new C().c(p);
  }
}
public class C {
  public void c(Profile p){
    tv0.setText("First Name");
    tv1.setText(p.firstName);
    tv2.setText("Last Name");
    tv3.setText(p.lastName);
  }
}
```

Figure 2: S-Hush applied to the running example from Section 2.

7 D-HUSH

In this section, we describe D-Hush, the confirmation stage of the Hush analysis pipeline. The goal of this stage is to confirm the set of candidate SIFON vulnerabilities reported by the prior static analysis of an app using a semi-automated dynamic analysis. Recall that this step is necessary as the static analysis can only reason about the structure of models and fields. However, it is impossible for any static analysis to attest whether sensitive fields are populated with data at runtime.

The reason that the dynamic analysis is not completely automated is due to current limitations in automatic exploration of GUI-based apps. The flows discovered with S-Hush are commonly too complicated to be realized by an automated dynamic UI exploration system (for an example refer to Section 9). However, D-Hush is flexible and allows us to immediately benefit from future improvements in automated UI exploration.

D-Hush consists of four sequential steps:*(i)* hooking setup, *(ii)* app seeding, *(iii)* user interface exploration and data collection, and *(iv)* SIFON vulnerability confirmation. In the remainder of this section, we elaborate on the details of each of the dynamic analysis steps.

7.1 Hooking Setup

The first step of this analysis stage uses the information extracted in the initial preprocessing stage to place hooks in the app under test (AUT). These hooks are used to log data at specific program points during dynamic exploration, which includes call sites for methods that receive data from the network, call sites that deserialize data model objects, and call sites that populate UI elements. For each of these categories of call sites, a specific set of information is logged. For network call sites, the data received from the remote endpoint is collected. For UI call sites, the data set for the UI element is recorded. For deserialization call sites the input data and the output object model is logged. Furthermore, we instrument the resulting deserialized object to hook all method invocations, including property accesses, on this object.

7.2 Application Seeding

The second step of the dynamic analysis stage is a manual preparatory step that overcomes a fundamental difficulty in dynamically analyzing complex AUTs. In particular, many apps require an initial configuration in order to exhibit the majority of their functionality. The canonical is the requirement to create a user account with the app's cloud back-end. Without an account, the available functionality exposed by an AUT is often limited, resulting in poor coverage of the AUT at runtime. Therefore, the dynamic analysis stage requires an analyst to manually perform any necessary and arbitrarily complicated initial configuration, including but not limited to user account creation and the configuration of the app's settings.

7.3 UI Exploration and Data Collection

The third step of the dynamic analysis explores the UI state space of the app. Hooks triggered during this step record information according to the category of the call site that has been instrumented. The main goal of this manual analysis is to maximize coverage of the AUT by exercising the UI state space as exhaustively as possible. We note that this exploration does not entail providing invalid, unexpected, or random data as inputs – i.e., the analyst does not fuzz the app, nor tamper with network traffic as a means of discovering data leakage. Rather, the AUT is explored exactly as if a regular user is using it.

7.4 SIFON Vulnerability Confirmation

The fourth step of D-Hush processes the outputs generated during the dynamic exploration step. Given the output of the static analysis stage, the goal of this step is to confirm that*(i)* statically-identified data flows exist at runtime from network sources to deserialization sinks, *(ii)* statically-labeled candidate model object fields are populated with data by the cloud service.

To that end, this step processes the logs generated by the hooks in temporal order on a per-thread basis. From this, a first bipartite graph $B = (\mathcal{N}, \mathcal{D}, \mathcal{F})$ is generated where $\mathcal{N}$ is the set of network sources, $\mathcal{D}$ is the set of deserialization sinks, and $\mathcal{F}$ is the set of edges representing flows from $\mathcal{N}$ to $\mathcal{D}$. For each flow $f \in \mathcal{F}^*$, the dynamically-generated logs are analyzed to enumerate the fields that were populated with data by the deserialization operation. These populated fields are now used as source ($\mathcal{N}'$) to create a new bipartite graph $B' = (\mathcal{N}', \mathcal{D}', \mathcal{F}')$, where $\mathcal{D}'$ is the set of methods invoked on the deserialized object during runtime. Sources that have no flows to sinks represent fields that are populated by the cloud backend but are never used by the app. These object fields are labeled as *confirmed* vulnerabilities to be reported and reviewed by the analyst.

8 HUSH IMPLEMENTATION

In this section, we discuss details of our open source prototype implementation of Hush [22] and elaborate on specific challenges that we had to overcome to realize Hush. Our Hush prototype was implemented on top of a series of open source tools In particular, the prototype focuses on the Google Gson [23] deserialization library. According to AppBrain [24], this library is the #1 serialization solution and is included by 14.66% of all installed

Android apps. Beyond Gson, Hush supports the Google Protocol Buffers [25], FasterXML Jackson [26], and FlexJson [27] data serialization libraries. We note that Hush supports any data deserialization method for which the model can be automatically extracted. During the analysis, when a deserialization point is reached, Hush will attempt to extract the model class from a method parameter of type `java.lang.Class`. If this parameter does not exist, Hush uses the method's class as the model. This approach provides the flexibility necessary to support arbitrary deserialization libraries by simply adding new deserialization methods to the configuration. No code changes are necessary to support additional libraries.

8.1 Preprocessing

Method signatures are extracted from the app byte code using the Androguard [28] reverse engineering framework. Additionally, the presence of the INTERNET permission is checked with `aapt`, a tool included in the Android SDK.

Before Hush can perform its intended analysis task, obfuscated apps must be deobfuscated. Specifically, our prototype aims to automatically identify invocations of the deserialization method `com.google.gson.Gson.fromJson` by heuristic signature matching. To this end, we perform a one-time, offline, lightweight static source code analysis over all versions of the Gson library to extract a set of strings that uniquely identify the class `com.google.gson.Gson` (e.g., error messages). Deobfuscation occurs by first decompling the AUT and identifying candidate Gson classes by matching against the previously identified strings. Next, the name and signature of each `fromJson` function in each one of these classes are extracted using signature matching. Note that the `fromJson` method in `com.google.gson.Gson` has four different signatures. As three of these signatures only take Java primitive types as parameters, deobfuscation is trivial through method signature matching since Java primitive types are never obfuscated (otherwise, method resolution would fail). In particular, it is sufficient to extract call sites to methods with an identical signature to these three `fromJson` methods, even if the name of the method has been obfuscated. The fourth signature, however, takes as one of its parameters the type `com.google.gson.stream.JsonReader`, which is subject to obfuscation. In this case, we simply omit this parameter from the signature and match against the remaining parameter of `java.lang.reflect.Type` and a return value of type `java.lang.Object`. For each identified function, we output the package name, the class name and the function signature. Despite the heuristic nature of this deobfuscation algorithm, the reverse transformation works well in practice.

8.2 S-Hush

The S-Hush static analysis stage is written in Java and uses a modified version of FlowDroid [3] to perform static data flow analysis. To obtain the required static coverage necessary to find hidden data from a network response in modern Android apps, several modifications to FlowDroid were necessary [29], including AsyncTasks and Fragment support. Fragments are used by 67% of the apps in our dataset and thus not supporting this functionality in the analysis was not an option.

We specify network input sources in a precomputed database derived from enumerating all methods in the Android framework that receive a response from an HTTP(S) connection. As examples, these include the `getInputStream` method from `HttpURLConnection` and `java.net.URLConnection` classes, `getEntity` from the org.apache.http.HttpResponse class. Fortunately, as these methods are part of the Android framework, we do not need to worry about obfuscation. User interface element sinks were manually compiled by identifying methods in the Android SDK that allow text to be displayed to the user. Examples of these sink methods include `setText` from `android.widget.TextView`, `setTitle` from `android.app.AlertDialog`, and the `loadData` method from the `android.webkit.WebView` class.

8.3 D-Hush

The prototype implementation of the D-Hush dynamic analysis stage relies on the Xposed hooking framework [30]. Xposed operates by replacing the *Zygote* process with an extended version of the `app_process` executable that launches Zygote[1]. When a new Dalvik virtual machine is created, this Xposed-modified version of `app_process` loads external packages – in this case, the hooking and logging code required to collect the information described in Section 7.

The hooking initialization procedure takes as input the set of methods to hook provided by the preprocessing step. For both UI and network methods, we used the same set as for S-Hush (see Section 8.2). For deserialization methods, however, we include logging for another JSON library, `org.json.JSONObject`, that cannot easily be analyzed statically. (For a more detailed discussion please refer to Section 10.1) The goal of this additional feature is two-fold: first, to demonstrate that SIFON vulnerabilities affects several deserialization libraries and second, to support our claim that Hush is library independent and hence highly extensible. Thus, we analyze two different categories of deserialization: JSON data deserialized to models using Gson, and JSON data mapped to dictionaries, or key-value pairs using `org.json.JSONObject`. Note that the second family is constructed dynamically, where dictionary keys are retrieved from the server at runtime and cannot be tracked by S-Hush. The dynamic analysis, however, can process these objects and extract the necessary information to detect SIFON vulnerabilities. During the hooking setup, if the data is mapped to a model, both data and model are logged. If the data is mapped to a JSON dictionary, then the dictionary's key-value mappings are logged too. Moreover, as JSON dictionaries are not mapped to models, accessing the object data is achieved through a set of accessor and mutator methods that should be hooked to track data modification.

Detecting SIFON vulnerabilities in the JSONObject scenario is slightly different from Gson. The main idea remains the same: data that is sent by the server and never used is considered *oversharing*. As the constructor of the JSONObject class is hooked, the data (i.e., key-value pairs) contained in the resulting object is also known. This includes the set of keys that are read (through the hooking of accessor methods) and those that are written to (through the

[1]On Android Zygote is the parent of all apps.

hooking of mutator methods). Keys that are created but never used (either read or write) are considered leaked information.

We note that both algorithms rely on data accesses (read or write) to assess whether a field is used or not. This process might generate false negatives. For Gson hooking, the static analysis can flag a method as accessing a field; however, this access is in a branch that is never taken. The JSON algorithm is more subtle as SIFON vulnerabilities detection is fully based on dynamic analysis. The idea is that a key might be accessed for an internal computation but never shown to the user. For instance, we noticed that in social and dating apps, user email addresses are usually sent by the server but hidden from the UI.

We therefore provide an additional operating mode (called Fuzzy mode) for users aiming to detect all SIFON vulnerabilities even if the data is used internally by the app. The downside of this technique, however, is a high number of false positives. More precisely, the algorithm performs a fuzzy string matching between the data received from the server and information displayed to the user. If a value is sent but not displayed, the algorithm flags it as suspicious. To do so, we proceed as follows: for all JSON objects received from the server, key-value pairs are extracted. Then, all strings written to the UI are collected via the hooking of UI methods set during the first phase. Finally, the algorithm compares the values received from the server and the strings set in the UI with a fuzzy string comparison algorithm.

9 EVALUATION

We evaluate the effectiveness of Hush over a large corpus (i.e., 31,559) of Android apps.To this end, we analyzed all apps in the social category of Google Play as archived by the Playdrone [16] project. We chose this category, as apps therein are likely to handle sensitive information that should not be shared beyond the owner of the data and the app provider indiscriminately.

Experimental Setup To identify likely SIFON vulnerabilities, S-Hush was run in parallel using 8 workers each allocated with 20 GB of RAM and sharing 16 2.27 GHz CPUs. To ensure forward progress of our analysis through the dataset, we set a timeout of 2 hours for the analysis of each app. D-Hush was executed on a MacBook Pro with 16 GB of RAM and two 2.6 GHz CPUs. The dynamic analysis used the Genymotion Android emulator running Android 4.1.1 with Xposed framework version 2.6.1.

S-Hush Results We started with 31,559 apps contained in the Playdrone data set under the social category. 5,481 of these apps passed the preprocessing filter (see Section 5). That is, all apps that request the `INTERNET` permission, communicate with the network, and are found to invoke a (potentially obfuscated) deserialization method. Further, 10.46% of these apps contained obfuscated deserialization methods. During the analysis, 315 apps resulted in FlowDroid requesting more than the allocated memory (i.e., run out of memory), 177 did not finish within the 2 hour time limit (i.e., run out of time) and 553 terminated due to fatal errors (e.g., FlowDroid runtime errors).

The analysis found that 998 apps have static data flows from network sources to deserialization points among which 951 have also flows between deserialized object fields and UI elements. Finally, 126 apps contained models with sensitive information that is not displayed to the user. These apps are then forwarded to D-Hush for confirmation.

The average analysis time for an app was 5 minutes 33 seconds, which supports our decision to set the timeout to 2 hours. Furthermore, the average memory usage of the static analysis tasks was 2.39 GB. In total, S-Hush processed 5,481 apps and we manually investigated the 177 apps whose analysis was terminated due to the timeout. This investigation revealed that the single most prevalent reason (i.e., 161/177 apps) for the timeout was that the interprocedural, finite, distributive subset (IFDS) solver used by FlowDroid got stuck and did not make further progress in the analysis.

D-Hush Results S-Hush reported 126 apps with potential SIFON vulnerabilities. These apps were first examined to determine compatibility with D-Hush. We found 38 were not runnable due to either a programming bug (i.e., app crashes at starting for 11 apps) or to network endpoints that were not reachable at the time of our analysis (27 apps). Additionally, we were unable to analyze 12 apps as 6 apps had user interfaces in non-latin languages, which prevented the authors from meaningfully engaging with the UI, and 6 apps required special credentials (e.g., special pin code).

Thus, we were left with 76 apps that potentially contain SIFON vulnerabilities. Our evaluation confirmed SIFON vulnerabilities in a total of eight apps. Table 1 presents the sensitive data that was found to be leaked in each app accompanied by the number of installs in the Google Play Store. In the following sections, we select App A and App B as case studies to explore the SIFON vulnerabilities in more detail.

9.1 Case Study: App A

App A is an online dating app. S-Hush identified this app as having a potential SIFON vulnerability due to the server responding with the first and last name, date of birth, ZIP code, and email address during a profile request indiscriminately of the user who is sending the request. The detailed results of the static analysis are displayed in Table 2.

The data is deserialized to the object specified by "Model." "Shown" represents the number of fields presented to the user while "Hidden" are those not displayed by a UI element. "Sensitive" represents the number of fields that are hidden and also considered sensitive. This is the most important metric when determining if the app may have a SIFON vulnerability. Additionally, we report the number of fields in the model that are never used anywhere in the app as "Unused." The same metrics are reported for each case study. App A is a textbook example of a SIFON vulnerability. S-Hush identified that the `Contact` model is deserialized from three network responses, and the hidden model fields which are considered sensitive are different across all three cases. This indicates that the developers implemented a local access control policy in an attempt to secure the data. App A was then forwarded to D-Hush for validation. Upon installation of the app, manual interaction was required to create an account. Dynamic analysis confirmed that when a search is performed, the server responds with a list of matches. When one of these matches is clicked, for whom we are not connected to, the server returns the information as identified by the static analysis. These results are then deserialized by the Gson library. Overall, the dynamic analyses identified 67 fields,

Apps	Leaked Data	Number Installs
App A	first and last name, DOB, last action*, ZIP, gender*, user ID*, email, profile status	10,000 - 50,000
App B	email, home page, street, ZIP code, phone number	10,000 - 50,000
App C	Client OS*, email*, friend list*, user ID*, latitude, longitude	10,000 - 50,000
App D	latitude, longitude, last action	5,000 - 10,000
App E	Phone number	1000 - 5000
App F	userRelationID, latitude*, longitude*	500 - 1,000
App G	address, DOB, phone number, email*, deviceOS, Facebook ID, encrypted latitude, longitude, and password	100-500
App H	DOB, hashed password, last login, user type	100 - 500

Table 1: Overshared information per app. (*) indicates fields discovered with the fuzzy mode.

out of which 52 were shown to the user. From the remaining 15, one field was not sent by the server, seven were sent with "---" (i.e., three dashes) as content, and seven contained sensitive information. In addition, this particular app appears to use integers for the member IDs, potentially allowing an adversary to enumerate member records to perform unauthorized bulk collection of sensitive user data.

9.2 Case Study: App B

App B is a social network connecting dog owners, breeders, and dogs. S-Hush identified this app as having a potential SIFON vulnerability due to the server responding with the first and last name, street address, ZIP code, country, phone, and email address during a profile request, while keeping this data hidden from the user. The detailed results of S-Hush are displayed in Table 2. When the server responds to a request to obtain the profile, the data returned is deserialized in the `UserEntity` model. This app only displays data from four of the 28 model fields, eight of which are considered sensitive. App B was then forwarded to D-Hush for validation. This app also required the creation of a user account. However, account creation is not supported from the mobile client. Thus, we had to register the account through the app's accompanying website. Overall, 146 fields were analyzed, 60 were shown to the user, and five were identified as sensitive. Among the 86 unused fields 46 are never sent by the server, 3 are empty, and 37 have some value. A quick analysis of these values showed that the `AnimalOwnerEntity` is the only model leaking sensitive information. Among its seven unused fields, the first and last name are not considered sensitive, and thus there are five detected leaks. The SIFON vulnerability is triggered as follows. When browsing a user profile, a list a of dogs owned by the user is presented. By clicking on a dog, a request for the dog's details is made to the server. The response includes the owners information (i.e., the `AnimalOwnerEntity` model). This information is sent but never shown to the user. This result shows how difficult it can be to prevent SIFON vulnerabilities: while explicit user data (i.e., `UserEntity`) was never sent by the server, that same data was nevertheless leaked through a different model (`AnimalOwnerEntity`).

10 CHALLENGES AND LIMITATIONS

As Hush relies on static and dynamic analysis techniques, Hush is subject to their fundamental limitations, such as limited path coverage for dynamic analysis and potentially high false positives due to over-approximation in the static analysis phase. However, beyond these globally applicable limitations, our prototype also suffers from challenges and limitations that result from our implementation. These limitations mainly arise from shortcomings of our static and dynamic analyses.

Model	Shown	Hidden	Sensitive	Unused	Total
LoginObj	7	1	1	0	8
BlogArticle	4	4	0	0	8
InboxMsg	7	5	0	0	12
Contact	50	7	1	1	57
Contact	41	16	1	10	57
Contact	43	14	1	8	57
UserEntity	4	24	8	2	28
VerisionEntity	0	1	0	0	1
RegisterEntity	3	15	0	14	18

Table 2: Static analysis results for App A (top) and App B (middle)

10.1 Static analysis limitations

The preprocessing step in Hush uses heuristics to deobfuscate apps before submitting them for static analysis. Thus, obfuscation techniques that go beyond merely renaming method names can potentially thwart this step. However, recall that Hush aims at analyzing benign apps and thus we would not anticipate advanced obfuscation techniques to be beneficial to regular app developers.

As demonstrated in Section 8, S-Hush is flexible enough to handle a variety of serialization libraries. However, one method of deserializing network data into Java data-structures is based on Android's `JSONObject` class. Instead of deserializing network data into a model of a specific type, `JSONObject` will simply deserialize a JSON string into a nested `java.util.Map`. The app can then access individual values in this structure by indexing the map with a *key* value. The Map instances returned by the `JSONObject` class uses regular strings as keys. Thus, to accurately reason about these nested structures would require a precise handling of string values and operations – a capability currently not supported by S-Hush.

Finally, as S-Hush is implemented on top of FlowDroid, it inherits FlowDroid's soundness characteristics. As most static analysis systems that target complex apps, FlowDroid and by extension S-Hush cannot be *sound* "as this would make the analysis unscalable or imprecise to the point of being useless" [31].

10.2 Dynamic analysis limitations

Android apps are interactive and event-driven. Thus, inputs are normally in the form of events that correspond to user interactions (UI events), or system events, such as an incoming phone call or text message. While testing tools can generate such events automatically, previous research (e.g., [14]) concluded that simple approaches outperform advanced user interface exploration techniques along many dimensions. For the use case considered in Hush, the most important dimension is code coverage. Unfortunately, Choudhary et al. [14] found that code coverage of any existing automated GUI exploration tool is sobering (∼ 48%) with Monkey scoring the highest. Based on these results, we experimented with the Monkey [32] to see whether we would achieve sufficient coverage to confirm the suspected SIFON vulnerabilities automatically. While this approach worked for App H, none of the other vulnerabilities could be confirmed by this fully automated technique. The reason is that the user interactions to manifest the vulnerabilities were too specific for Monkey to trigger.

However, in a scenario where a test engineer applies Hush to detect SIFON vulnerabilities for software testing purposes, the engineer could simply trigger (or create if necessary) test cases that mimic the corresponding user interactions. Testing frameworks such as Expresso [33] already support such user interface testing.

Similar to improved static analysis tools, Hush can immediately benefit from improved UI exploration techniques once these techniques manage to generate the complex user interactions required to trigger SIFON vulnerabilities.

11 RELATED WORK

There have been many research efforts to detect, measure, and protect sensitive information leakage in mobile platforms. These efforts fall into three major categories.

Static Analysis For static analysis, a natural starting point is the Android permission system. Hence, it has been widely scrutinized. For example, at a system level, PScout [34] extracts and analyzes the permission specification from Android OS source code and shows that at least 22% of the permissions are not documented. Stowaway [35] considers the same problem from the app's point of view, and shows that many apps are over-privileged. Note that as SIFON vulnerabilities occur at the cloud service, enforcing permissions at the client side cannot be an effective countermeasure.

A popular use of static analysis is for vulnerability discovery. For example, CryptoLint [36] uses program slicing to find cryptographic misuse in Android apps. CHEX [6], in comparison, scans Android apps for component hijacking vulnerabilities. Compared to related work, Hush is a novel detection system specifically for SIFON vulnerabilities, which have not been investigated in prior work to the best of our knowledge.

Outgoing data leakages from mobile apps to external, third-party services has been widely studied through static analysis [2–5, 37, 38]. For example, LeakMiner [38], FlowDroid [3], DroidSafe [4], and DidFail [5] perform static data flow analysis to identify data leakages in Android apps. Note that these tools consider only information leakage originating from the client to a third-party service. S-Hush, in contrast aims to identify sensitive, hidden data received by the app *from* a network endpoint.

Dynamic Analysis TaintDroid [39] was the first taint tracking framework to uncover information leakage at runtime for the Android platform. Dynamic instrumentation is now widely used for mobile malware analysis. Sandbox systems such Andrubis [40] and DroidBox [41] use custom instrumentation of the Android system coupled with taint tracking. VMI-based dynamic analysis systems such as DroidScope [42] and CopperDroid [43] are proposed as a general-purpose VM-based out-of-the-box framework to reconstruct Android malware behaviors. SmartDroid [44] and AppsPlayground [45] improve code coverage by intelligently stimulating the app during dynamic analysis to reveal malicious behavior.

Similarly to existing static analysis systems, current dynamic analysis systems for mobile apps are tailored to detect either information leakage from client to an external server, or to detect specific malicious activities on the device. Hush, in contrast, specifically targets SIFON vulnerabilities that, to our knowledge, have not been studied before.

Access Control Enforcement Several approaches have been proposed to enhance the security of the Android OS, and prevent data leakage. For example, Kirin [46] enforces the security of apps by limiting permissions for the app being installed. Frameworks such as XManDroid [47], Saint [48], TrustDroid [49], and many others [50] focus on controlling the communication between components in different apps. In comparison, TISSA [9], MockDroid [7], and AppFence [8] allow the specification of fine-grained policies, and the substitution of fake information returned from the Android API. Note that all these protection mechanisms are deployed on the client, and, hence, cannot prevent SIFON vulnerabilities. Finally, access control can be guaranteed by the programming language. Frameworks such as swift [51] and SIF [52] aim at building web applications that are secure by construction. While such frameworks can be used to prevent SIFON vulnerabilities, they are not widely adopted as they require a specific programming language (i.e., JIF) and the data to be annotated.

12 CONCLUSION

In this work, we presented SIFON, or server-based information oversharing, a new class of security vulnerabilities in multi-tier applications. SIFON vulnerabilities arise due to oversharing of information from server-side APIs that is not displayed by the application's user interface. We described Hush, a semi-automated approach to discover and confirm the presence of SIFON vulnerabilities. Hush first performs a scalable multi-stage static data flow analysis to screen applications for potential vulnerabilities, and then confirms the presence of candidate vulnerabilities with a human-assisted dynamic analysis. We implemented a prototype of Hush for the Android platform and demonstrates that it possible to quickly scan thousands of Android applications for SIFON vulnerabilities with minimal effort. Our work is a first step towards a systematic, fully automated framework for server side information leakage discovery and mitigation.

ACKNOWLEDGMENTS

This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless communications and mobile computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[2] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 2014.

[4] M. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-Flow Analysis of Android Applications in DroidSafe," in *Proceedings of the ISOC Network and Distributed Security Symposium (NDSS).* Internet Society, 2015.

[5] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014.

[6] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[7] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

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

[9] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, 2011.

[10] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Security and Privacy (SP), 2010 IEEE Symposium on.* IEEE, 2010, pp. 191–206.

[11] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, "Statistical identification of encrypted web browsing traffic," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on.* IEEE, 2002, pp. 19–30.

[12] K. Schwaber, *Agile project management with Scrum.* Microsoft Press, 2004.

[13] N. Huq, "Follow the data: Dissecting data breaches and debunking myths."

[14] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" *CoRR*, 2015.

[15] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "AdRob: Examining the Landscape and Impact of Android Application Plagiarism," in *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2013.

[16] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems.* ACM, 2014.

[17] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland).* IEEE Computer Society, 2012.

[18] Google, Inc., "ProGuard," http://developer.android.com/tools/help/proguard.html.

[19] "Proguard configuration for Gson ," https://github.com/google/gson/blob/master/examples/android-proguard-example/proguard.cfg.

[20] Privacy Rights Clearinghouse, "Chronology of Data Breaches," http://www.privacyrights.org/data-breach, 2015.

[21] Google, Inc., "TelephonyManager, Android Developers," http://developer.android.com/reference/android/telephony/TelephonyManager.html, 2015.

[22] "Hush," https://github.com/BUseclab/hush.

[23] Google, Inc., "Gson Deserialization Library," https://sites.google.com/site/gson/, 2015.

[24] AppBrain, "AppBrain Stats," http://www.appbrain.com/stats/libraries/dev, 2015.

[25] Google, Inc., "Protocol Buffers," https://developers.google.com/protocol-buffers/, 2015.

[26] FasterXML, LLC, "FasterXML, LLC," https://github.com/FasterXML, 2015.

[27] Charlie Hubbard , "FLEXJSON," http://flexjson.sourceforge.net/, 2015.

[28] Androguard Team, "Androguard," https://github.com/androguard/androguard, 2015.

[29] "FlowDroid for Hush," https://github.com/BUseclab/soot-infoflow-android.

[30] rovo89, "Xposed Module Repository," http://repo.xposed.info/, 2015.

[31] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Mller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, 2015.

[32] Google, "The Monkey UI android testing tool," http://developer.android.com/tools/help/monkey.html.

[33] Facebook, "Espresso: Functional UI Testing Framework," http://developer.android.com/tools/testing-support-library/index.html#Espresso, 2015.

[34] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* ACM, 2012.

[35] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security.* ACM, 2011, pp. 627–638.

[36] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* ACM, 2013.

[37] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, in *18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, UNITED STATES, 2011.

[38] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Proceedings of the 2012 Third World Congress on Software Engineering (WCSE)*, 2012.

[39] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, 2014.

[40] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis – 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *Proceedings of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, 2014.

[41] P. Lantz, "Android Application Sandbox," http://code.google.com/p/droidbox/, February 2011.

[42] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[43] K. Tam, S. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," in *Proceedings of the ISOC Network and Distributed Security Symposium (NDSS)*, 2015.

[44] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[45] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

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

[47] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," Tech. Rep., 2011.

[48] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," *Security and Communication Networks*, 2012.

[49] Z. Zhao and F. C. C. Osono, "Trustdroid: Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking," *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, 2012.

[50] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies," in *Presented as part of the 22nd USENIX Security Symposium*, 2013.

[51] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007.

[52] S. Chong, K. Vikram, and A. C. Myers, "Sif: Enforcing confidentiality and integrity in web applications," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 1:1–1:16.