Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis

Andrea Continella*†, Yanick Fratantonio†, Martina Lindorfer†, Alessandro Puccetti†, Ali Zand†,
Christopher Kruegel†, and Giovanni Vigna†
*Politecnico di Milano †UC Santa Barbara
andrea.continella@polimi.it, {yanick,martina,chris,vigna}@cs.ucsb.edu, ale.puccetti@gmail.com, ali.zand@gmail.com

***Abstract*—Mobile apps are notorious for collecting a wealth of private information from users. Despite significant effort from the research community in developing privacy leak detection tools based on data flow tracking inside the app or through network traffic analysis, it is still unclear whether apps and ad libraries can hide the fact that they are leaking private information. In fact, all existing analysis tools have limitations: data flow tracking suffers from imprecisions that cause false positives, as well as false negatives when the data flow from a source of private information to a network sink is interrupted; on the other hand, network traffic analysis cannot handle encryption or custom encoding.**

We propose a new approach to privacy leak detection that is not affected by such limitations, and it is also resilient to obfuscation techniques, such as encoding, formatting, encryption, or any other kind of transformation performed on private information before it is leaked. Our work is based on black-box differential analysis, and it works in two steps: first, it establishes a baseline of the network behavior of an app; then, it modifies sources of private information, such as the device ID and location, and detects leaks by observing deviations in the resulting network traffic. The basic concept of black-box differential analysis is not novel, but, unfortunately, it is not practical enough to precisely analyze modern mobile apps. In fact, their network traffic contains many sources of non-determinism, such as random identifiers, timestamps, and server-assigned session identifiers, which, when not handled properly, cause too much noise to correlate output changes with input changes.

The main contribution of this work is to make black-box differential analysis practical when applied to modern Android apps. In particular, we show that the network-based non-determinism can often be explained and eliminated, and it is thus possible to reliably use variations in the network traffic as a strong signal to detect privacy leaks. We implemented this approach in a tool, called AGRIGENTO, and we evaluated it on more than one thousand Android apps. Our evaluation shows that our approach works well in practice and outperforms current state-of-the-art techniques. We conclude our study by discussing several case studies that show how popular apps and ad libraries currently exfiltrate data by using complex combinations of encoding and encryption mechanisms that other approaches fail to detect. Our results show that these apps and libraries seem to deliberately hide their data leaks from current approaches and clearly demonstrate the need for an obfuscation-resilient approach such as ours.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23465

I. INTRODUCTION

One main concern of mobile app users is the leakage of private information: Mobile apps, and third-party advertisement (ad) libraries in particular, extensively collect private information and track users in order to monetize apps and provide targeted advertisements. In response, the security community has proposed numerous approaches that detect whether a given app leaks private information to the network or not. The majority of approaches utilize data flow analysis of the app's code, both through static and/or dynamic taint analysis. Tools based on static taint analysis, such as FlowDroid [6], identify possible sources of private information and determine how their values flow throughout the app and, eventually, to sinks, such as the network. Dynamic taint analysis tools, such as TaintDroid [13], execute apps in an instrumented environment and track how private information is propagated while the app is running. Finally, AppAudit combines both approaches, determining critical flows that leak data through static analysis and verifying them through an approximated dynamic analysis [46].

While these tools provide useful insights, they suffer from several limitations that affect their adoption, especially when the threat model considers apps that try to hide the fact that they are leaking information. Adversaries can deliberately add code constructs that break the flow of information throughout an app and make data flow analysis approaches "lose track" of tainted values. Related works [9], [36] have already demonstrated how an app can, for example, use indirections through implicit control flows or through the file system to efficiently bypass static and dynamic data flow analysis. Furthermore, static and dynamic analysis approaches for mobile apps usually only inspect data flow in Dalvik bytecode (i.e., the Java component of the app) and miss data leaks in native code components, which are becoming more and more prevalent [5], [26]. Both static and dynamic analysis can also have false positives, mainly due to a phenomenon called overtainting: imprecisions in modeling how information flows through specific instructions, or imprecisions introduced to make the analysis scalable might establish that a given value is "tainted" with private information even when, in fact, it is not.

Since static analysis does not perform real-time detection of privacy leaks, and dynamic analysis requires heavy instrumentation, and is thus more likely to be used by app stores than by end users, researchers have recently proposed a more light-weight alternative: identifying privacy leaks on the network layer through traffic interception [24], [27], [34], [35], [39]. However, obfuscation is out-of-scope for the majority of approaches as they perform simple string matching and essentially "grep" for hardcoded values of private information and some well-

known encodings such as Base64 or standard hashing algorithms. ReCon [35] is the most resilient to obfuscation as it identifies leaks based on the structure of the HTTP requests, for example by learning that the value following a "deviceid" key in a HTTP GET request is probably a device ID. Still, the underlying machine learning classifier is limited by the data it is trained on, which is collected through TaintDroid and manual analysis—if the labelling process misses any leak and its corresponding key, e.g., due to obfuscation or custom encoding, ReCon will not be able to detect it.

In general, the transformation of privacy leaks, from simple formatting and encoding to more complex obfuscations, has gotten little attention so far. Only BayesDroid [42] and MorphDroid [17] have observed that the leaked information does not always exactly match the original private information, but focused on leaks consisting of subsets or substrings of information instead of obfuscation. It is unclear to what extent apps can hide their information leaks from state-of-the-art tools. For this purpose, we developed a novel automatic analysis approach for privacy leak detection in HTTP(S) traffic that is agnostic to how private information is formatted or encoded. Our work builds on the idea of observing network traffic and attempts to identify leaks through a technique similar to the differential analysis approach used in cryptography: first, we collect an app's network traffic associated with multiple executions; then, we modify the input, i.e., the private information, and look for changes in the output, i.e., the network traffic. This allows us to detect leaks of private information even if it has been heavily obfuscated.

The idea to perform differential black-box analysis is intuitive, and in fact, has already been explored by Privacy Oracle [23] for the detection of information leaks in Windows applications. One of the main challenges of performing differential analysis is the elimination of all sources of non-determinism between different executions of an app. Only by doing this one can reliably attribute changes in the output to changes in the input, and confirm the presence of information leaks. While Privacy Oracle was mainly concerned with *deterministic executions* to eliminate OS artifacts that vary between executions and could interfere with the analysis, we observed that *non-deterministic network traffic* poses a far greater challenge when analyzing modern apps. Due to the frequent use of random identifiers, timestamps, server-assigned session identifiers, or encryption, the network output inherently differs in every execution. These spurious differences make it impractical to detect any significant differences caused by actual privacy leaks by simply observing variations in the raw network output.

One key contribution of this work is to show that, in fact, it is possible to explain the non-determinism of the network behavior in most cases. To this end, we conducted a small-scale empirical study to determine the common causes of non-determinism in apps' network behavior. Then, we leveraged this knowledge in the development of a new analysis system, called AGRIGENTO, which eliminates the root causes of non-determinism and makes differential analysis of Android apps practical and accurate.

Our approach has the key advantage that it is "fail-safe": we adopt a conservative approach and flag any non-determinism that AGRIGENTO cannot eliminate as a "potential leak." For each identified leak, AGRIGENTO performs a risk analysis to quantify the amount of information it contains, i.e., its risk, effectively limiting the channel capacity of what an attacker can leak without raising an alarm. We performed a series of experiments on 1,004 Android apps, including the most popular ones from the Google Play Store. Our results show that our approach works well in practice with most popular benign apps and outperforms existing state-of-the-art tools. As a result, AGRIGENTO sheds light on how current Android apps obfuscate private information before it is leaked, with transformations going far beyond simple formatting and encoding. In our evaluation, we identified several apps that use custom obfuscation and encryption that state-of-the-art tools cannot detect. For instance, we found that the popular InMobi ad library leaks the Android ID using several layers of encoding and encryption, including XORing it with a randomly generated key.

It is not surprising that developers are adopting such stealth techniques to hide their privacy leaks, given the fact that regulators such as the Federal Trade Commission (FTC) have recently started to issue sizable fines to developers for the invasion of privacy of their users [14], [15]: aforementioned InMobi for example is subject to a penalty of $4 million and has to undergo bi-yearly privacy audits for the next 20 years for tracking users' location without their knowledge and consent [16]. Also, counterintuitively to the fact that they are collecting private information, app developers are also seemingly becoming more privacy-aware and encode data before leaking it. Unfortunately, it has been shown that the structured nature of some device identifiers makes simple techniques (e.g., hashing) not enough to protect users' privacy [12], [18]. Consequently, on one hand there is a clear motivation for developers to perform obfuscation—either to maliciously hide data leaks, or to secure user data by not transmitting private information in plaintext—on the other hand privacy leak detection tools need to be agnostic to any kind of obfuscation.

In summary, we make the following contributions:

- We developed AGRIGENTO, a tool that performs root cause analysis of non-determinism in the network behavior of Android apps.
- We show that, in most cases, non-determinism in network behavior can be explained and eliminated. This key insight makes privacy leak detection through differential black-box analysis practical.
- The results of our empirical study provide new insights into how modern apps use custom encoding and obfuscation techniques to stealthily leak private information and to evade existing approaches.

In the spirit of open science, we make all the datasets and the code developed for this work publicly available.[1]

II. MOTIVATION

This section discusses a real-world example that motivates our work. Consider the snippet of code in Figure 1. The code first obtains the Android ID using the Java Reflection API, hashes the Android ID with SHA1, XORs the hash with a randomly generated key, stores the result in JSON format, and encrypts the JSON using RSA. Finally, it sends the encrypted JSON and the XOR key through an HTTP POST request.

[1] https://github.com/ucsb-seclab/agrigento

```
StringBuilder json = new StringBuilder();
// get Android ID using the Java Reflection API
Class class = Class.forName("PlatformId")
String aid = class.getDeclaredMethod("getAndroidId",
             Context.class).invoke(context);
// hash Android ID
MessageDigest sha1 = getInstance("SHA-1");
sha1.update(aid.getBytes());
byte[] digest = sha1.digest();
// generate random key
Random r = new Random();
int key = r.nextint();
// XOR Android ID with the randomly generated key
byte[] xored = customXOR(digest, key);
// encode with Base64
String encoded = Base64.encode(xored);
// append to JSON string
json.append("O1:\'");
json.append(encoded);
json.append("\'");
// encrypt JSON using RSA
Cipher rsa = getInstance("RSA/ECB/nopadding");
rsa.init(ENCRYPT_MODE, (Key) publicKey);
encr = new String(rsa.doFinal(json.getBytes()));
// send the encrypted value and key to ad server
HttpURLConnection conn = url.openConnection();
OutputStream os = conn.getOutputStream();
os.write(Base64.encode(encr).getBytes());
os.write(("key=" + key).getBytes());
```

Fig. 1. Snippet of code leaking the Android ID using obfuscation and encryption. The example is based on real code implemented in the popular InMobi ad library.

Depending on how this functionality is implemented, existing tools would miss the detection of this leak. Existing approaches based on static analysis would miss this privacy leak if the functionality is implemented in native code [5], dynamically loaded code [31], or in JavaScript in a WebView [28]. Furthermore, the use of the Java Reflection API to resolve calls at runtime can severely impede static analysis tools.

More fundamentally, the complex lifecycle and component-based nature of Android apps make tracking private information throughout an app extremely challenging, and both static and dynamic approaches are sensitive to the disruption of the data flow. For instance, many existing tools would miss this leak if this functionality is implemented in different components. Similarly, if the app first writes the private information to a file, e.g., its settings, and only later reads it from there to transmit it via a network sink, any data flow dependency would be lost. Furthermore, data flow is also lost when the implementation is incomplete and fails to propagate data flows through relevant functions: TaintDroid for example does not track data flows through hashing functions [32].

Existing black-box approaches that analyze the network traffic would miss the detection of this leak as well, as they only consider basic encodings, such as Base64 or standard hashing algorithms, and cannot handle complex obfuscations techniques that combine multiple different encodings and algorithms such as the example code in Figure 1.

Our work attempts to fill this gap: we focus on designing and developing an approach able to detect privacy leaks even when custom obfuscation mechanisms are used. Our approach is black-box based, so it is not affected by code obfuscation or complex program constructs. Furthermore, our approach can handle obfuscations of the actual data since it does not look for specific tokens that are known to be associated with leaks, but rather treats *every* inexplicable change in the network traffic as a potential leak.

We stress that the example we discussed in this section is not synthetic, but it is actually the simplified version of a snippet taken from one of the most popular apps in the Google Play Store. Specifically, this example is the simplified version of a functionality implemented in the popular InMobi ad library. We also note that this case of nested encodings and encryption is not just an isolated example: our experiments, discussed at length in §VI, show that these obfuscated leaks occur quite frequently and that existing black-box approaches are unable to detect them.

III. Sources of Non-Determinism

One of the key prerequisites for performing differential analysis is to eliminate any sources of non-determinism between different executions. Only by doing so, one can reliably attribute any changes in the network output following changes in private input values to information leakage. While previous work has focused on deterministic executions through the use of OS snapshots [23], according to our experiments the network itself is by far the largest source of non-determinism.

When executing an app multiple times on exactly the same device, with the same settings, and using the same user input, one would intuitively expect an app to produce exactly the same (i.e., deterministic) network traffic during every execution. However, our preliminary experiments showed that this is not the case: the network traffic and more specifically the transmitted and received data frequently changes on every execution, and even between the same requests and responses during a single execution.

This non-determinism is not necessarily something that is introduced by the app developer intentionally to evade analysis systems, but, instead, it is most often part of the legitimate functionality and standard network communication. We conducted a small-scale study on 15 Android apps randomly selected from the Google Play Store, and we investigated the most common sources of non-determinism in network traffic. We were able to identify the following categories:

- **Random values**. Random numbers used to generate session identifiers or, for instance, to implement game logic. Also, the Android framework provides developers with an API to generate 128-bit random universally unique identifiers (UUID). In the most common scenario, apps use this API to generate an UUID during the installation process.
- **Timing values**. Timestamps and durations, mainly used for dates, logging, signatures, and to perform measurements (e.g., loading time).
- **System values**. Information about the state and the performance of system (e.g., information about free memory and available storage space).
- **Encrypted values**. Cryptographic algorithms use randomness to generate initialization vectors (IV) and padding.

- **Network values**. Information that is assigned by a network resource (e.g., cookies, server-assigned session identifiers).
- **Non-deterministic execution**. Randomness inherent to the execution of an app, such as different loading times affecting the UI exploration.

IV. Approach

For any given app, our analysis consists of two main phases. In the first phase (see §IV-A), called *network behavior summary extraction*, we execute the app multiple times in an instrumented environment to collect raw *network traces*, and *contextual information*, which allows us to attribute the non-determinism that we see in the network behavior to the sources discussed in §III. We then combine these network traces with the contextual information to create a *contextualized trace* for each run. Finally, we merge the contextualized network traces of all runs into a *network behavior summary* of the app.

In the second phase of our approach (see §IV-B), we run the app again in exactly the same instrumented environment, with the only difference that we change one of the input sources of private information (e.g., IMEI, location). We then compare the contextualized trace collected in this final run with the network behavior summary of the previous runs to identify any discrepancy. We perform this comparison in two steps: *differential analysis*, which identifies differences, and *risk analysis*, which scores the identified differences to determine potential privacy leaks.

Figure 2 shows a high-level overview of our approach, while Figure 3 illustrates the individual steps in more detail using a simplified example.

A. Network Behavior Summary Extraction

Network Trace & Contextual Information. For each execution of the app in our instrumented environment, we collect a network trace, which contains the raw HTTP flows generated by the app, and *contextual information*, which contains the values generated by any of the sources of non-determinism we described earlier. Our approach here goes beyond simple network traffic analysis, and includes instrumenting the way the app is interacting with the Android framework. Specifically, AGRIGENTO is able to eliminate the different sources of non-determinism by intercepting calls from the app to certain Android API calls and recording their return values, and in some cases replacing them—either by replaying previously seen values or by returning constant values. First, AGRIGENTO records the timestamps generated during the first run of each app, and replays the same values in the further runs. Second, it records the random identifiers (UUID) generated by the app. Third, it records the plaintext and ciphertext values whenever the app performs encryption. Finally, the instrumented environment sets a fixed seed for all random number generation functions, and replaces the values of system-related performance measures (e.g., free memory, available storage space) with a set of constants.

Note that when an app uses its own custom encryption routines, or generates random identifiers itself without relying on Android APIs, AGRIGENTO will not be able to detect these as sources of non-determinism. However, as we explain in the next paragraph, our approach is conservative, which means this would produce a false positive, but not a false negative.

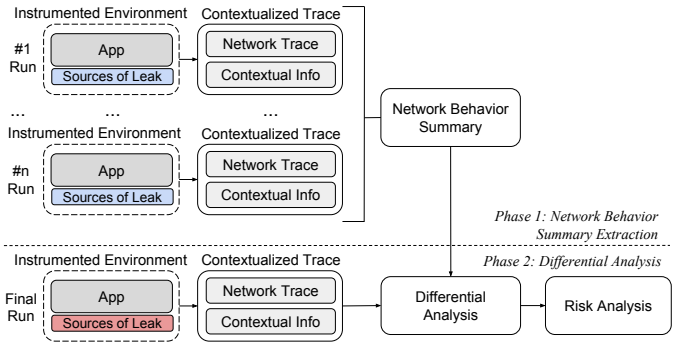

Fig. 2. High-level overview of AGRIGENTO: during the *network behavior summary extraction* it first generates a baseline of an app's network behavior during n runs, taking into account non-determinism in the contextual information; during the *differential analysis* it then modifies the sources of private information and identifies privacy leaks based on differences in the network behavior of the final run compared to the network behavior observed in the previous runs.

Contextualized Trace. We build the contextualized trace by incorporating the contextualized information into the raw network trace. To do this, we remove all sources of non-determinism (i.e., values stored in the contextual information) we encountered during the execution, by labeling all timestamps-related values, random identifiers, and values coming from the network, and decrypting encrypted content by mapping the recorded ciphertext values back to their plaintext. Essentially, we look at the raw network trace and try to determine, based on string comparison, values in the HTTP traffic that come from potential sources of non-determinism. This is similar to the techniques that previous works use to find certain values of private information in the network traffic. The key difference is that we do not perform the string matching to find leaks, but, rather, to *explain* sources of non-determinism. This is essentially the opposite goal of previous work: rather than finding leaks, we use string matching techniques to flag potential leaks as "safe." This approach has the advantage of being conservative. In fact, we flag any source of non-determinism that we cannot explain. While in previous work a failure of the string matching would lead to a missed leak (i.e., a false negative), our approach would produce, in the worst case, a false positive.

Network Behavior Summary. When AGRIGENTO builds the contextualized network traces, it essentially removes all common sources of non-determinism from the network traffic. However, it cannot fully eliminate non-determinism in the execution path of the app. Even though AGRIGENTO runs the app in an instrumented environment and replays the same sequence of events for each run, different loading times of the UI and other factors can result in different execution paths. To mitigate this issue, we run each app multiple times and merge the contextualized traces collected in the individual runs to a network behavior summary. Intuitively, the network behavior summary includes all the slightly different execution paths, generating a more complete picture of the app's network behavior. In other words, the network behavior summary represents "everything we have

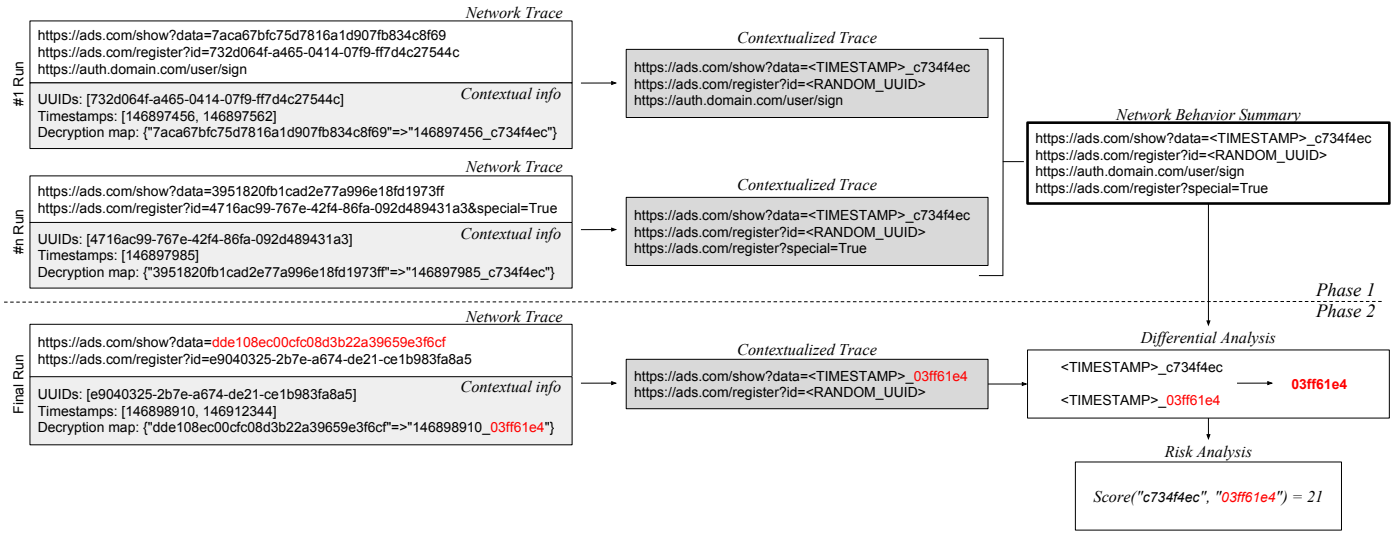

Fig. 3. Example of how AGRIGENTO performs its analysis in two phases. (1) In the first phase it builds a network behavior summary and replaces common sources of non-determinism. (2) In the second phase performs differential analysis by changing the value of an input source of private information to identify differences in the network behavior, which it then scores as potential privacy leaks.

seen" during the executions of the given app and aims at providing a trusted baseline behavior of the app.

A distinctive aspect of AGRIGENTO is how it determines the number of times each app should be executed. Intuitively, the number of runs affects the performance of our tool in terms of false positives. However, we observed that this parameter strongly depends on the complexity of the app. Therefore, our approach is iterative and decides after each run if another one is required. After each run AGRIGENTO performs the differential analysis using the collected contextualized traces. By analyzing the discrepancies in the network behavior *without* having altered any source of private information, we can understand when AGRIGENTO has sufficiently explored the app's behavior, i.e., when the network behavior summary reaches *convergence*. In practice, we say that an app reaches convergence when we do not see any discrepancies in the network behavior summary for K consecutive runs. In §VI-C, we show how this parameter sets a trade-off between the ability of explaining non-determinism and the overall time it takes AGRIGENTO to analyze an app (i.e., the average number of runs). Also, because some apps might never reach convergence, we set a maximum number of runs.

B. Differential Analysis

In a second phase, we run the app in the same environment as before, but modify the value of private information sources, such as the IMEI and location, we want to track. We can do this (a) once for all values to detect if an app is stealthily leaking information in general, or (b) multiple times—once for every unique identifier—to precisely identify the exact type of information the app is leaking. In the example in Figure 3, AGRIGENTO changed the value of a source of private information from `c734f4ec` to `03ff61e4`.

Differential Analysis. As in the previous phase, we collect a network trace and contextual information to build a contextualized trace. Then, we compare this contextualized trace against the network behavior summary, which we extracted in the previous phase. To extract the differences, we leverage the Needleman-Wunsch algorithm [29] to perform a pairwise string sequence alignment. The algorithm is based on dynamic programming and can achieve an optimal global matching. It is well-suited for our scenario: in fact, it has been successfully applied to automatic network protocol reverse engineering efforts [7], [45], [50], which conceptually have a similar goal than our network behavior summary, in that they extract a protocol from observing the network behavior during multiple executions.

At this point of our analysis, we eliminate the final source of non-determinism: values that come from the network. For each difference, AGRIGENTO checks if its value has been received in a response to a previous network request (e.g., the value is a server-assigned identifier). We assume that leaked information is not part of the payload of previous responses. This is reasonable since, in our threat model, the attacker does not know the value of the leaked source of private information in advance.

After this filtering step, AGRIGENTO raises an alert for each remaining difference between the contextualized trace in the final run and the network behavior summary. This is a conservative approach, which means that, if there is some source of non-determinism AGRIGENTO does not properly handle (e.g., apps that create UUIDs themselves or perform custom encryption without leveraging the Android framework), it will flag the app: In the worst case, this will produce a false positive.

Risk Analysis. In the last phase of our approach, AGRIGENTO quantifies the amount of information in each identified difference to evaluate the risk that an alert is caused by an actual information leak. Our key intuition is that not all identified differences bear the same risk. Thus, we assign a score to each alert based on how much the information differs from the network behavior summary. Specifically, we leverage two distance metrics, the Hamming distance and the Levenshtein distance, to compare each alert value to the corresponding value in the network behavior summary. Finally, for each app we

compute a cumulative score S as the sum of the scores of all the alerts that AGRIGENTO produced for the app. This score provides a measure of the amount of information (i.e., the number of bits) an app can potentially leak, and it can thus be used as an indirect measure of the overall risk of a privacy leak in a given app.

V. SYSTEM DETAILS

We implemented AGRIGENTO in two main components: an on-device component, which instruments the environment and collects contextual information, and the core off-device component, which intercepts the network traffic, extracts the network behavior summary, and performs the differential analysis.

A. Apps Environment Instrumentation

We implemented a module, based on Xposed [4], which hooks method calls and records and modifies their return values. As a performance optimization, AGRIGENTO applies the contextualization steps only when needed (i.e., only when it needs to address values from a non-deterministic source).

Random values. To record Android random identifiers (UUIDs) the module intercepts the return of the Android API `randomUUID()` and reads the return value. However, recording the randomly generated values is not enough: apps frequently process these numbers (e.g., multiply them with a constant), and thus they usually do not appear directly in the network traffic. To handle this scenario, we set a fixed seed for random number generation functions. By doing so, we can observe the same values in the output network traffic for each run, even without knowing how they are transformed by an app. However, always returning the same number is also not an option since this might break app functionality. Thus, we rely on a precomputed list of randomly generated numbers. For each run, the module modifies the return value of such functions using the numbers from this precomputed list. In case the invoked function imposes constraints on the generated number (e.g., integers in the interval between 2 and 10), we adapt the precomputed numbers in a deterministic way (e.g., by adding a constant), to satisfy the specific requirements of a function call.

Timing values. Also in the case of timing information, only recording the values is not enough since timestamps are often used to produce more complex values (e.g., for the generation of signatures). To deal with timestamp-related values, the module hooks all the methods providing time-related information, such as `System.currentTimeMillis()`, stores the return values in a file during the first run, and modifies the return values reading from the file in the next runs. It reads the stored timestamps in the same order as they were written and, in case one of the next runs performs more calls to a specific method than the first run (this could be due to a different execution path), it leaves the original return values unmodified for the exceeding calls.

System values. We set to constants the return values of Android APIs that apps use to perform performance measurements and fingerprint the device for example by reading information about the available storage space from `StatFs.getAvailableBlocks()`, or by querying `ActivityManager.getMemoryInfo()` for information about available memory.

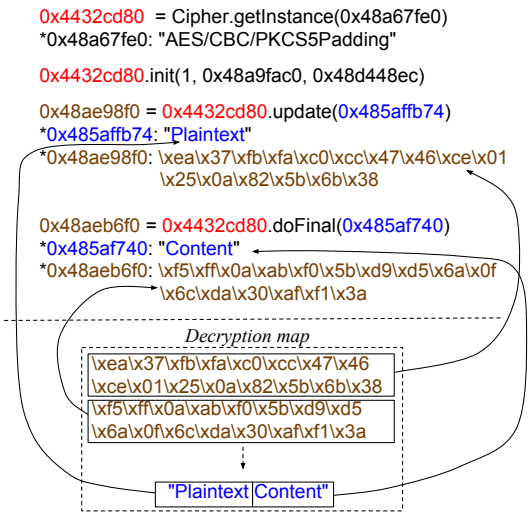

Fig. 4. Example of how AGRIGENTO leverages Crypto API traces to build an entry of the decryption map that maps ciphertext to its corresponding plaintext (**address* represents the content stored at that *address*).

Encrypted values. In order to decrypt encrypted content, we hook the Android Crypto APIs (i.e., `Cipher`, `MessageDigest`, `Mac`) and store the arguments and return value of each method. Our module parses the API traces to build a decryption map that allows it to map ciphertext to the corresponding original plaintext. Since the final ciphertext can be the result of many Crypto API calls, AGRIGENTO combines the values tracking the temporal data dependency. Figure 4 shows an example of how we use Crypto API traces to create a map between encrypted and decrypted content. Specifically, the example shows how AGRIGENTO creates an entry in the decryption map by tracing the API calls to a `Cipher` object and by concatenating the arguments of such calls (`update()`, `doFinal()`).

Patching JavaScript code. We observed many applications and ad libraries downloading and executing JavaScript (JS) code. Often, this code uses random number generation, time-related, and performance-related functions. We implemented a module in the proxy that inspects the JS code and patches it to remove non-determinism. Specifically, this module injects a custom random number generation function that uses a fixed seed, and replaces calls to `Math.random()` and `getRandomValues()` with our custom generator. Also, the JS injector replaces calls to time-related functions (e.g., `Date.now()`) with calls to a custom, injected timestamp generator, and sets constant values in global performance structures such as `timing.domLoading`.

B. Network Setup

Our implementation of AGRIGENTO captures the HTTP traffic and inspects GET and POST requests using a proxy based on the `mitmproxy` library [2]. In order to intercept HTTPS traffic, we installed a CA certificate on the instrumented device.

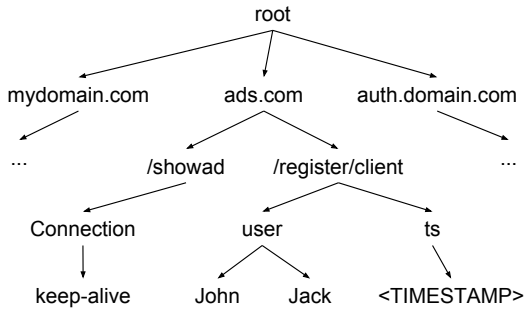

Fig. 5. Example of the tree-based data structure used to model a network behavior summary.

Furthermore, to be able to capture traffic also in the case apps use certificate pinning, we installed `JustTrustMe` [1] on the client device, which is an Xposed module that disables certificate checking by patching the Android APIs used to perform the check (e.g., `getTrustManagers()`). However, if an app performs the certificate check using custom functionality or native code, we cannot intercept the traffic.

We limit our study to HTTP(S) traffic (further referred to both as HTTP), since related work found this to be the protocol of choice for the majority (77.77%) of Android apps [11]. However, this is only a limitation of our prototype implementation of AGRIGENTO, and not a fundamental limitation of our approach.

Finally, to filter only the network traffic generated by the analyzed app, we use `iptables` to mark packets generated by the UID of the app, and route only those packets to our proxy.

C. Network Behavior Summary

We model the network behavior summary using a tree-based data structure, which contains the HTTP GET and POST flows from all the contextualized traces. The tree has four layers. The first layer contains all the domain names of the HTTP flows. The second layer contains the paths of the HTTP flows recorded for each domain. The third and fourth layers contain key-value pairs from the HTTP queries and HTTP headers. Also, we parse known data structures (e.g., JSON) according to the HTTP Content-Type (e.g., `application/json`). Figure 5 shows an example of a tree modeling a network behavior summary.

This structure is useful to group the fields of the HTTP flows that we track according to their "type" and position in the tree. In fact, when performing the differential analysis, we want to compare fields in the same position in the tree. For instance, if an HTTP request contains an HTTP value that is not part of the tree, we compare it with the other values from requests with the same domain, path, and key.

AGRIGENTO looks for privacy leaks at all levels of the tree, i.e., in all parts of the HTTP request: the domain, path, key, and values, as well as the headers and the payload. In the current implementation AGRIGENTO includes parsers for `application/x-www-form-urlencoded`, `application/json`, and any content that matches a HTTP query format (i.e., variable=value). However, AGRIGENTO can be easily extended with parsers for further content types.

Algorithm 1 Differential Analysis.

```
1:  procedure DIFFERENTIALANALYSIS(context_trace, summary)
2:      diffs ← ∅
3:      for http_flow ∈ context_trace do
4:          if http_flow ∉ summary then
5:              field ← getMissingField(http_flow, summary)
6:              fields ← getSamePositionField(field, summary)
7:              diffs.add(COMPARE(field, fields))
8:      return diffs
9:
10: procedure COMPARE(field, fields)
11:     diffs ← ∅
12:     most_similar ← mostSimilar(field, fields)
13:     if isKnownDataStructure(field, most_similar) then
14:         subfields ← parseDataStructure(field)
15:         similar_subfields ← parseDataStructure(most_similar)
16:         for i ∈ subfields do
17:             diffs.add(COMPARE(subfields_i, similar_subfields_i))
18:         return diffs
19:     if isKnownEncoding(field, most_similar) then
20:         field ← decode(field)
21:         most_similar ← decode(most_similar)
22:     alignment ← align(field, most_similar)
23:     regex ← getRegex(alignment)
24:     diffs ← getRegexMatches(field)
25:     diffs ← removeNetworkValues(diffs)
26:     diffs ← whitelistBenignLibaries(diffs)
27:     return diffs
```

D. Modifying Sources of Private Information

In our implementation we track the following sources of private information: Android ID, contacts, ICCID, IMEI, IMSI, location, MAC address, and phone number. For ICCID, IMEI, IMSI, MAC address and phone number we leverage the Xposed module to alter the return values of the Android APIs that allow to retrieve such data (e.g., `TelephonyManager.getDeviceId()`). For the Android ID we directly modify the value in the database in which it is stored, while to alter the contact list we generate intents through `adb`. We also use mock locations, which allow to set a fake position of the device for debug purposes.

E. Differential Analysis

In the second phase of our approach, AGRIGENTO modifies the input sources of private information as described in the previous section, reruns the app in the instrumented environment, and compares the new contextualized trace with the network behavior summary to identify changes in the network traffic caused by the input manipulation.

We implemented the differential analysis following the steps defined in Algorithm 1. For each HTTP flow in the contextualized trace collected from the final run, AGRIGENTO navigates the tree and checks if each field of the given flow is part of the tree. If it does not find an exact match, AGRIGENTO compares the new field with the fields in the same position in the tree (e.g., requests to the same domain, path, and key). Specifically, AGRIGENTO performs the comparison between the new field and the most similar field among those in the same position in the tree. During the comparison phase, AGRIGENTO recognizes patterns of known data structures such as JSON. If any are found, AGRIGENTO parses them and performs the comparison on each subfield. This step is useful to improve the alignment quality and it also improves the performance since aligning

shorter subfields is faster than aligning long values. Furthermore, before the comparison, AGRIGENTO decodes known encodings (i.e., Base64, URLencode). Then, AGRIGENTO leverages the Needleman-Wunsch algorithm to obtain an alignment of the fields under comparison. The alignment identifies regions of similarity between the two fields and inserts gaps so that identical characters are placed in the same positions. From the alignment, AGRIGENTO generates a regular expression. Essentially, it merges consecutive gaps, and replaces them with a wildcard (i.e., $\star$). Finally, it obtains a set of differences by extracting the substrings that match the wildcards of the regular expression from a field. AGRIGENTO then discards any differences caused by values that have been received by previous network requests (e.g., server-assigned identifier). Finally, AGRIGENTO also whitelists benign differences caused by known Google libraries (e.g., `googleads`), which can be particularly complex to analyze and that contain non-determinism AGRIGENTO cannot efficiently eliminate.

Example. For instance, in this simplified case, the network behavior summary tree contains the following HTTP flows:

```
domain.com/path?key=111111111_4716ac99767e
domain.com/path?key=111111111_6fa092d4891a
other.com/new?id=28361816686630788
```

The HTTP flow in the contextualized trace collected from the final run is:

```
domain.com/path?key=999999999_4716ac99767e
```

AGRIGENTO navigates the tree from `domain.com` to `key`, and then determines that `999999999_4716ac99767e` is not part of the tree. Hence, it selects the most similar field in the tree, and performs the comparison with its value. In this case, it aligns `999999999_4716ac99767e` with `111111111_4716ac99767e`. Starting from the alignments it produces the regular expression `*_4716ac99767e` and determines `999999999` as the difference in the network behavior of the final run compared to the network behavior summary of previous runs.

F. Risk Analysis

As mentioned in §IV-B we combine the Hamming and the Levenshtein distance to compute a score for each of the differences AGRIGENTO identifies during differential analysis. In particular, we are interested in quantifying the number of bits that differ in the network traffic of the final run from what we have observed in the network behavior summary.

For each field that the differential analysis flagged as being different from the previously observed network traffic, we compute a score based on the distance of its value to the most similar value in the same position of the network behavior summary. This is equivalent to selecting the minimum distance between the value and all other previously observed values for a specific field.

Given an app A, D (= the differences detected by analyzing A), and F (= all the fields in the tree of A's network behavior), we then compute an overall score S_A that quantifies how many bits the app is leaking:

$$distance(x,y) = \begin{cases} Hamming(x,y) & \text{if } len(x) = len(y) \\ Levenshtein(x,y) * 8 & \text{otherwise} \end{cases}$$

$$S_A = \sum_{\forall d \in D} \min_{\forall f \in F} distance(d,f)$$

We combine the Hamming and the Levenshtein distance as follows: if the values under comparison are of equal length we use the Hamming distance, otherwise we use the Levenshtein distance. While we apply the Hamming distance at the bit level, the Levenshtein distance calculates the minimum number of single-character edits. In the latter case, to obtain the number of different bits, we simply map one character to bits by multiplying it with 8. We note that this distance metric does not provide a precise measurement, but we believe it provides a useful estimation of the amount of information *contained* in each difference. Moreover, we note that BayesDroid [42] also applied the Hamming and Levenshtein distances, although only on strings of the same length, to provide a rough indication on how much information is contained in a given leak. Both metrics share the very same intuition and, therefore, provide a similar numeric result.

VI. EVALUATION

For our evaluation, we first performed an experiment to characterize non-determinism in network traffic and demonstrate the importance of leveraging contextual information when applying differential analysis to the network traffic of mobile apps. Second, we compared the results of our technique with existing tools showing that AGRIGENTO outperformed all of them, and identified leaks in several apps that no other tool was able to detect. Then, we describe the results of our analysis on current popular apps and present some interesting case studies describing the stealthy mechanisms apps use to leak private information. Finally, we assess the performance of AGRIGENTO in terms of runtime.

A. Experiment Setup

We performed our experiments on six Nexus 5 phones, running Android 4.4.4, while we deployed AGRIGENTO on a 24 GB RAM, 8-core machine, running Ubuntu 16.04. The devices and the machine running AGRIGENTO were connected to the same subnet, allowing AGRIGENTO to capture the generated network traffic.

We chose to perform our experiments on real devices since emulators can be easily fingerprinted by apps and ad libraries [30], [43]. Especially ad libraries are likely to perform emulator detection as ad networks, such as Google's AdMob [21], encourage the use of test ads for automated testing to avoid inflating ad impressions. By using real devices instead of emulators our evaluation is thus more realistic. Furthermore, we set up a Google account on each phone to allow apps to access the Google Play Store and other Google services.

For each execution, we run an app for 10 minutes using Monkey [3] for UI stimulation. We provide Monkey with a fixed seed so that its interactions are the same across runs. Although the fixed seed is not enough to remove all randomness from the UI interactions, it helps to eliminate most of it. At the end of each run, we uninstall the app and delete all of its data.

B. Datasets

We crawled the 100 most popular free apps across all the categories from the Google Play Store in June 2016. Additionally, we randomly selected and downloaded 100 less popular apps. We distinguish between those two datasets based on the intuition that these two sets of apps might differ significantly in their characteristics and overall complexity.

In order to compare our approach with existing techniques, we also obtained the dataset from the authors of ReCon [35], which they used to compare their approach to state-of-the-art static and dynamic data flow techniques. This dataset contains the 100 most popular free apps from the Google Play Store in August 2015 and the 1,000 most popular apps from the alternative Android market AppsApk.com. Ultimately, we use 750 of those apps for analysis, since those apps were the ones that produced any network traffic in ReCon's experiments. We further obtained the dataset of BayesDroid [42], which contains 54 of the most popular apps from the Google Play Store in 2013.

C. Characterizing Non-Determinism in Network Traffic

One key aspect of our work is being able to characterize and explain non-determinism in network traffic. In fact, we want to distinguish what changes "no matter what" and what changes "exactly because we modified the input." First, we show that trivially applying approaches based on differential analysis is ineffective when applied to modern Android apps. Second, our technique allows us to pinpoint which apps are problematic, i.e., for which apps we cannot determine why the network output changes. In this case, we cannot reliably correlate the differences in output with the differences in input and, therefore, we flag them as potentially leaking private information. We note that we can adopt this conservative aggressive policy only because we rarely encounter inexplicable differences in the network traffic of apps that do *not* leak private information. In other words, changes in network traffic that cannot be explained by our system are strong indicators that private information is leaked.

To demonstrate how poorly a naïve differential analysis approach without considering any network-based non-determinism would perform, we analyzed the 100 popular Google Play apps from the ReCon dataset twice: the first time, we trivially applied the differential analysis *without* leveraging any contextual information; the second time, instead, we applied our full approach, executing the apps in our instrumented environment and exploiting the collected contextual information. In both cases, we measured the number of runs needed to converge, setting 20 as the maximum number of runs.

Figure 6 shows the cumulative distribution functions of the number of runs required to reach convergence in the two scenarios. While in the first case almost all the apps did not reach convergence (within a maximum number of 20 runs), our approach correctly handled most of the cases. This demonstrates two things: (1) network traffic is very often non-deterministic, (2) in most cases, the contextual information recorded during the app's analysis is enough to determine the real source of non-determinism.

In order to further confirm this finding, we evaluated how the number of runs per app affects the number of apps for which

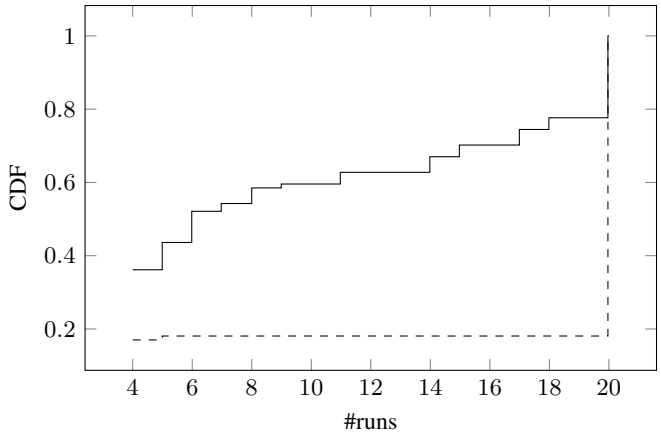

Fig. 6. Cumulative distribution function (CDF) of the number of runs required for convergence (for $K = 3$) applying AGRIGENTO's full approach (solid line), and the trivial differential analysis approach (dashed line) that does not consider any non-determinism in the network behavior.

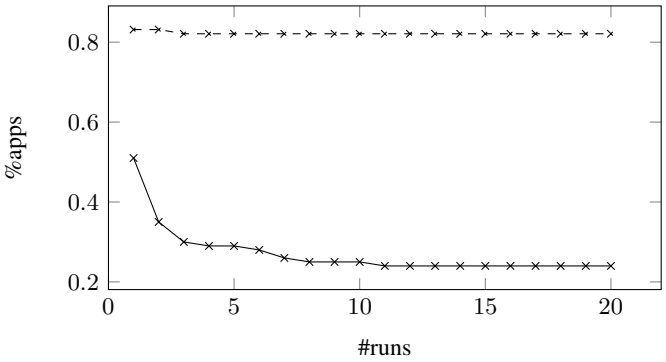

Fig. 7. Percentage of apps with non-deterministic network traffic in an increasing number of runs when applying AGRIGENTO's full approach (solid line), and the trivial differential analysis approach without leveraging contextual information (dashed line).

AGRIGENTO cannot completely explain some source of non-determinism. To do so, we performed a final execution *without* altering any source of private information, and measured the number of apps that contained non-determinism in the network traffic (i.e., the number of apps for which AGRIGENTO raised an alert). Figure 7 shows that, in contrast to our full approach, when applying the differential analysis trivially, increasing the number of runs is not enough to reduce non-determinism (82.1% of the apps generated non-deterministic network traffic).

Finally, we evaluated how the choice of K (i.e., the number of consecutive runs without discrepancies considered to reach convergence) affects AGRIGENTO's ability to explain non-determinism. We performed the evaluation on two datasets: the 100 most popular apps from the Google Play Store and 100 randomly selected less popular apps from the Google Play Store. We run the analysis *without* altering any source of private information. By doing this, any alert is caused by the fact that there is some non-determinism in the network traffic that AGRIGENTO could not explain. Table I shows that $K = 3$ minimizes the number of apps with unexplained non-determinism in their network traffic, at the cost of a small increase in the average number of runs required per app. This evaluation also shows that the popular apps indeed seem to

TABLE I. CHOICE OF K (= NUMBER OF CONSECUTIVE RUNS TO REACH CONVERGENCE) AND ITS EFFECT ON THE AVERAGE NUMBER OF RUNS PER APP, AND NUMBER OF APPS WITH NON-DETERMINISM IN THE NETWORK TRAFFIC THAT AGRIGENTO CANNOT EXPLAIN.

K	Popular		Non-Popular		All	
	#apps	avg #runs	#apps	avg #runs	#apps	avg #runs
1	39	6.02	16	3.10	55	4.56
2	30	8.28	14	4.44	44	6.36
3	**28**	**9.85**	**11**	**5.67**	**39**	**7.76**
4	28	12.42	11	6.78	39	9.60
5	28	13.82	11	8.01	39	10.92

be more complex than the randomly selected ones, for which AGRIGENTO required a lower number of runs on average and could fully explain all sources of non-determinism in more cases overall.

D. Comparison with Existing Tools

To evaluate our approach and establish the presence of false positives and false negatives, we compared AGRIGENTO to existing state-of-the-art analysis tools. Generally, comparing the results of this kind of systems is far from trivial, the main problem being the absence of ground truth. Also, especially in the case of obfuscated leaks, the detected information leaks are often hard to verify by looking at the network traffic alone. Therefore, we manually reverse engineered the apps to the best of our ability to confirm our results. Finally, dynamic analysis results are influenced by limited coverage and different UI exploration techniques, which impedes the comparison.

The only currently available benchmark for privacy leak detection is the DroidBench[2] test suite, which is commonly used to evaluate approaches based on both static and dynamic analysis. We found, however, that it contains very few test cases for dynamic analysis, and those focus mainly on emulator detection (not affecting us since we run our evaluation on real devices). It also does not address complex obfuscation scenarios such as the ones we observed in this work, and, thus, none of the test cases are appropriate for the evaluation of AGRIGENTO.

We thus performed the comparison against existing tools using two datasets on which related work was evaluated: 750 apps from ReCon, and 54 apps from BayesDroid.

ReCon dataset. A similar comparison to evaluate state-of-the-art analysis tools from different categories (static taint analysis, dynamic taint analysis, and a combination of both) has been performed recently to evaluate ReCon [35], which itself is based on network flow analysis. Table II shows the comparison between our tool and AppAudit [46], Andrubis [26] (which internally uses TaintDroid [13]), FlowDroid [6], and ReCon. We base our comparison on the number of apps flagged by each tool for leaking information. For the comparison we considered the following sources of private information: Android ID, IMEI, MAC address, IMSI, ICCID, location, phone number, and contacts.

Compared to ReCon, AGRIGENTO detected 165 apps that ReCon did not identify, while it did not flag 42 apps that

[2]https://github.com/secure-software-engineering/DroidBench

TABLE II. COMPARISON OF AGRIGENTO WITH EXISTING TOOLS ON THE RECON DATASET (750 APPS)

Tool (Approach)	#Apps detected
FlowDroid (Static taint analysis)	44
Andrubis/TaintDroid (Dynamic taint analysis)	72
AppAudit (Static & dynamic taint flow)	46
ReCon (Network flow analysis)	155
AGRIGENTO	278

ReCon identified. We manually checked the results to verify the correctness of our approach. Among the 42 AGRIGENTO did not detect, 23 did not generate any network traffic during our analysis. This may be due to different reasons, for instance different UI exploration (ReCon manually explored part of the dataset), or because the version of the app under analysis does not properly work in our test environment. We manually inspected the network traffic generated by the remaining 19 apps. In particular, we manually verified whether each network trace contained any of the values of the sources of private information that we considered, and we also checked for known transformations, such as MD5 hashes and Base64 encoding. In all cases, we did not identify any leak (i.e., we did not identify any false negatives). We acknowledge that this manual evaluation does not exclude the presence of false negatives. However, we consider this an encouraging result nonetheless.

To perform a more thorough evaluation of false negatives, we also performed an additional experiment. Since one main challenge when comparing approaches based on dynamic analysis is related to GUI exploration differences, we asked the authors of ReCon to run their tool on the network traffic dumps we collected during our analysis. In this way, it is possible to compare both tools, ReCon and AGRIGENTO, on the same dynamic trace. On this dataset, ReCon flagged 229 apps for leaking information. AGRIGENTO correctly detected all the apps identified by ReCon, and, in addition, it detected 49 apps that ReCon did not flag. This evaluation shows that, also for this experiment, AGRIGENTO did not show any false negatives. Moreover, we also looked for false positives, and we manually verified the 49 apps detected by AGRIGENTO and not by ReCon. Our manual analysis revealed that 32 of the 49 apps did indeed leak at least one source of private information, which should then be considered as true positives (and false negatives for ReCon). For further 5 apps we could not confirm the presence of a leak and thus classify them as false positives produced by our system. We cannot classify the remaining 12 cases as either true or false positives because of the complexity of reversing these apps.

BayesDroid dataset. We obtained the dataset used by BayesDroid and analyzed the apps with AGRIGENTO. For the comparison we considered the common sources of information supported by both AGRIGENTO and BayesDroid (i.e., IMEI, IMSI, Android ID, location, contacts). BayesDroid flagged 15 of the 54 apps. However, since this dataset contains older app versions (from 2013) 10 apps did not work properly or did not generate any network traffic during our analysis. Nevertheless, AGRIGENTO flagged 21 apps, including 10 of the 15 apps identified by BayesDroid. As we did for the ReCon dataset, we manually looked at the network traces of the remaining

TABLE III. NUMBER OF APPS DETECTED BY AGRIGENTO IN THE 100 MOST POPULAR APPS (JULY 2016) FROM THE GOOGLE PLAY STORE. THE COLUMN "ANY" REFERS TO THE NUMBER OF APPS THAT LEAK AT LEAST ONE OF THE PRIVATE INFORMATION SOURCES.

Results		Any	Android ID	IMEI	MAC Address	IMSI	ICCID	Location	Phone Number	Contacts
TPs	Plaintext	31	30	13	5	1	0	1	0	0
	Encrypted	22	18	9	3	5	0	0	0	0
	Obfuscated	11	8	5	6	0	0	1	0	0
	Total	42	38	22	11	6	0	1	0	0
FPs		4	5	9	11	13	13	11	16	13

5 apps and we did not see any leak (3 of them did not produce any network traffic, furthermore BayesDroid used manual exploration of all apps). Interestingly, AGRIGENTO detected 11 apps that BayesDroid did not. We found that 6 of these apps used obfuscations that BayesDroid does not detect. For instance, one app included the InMobi SDK that performs a series of encodings and encryptions on the Android ID before leaking it. We describe this case in detail in §VI-F. Moreover, the other 5 apps used Android APIs to hash or encrypt data structures (e.g., in JSON format) containing private information sources, again showing that our system detects cases that previous work cannot.

E. Privacy Leaks in Popular Apps

To evaluate AGRIGENTO on a more recent dataset, we analyzed the current (July 2016) 100 most popular apps from the Google Play Store in more detail. AGRIGENTO identified privacy leaks in 46 of the 100 apps. We manually verified the results of our analysis to measure false positives. We found that 42 of these apps are true positives, that is, they leak private information, while four apps were likely false positives. Note that, in some cases, to distinguish true positives from false positives we had to manually reverse the app. During our manual analysis, we did not encounter any false negative. Once again, we acknowledge that, due to the absence of a ground truth, it is not possible to fully exclude the presence of false negatives. In particular, as further discussed in §VII, AGRIGENTO is affected by a number of limitations, which a malicious app could take advantage of.

We then used our risk analysis to rank the risk associated with these false positives. Interestingly, we found that while two of the four apps that caused false positives have high scores (i.e., 8,527 and 8,677 bits), for the other two apps, one in particular, AGRIGENTO assigned low scores of 6 and 24 bits. We note that although for this work we use our risk analysis only to rank the risk of a data leak in each detected app, we believe it could be used to build, on top of it, a further filtering layer that discards low bandwidth leaks. We will explore this direction in future work.

We further classified the type of leak in three groups: plaintext, encrypted, and obfuscated. The first group contains apps that leak the information in plaintext. The second group contains apps for which we observed the leaked information only after our decryption phase (i.e., the leaked value has been encrypted or hashed using the Android APIs). Finally, the third group contains apps that obfuscate information leaks by other means (i.e., there is no observable evidence of the leaked value in the network traffic).

As a first experiment, we considered leaks only at the app level since we are interested in determining whether an app leaks information or not, independently from the number of times. In other words, we are interested to determine whether a given app leaks any sensitive information. Thus, for each app analysis we performed just one final run for which we modified all the sources simultaneously. As a result, AGRIGENTO produces a boolean output that indicates whether an app leaks private information or not, without pointing out which particular source has been leaked. Table III shows the results of this experiment. For this experiment, we consider an app as a true positive when it leaks any of the monitored sources and AGRIGENTO flags it, and as a false positive when AGRIGENTO flags it although it does not leak any information.

While this experiment provides valuable insights, it provides only very coarse-grained information. Thus, as a second experiment, we performed the same evaluation but we looked at each different source of information individually. In this case, we ran the app and performed the differential analysis changing only one source at a time, and we consider an app as a true positive only if it leaks information from the *modified* source and AGRIGENTO correctly identifies the leak. Our evaluation shows that, while AGRIGENTO produces higher false positives in identifying leaks for a specific source of information, it has very few false positives in detecting privacy leaks in general. The higher false positive rate is due to some sources of non-determinism that AGRIGENTO failed to properly handle and that consequently cause false positives when an app does not leak data. For instance, consider the scenario in which an app leaks the Android ID and also contains some non-determinism in its network traffic that AGRIGENTO could not eliminate. In this case, when considering leaks at app-level granularity, we consider the app as a true positive for the Android ID, since it does leak the Android ID. Instead, for any other source of information (e.g., the phone number) we consider the app as a false positive because of the non-determinism in the network traffic. Finally, we could not classify 9 apps, for which AGRIGENTO identified leaks of some of the sources, because of the complexity of reversing these apps.

F. Case Studies

We manually reversed some apps that AGRIGENTO *automatically* identified as leaking obfuscated or encrypted information. Here, we present some case studies showing that current apps use sophisticated obfuscation and encryption techniques. Hence, as confirmed by the results of our evaluation, state-of-the-art solutions to identify privacy leaks are not enough since they do not handle these scenarios and mostly only consider standard encodings.

```
http://i.w.inmobi.com/showad.asm?u-id-map=iB7WTkCLJv
NsaEQakKKXFhk8ZEIZlnL0jqbbYexcBAXYHH4wSKyCDWVfp+q+Fe
LFTQV6jS2Xg97liEzDkw+XNTghe9ekNyMnjypmgiu7xBS1TcwZmF
xYOjJkgPOzkI9j2lryBaLlAJBSDkEqZeMVvcjcNkx+Ps6SaTRzBb
Yf8UY=&u-key-ver=2198564
```

```
https://h.online-metrix.net/fp/clear.png?ja=33303426
773f3a3930643667663b33383831303d343526613f2d36383024
7a3f363026663d333539347a31323838266c603d687c76722531
6325306625306616f6e74656e762f6a732c746370626f792663
6f652466723f6a74767025316127326625326661 6d6d2e65616f
656b69726b7573267270697867636e617730266a683d65616437
613732316431353c65613a31386e67606563303736363963434
3363266d64643f6561633336303b64336a393531666330366663
61373261363a6161633563676126 6d6733f353b32306d383230
613230643b6534643934383a31663636623b3232376761612661
6d65613d313933333133333133313133313331266174 3d636565
6e765f6f6f6a696c6d26617e3f767277717466656667 6e666572
2b6d6f606b6c652733632b392e3226342d3b
```

Fig. 8. Example of the requests performed by InMobi and ThreatMetrix libraries. InMobi leaks the Android ID, as described in §VI-F, in the value of `u-id-map`. ThreatMetrix leaks the Android ID, location, and MAC address in the `ja` variable.

Interestingly, all the leaks we found in these case studies were performed by third-party libraries, and thus may concern all the apps using those libraries.

Case study 1: InMobi. We found that InMobi, a popular ad library, leaks the Android ID using several layers of obfuscation techniques. The Android ID is hashed and XORed with a randomly generated key. The XORed content is then encoded using Base64 and then stored in a JSON-formatted data structure together with other values. The JSON is then encrypted using RSA (with a public key embedded in the app), encoded using Base64 and sent to a remote server (together with the XOR key). Figure 8 shows an example of such a request leaking the obfuscated Android ID. AGRIGENTO automatically identified 20 apps in our entire dataset leaking information to InMobi domains, including one app in the 100 most popular apps from the Google Play Store. Indeed, according to AppBrain[3], InMobi is the fourth most popular ad library (2.85% of apps, 8.37% of installs).

Case study 2: ThreatMetrix. The analytics library ThreatMetrix leaks multiple sources of private information using obfuscation. It first puts the IMEI, location, and MAC address in a HashMap. It then XORs this HashMap with a randomly generated key, hex-encodes it, and then sends it to a remote server. Figure 8 shows an example of such a request leaking the obfuscated Android ID, location, and MAC address. We found 15 instances of this scenario in our entire dataset, one of which is part of the 100 most popular apps from the Google Play Store. According to AppBrain, ThreatMetrix SDK is used by 0.69% of the apps in the Google Play Store, and is included by 4.94% of the installs.

Further ad libraries. We found several other apps and ad libraries (MobileAppTracking, Tapjoy) leaking private information using the Android encryption and hashing APIs. In the most common scenario, the values are combined in a single string that is then hashed or encrypted. In this scenario, even though the app uses known encodings or cryptographic functions, previous tools are not able to detect the leak of private information.

[3]http://www.appbrain.com/stats/libraries/ad

G. Performance Evaluation

We execute each app for 10 minutes during each run. The analysis time per app mainly depends on the complexity of the app (i.e., the number of runs required to reach convergence). Setting $K = 3$, AGRIGENTO analyzed, on average, one app in 98 minutes. Note that, while we executed each run sequentially, our approach can easily scale using multiple devices or emulators running the same app in parallel.

VII. LIMITATIONS AND FUTURE WORK

While we addressed the major challenges for performing differential analysis despite the overall non-determinism of the network traffic of mobile apps, our overall approach and the implementation of AGRIGENTO still have some limitations.

Even though AGRIGENTO improves over the existing state-of-the-art, it still suffers from potential false negatives. For example, as any other approach relying on the actual execution of an app, AGRIGENTO suffers from limited code coverage, i.e., an app might not actually leak anything during the analysis, even if it would leak sensitive data when used in a real-world scenario. This could happen for two main reasons: (a) An app could detect that it is being analyzed and does not perform any data leaks. We address this issue by performing our analysis on real devices; (b) The component of the app that leaks the data is not executed during analysis, for example due to missing user input. We currently use Monkey, which only generates pseudorandom user input and cannot bypass, for example, login walls. Related works such as BayesDroid and ReCon performed manual exploration of apps at least for part of the dataset, which also included providing valid login credentials. Unfortunately, manual exploration is only feasible for small-scale experiments and not on a dataset of over one thousand apps such as ours, especially given the fact that AGRIGENTO needs to generate the same consistent user input over multiple executions. As part of our future work, we are planning to explore whether it is possible to provide manual inputs for the first run of an app, and then replaying the same input with tools such as RERAN [20] in the subsequent runs. One option for collecting the initial manual inputs at scale is Amazon Mechanical Turk.

Second, AGRIGENTO still suffers from some covert channels that an attacker could use to leak information without being detected. For instance, a sophisticated attacker could leak private information by encoding information in the number of times a certain request is performed. However, this scenario is highly inefficient from the attacker point of view. Furthermore, we could address this issue with a more accurate description of the "network behavior summary." As a matter of fact, AGRIGENTO severely limits the bandwidth of the channel an attacker can use to stealthily trasmit private data.

We need to run each app multiple times: by nature, an approach using differential analysis requires at least two executions, one with the original inputs, and another one with different inputs to observe changes in the outputs. As we discussed in our evaluation, the non-deterministic network behavior of modern apps further requires us to perform the original execution more than once to build a more accurate network behavior summary. Since we conservatively flag any changes in the output as a possible leak, in practice the number of runs is a trade-off between the overall analysis time and the false positive rate.

Furthermore, we perform the final run once for each source of private information that we track. This requirement could be relaxed if our goal was to find privacy leaks in general, and not specific types of information. In our evaluation we performed all runs of a specific app consecutively on the same device. We could parallelize this process on different devices, however, with less control over device-specific artifacts that could potentially influence our analysis.

On the implementation side we suffer from two main limitations: First, we currently do not instrument calls to `/dev/random`, which could be used by native code directly as a source of randomness. We leave this issue for future work. Second, we are limited by the protocols we track: we only check HTTP GET and POST requests for leaks (and man-in-the-middle HTTPS even with certificate pinning in most cases). However, we share this limitation with other tools, such as ReCon, and leave an extension of AGRIGENTO to other protocols for future work.

By design, AGRIGENTO can only determine that a specific piece of private information was leaked, but not automatically determine how it was obfuscated. We can, however, perform the naïve approach employed by related tool of simple grepping for widely-used encodings and hashing algorithms of the value, to filter out those cases and focus manual reverse engineering efforts on the more complex and interesting ones.

Finally, we can only speculate why app developers are adopting the stealth techniques that we have uncovered in our analysis. This development could be related to the increasing awareness and opposition of users to the collection of their private data, as well as the investigative efforts of regulators such as the FTC. Currently, InMobi is very open about the data it collects in its privacy policy.[4] For future work we could investigate any malicious intent or deceptive practice behind sophisticated obfuscation techniques, based on automatically verifying whether those leaks are in violation of an app's privacy policy or not. Related work in this direction by Slavin [38] has so far only compared privacy policies against information flows identified with FlowDroid, but has not considered cases in which apps are hiding their leaks with the techniques AGRIGENTO uncovered.

VIII. RELATED WORK

Static taint analysis of Android apps is an active research topic, as several aspects of Android apps proved to be very challenging—in particular their component-based architecture and the multitude of entry points due to their user-centric nature and complex lifecycle. AndroidLeaks [19] was one of the first static taint analysis approaches, but lacks precision as it tracks data flow at the object-level instead of tainting their individual fields. FlowDroid [6] is more precise in this regard and one of the most widely used static taint analysis tools. Further approaches include EdgeMiner [9], which addresses the issue of reconstructing implicit control flow transitions, and Amandroid [44] and IccTA [25], which deal with inter-component instead of just intra-component data leaks. MorphDroid [17] argues that conventional data flow tracking approaches are too coarse-grained, and tracks atomic units of private information instead of the complete information (i.e., longitude and latitude instead of the location) to account for partial leaks. AppIntent [48] proposes to distinguish between user-intended and covert data leaks and uses symbolic execution to determine if a privacy leak is a result of user interaction. AppAudit [46] addresses the false positives of related static analysis approaches and verifies the detected leaks through approximated execution of the corresponding functions.

Dynamic taint analysis tracks information flow between sources of private information and sinks, e.g., the network, during runtime, either by modifying the device OS (TaintDroid [13]), the platform libraries (Phosphor [8]), or the app under analysis (Uranine [33]). AppFence [22] extends TaintDroid to detect obfuscated and encrypted leaks, and also performed a small-scale study on the format of leaks, but only found the ad library Flurry leaking data in non-human readable format in 2011— a situation that has drastically changed since then as we showed in our study. BayesDroid [42] is similar to TaintDroid, but addresses the problem of partial information leaks. It compares tainted data tracked from a source of private information to a network sink, and uses probabilistic reasoning to classify a leak based on the similarity between the data at both points. While aforementioned approaches only track data flow in the Dalvik VM, there also exist approaches that also can track data flow in native code: DroidScope [47] and CopperDroid [41] perform full system emulation and inspect both an app's Dalvik and native code for the purpose of malware analysis, while the recent TaintART extends TaintDroid to native code [40]. However, ultimately, taint analysis approaches are vulnerable to apps deliberately disrupting the data flow: ScrubDroid [36] discusses how dynamic taint analysis systems for example can be defeated by relying on control dependencies (which related approaches usually do not track), or by writing and reading a value to and from system commands or the file system.

Most recently, related work has explored detecting privacy leaks at the network level, usually through network traffic redirection by routing a device's traffic through a virtual private network (VPN) tunnel and inspecting it for privacy leaks on the fly. Tools such as PrivacyGuard [39], AntMonitor [24], and Haystack [34], perform their analysis on-device using Android's built-in VPNService, but rely on hardcoded identifiers, or simply grep for a user's private information. Liu et al. [27] inspect network traffic at the ISP-level and identify private information leaks based on keys generated from manual analysis and regular expressions. Encryption and obfuscation are out of scope of the analysis, as the authors assume this scenario is only a concern for malware. ReCon [35] is another VPN-based approach, which uses a machine learning classifier to identify leaks and can deal with simple obfuscation. In the end, it relies on the data on which it is trained on—which can come from manual analysis and dynamic taint analysis tools—and it could benefit from a technique such as AGRIGENTO to deal with more complex obfuscation techniques.

Information leakage is not a new problem and not unique to Android apps: related work on desktop applications has focused on identifying (accidental) leaks of private information through differential analysis at the process-level. TightLip [49] and Croft et al. [10] perform differential analysis on the output of two processes, one with access to private data, and one without. Both consider timestamp-related information and random seeds as sources of non-determinism and share them between processes.

[4] http://www.inmobi.com/privacy-policy/

Ultimately, their main goal is to prohibit the accidental leakage of private information, more specifically, sensitive files, and not obfuscated content. To this end, TightLip checks if the system call sequences and arguments of the two processes diverge when the private input changes, and consequently raises an alarm if the output is sent to a network sink. In contrast, Croft et al. only allow the output of the process without access to private information to leave the internal company network. The approach of Privacy Oracle [23] is related to AGRIGENTO: it identifies privacy leaks based on divergences in the network traffic when private input sources are modified. However, it mainly addresses non-determinism at the OS-level (i.e., performing deterministic executions using OS snapshots) and does not consider non-determinism in network traffic. In fact, it cannot handle random tokens in the network traffic, nor encryption, and produces false positives when messages in network flows are reordered between executions.

Finally, Shu et al. [37] propose a sequence alignment algorithm for the detection of obfuscated leaks in files and network traffic, which assigns scores based on the amount of private information they contain. While this approach focuses on the detection of obfuscated leaks, it explicitly does not address intentional or malicious leaks, and only considers character replacement, string insertion and data truncation.

In contrast to related work, we are the first to address the topic of obfuscation of privacy leaks in order to deal with adversaries, i.e., apps or ad libraries actively trying to hide the fact that they are leaking information. As we have shown in our evaluation, this is a very realistic threat scenario and a practice that is already common amongst popular mobile apps and ad libraries.

IX. CONCLUSION

We showed that while many different approaches have tackled the topic of privacy leak detection in mobile apps, it is still relatively easy for app and ad library developers to hide their information leaks from state-of-the-art tools using different types of encoding, formatting, and cryptographic algorithms. This paper introduces AGRIGENTO, a new approach that is resilient to such obfuscations and, in fact, to any arbitrary transformation performed on the private information before it is leaked. AGRIGENTO works by performing differential black-box analysis on Android apps. We discussed that while this approach seems intuitive, in practice, we had to overcome several key challenges related to the non-determinism inherent to mobile app network traffic.

One key insight of this work is that non-determinism in network traffic can be often explained and removed. This observation allowed us to develop novel techniques to address the various sources of non-determinism and it allowed us to conservatively flag any deviations in the network traffic as potential privacy leaks. In our evaluation on 1,004 Android apps, we showed how AGRIGENTO can detect privacy leaks that state-of-the-art approaches cannot detect, while, at the same time, only incurring in a small number of false positives. We further identified interesting cases of custom and complex obfuscation techniques, which popular ad libraries currently use to exfiltrate data without being detected by other approaches.

ACKNOWLEDGEMENTS

We would like to thank our reviewers and our shepherd Matt Fredrikson for their valuable comments and input to improve our paper. We would also like to thank Jingjing Ren for her help comparing against ReCon, as well as Antonio Bianchi and David Choffnes for their insightful feedback. This work was supported in part by the MIUR FACE Project No. RBFR13AJFT. This material is also based upon work supported by the NSF under Award No. CNS-1408632, by SBA Research, and a Security, Privacy and Anti-Abuse award from Google. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF, SBA Research, or Google. This material is also 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] "JustTrustMe," https://github.com/Fuzion24/JustTrustMe.

[2] "mitmproxy," https://mitmproxy.org.

[3] "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html.

[4] "Xposed framework," http://repo.xposed.info.

[5] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupe, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy," in *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[6] 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 *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[7] M. Beddoe, "The Protocol Informatics Project," http://www.4tphi.net/~awalters/PI/PI.html, 2004.

[8] J. Bell and G. Kaiser, "Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs," in *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014.

[9] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," in *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[10] J. Croft and M. Caesar, "Towards Practical Avoidance of Information Leakage in Enterprise Networks," in *Proc. of the USENIX Conference on Hot Topics in Security (HotSec)*, 2011.

[11] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "NetworkProfiler: Towards Automatic Fingerprinting of Android Apps," in *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013.

[12] G. Eisenhaur, M. N. Gagnon, T. Demir, and N. Daswani, "Mobile Malware Madness, and How to Cap the Mad Hatters. A Preliminary Look at Mitigating Mobile Malware," in *Black Hat USA (BH-US)*, 2011.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[14] Federal Trade Commission, "FTC Approves Final Order Settling Charges Against Flashlight App Creator," https://www.ftc.gov/news-events/press-releases/2014/04/ftc-approves-final-order-settling-charges-against-flashlight-app, April 2014.

[15] ——, "Two App Developers Settle FTC Charges They Violated Children's Online Privacy Protection Act," https://www.ftc.gov/news-events/press-releases/2015/12/two-app-developers-settle-ftc-charges-they-violated-childrens, December 2015.

[16] ——, "Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission," https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked, June 2016.

[17] P. Ferrara, O. Tripp, and M. Pistoia, "MorphDroid: Fine-grained Privacy Verification," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[18] M. N. Gagnon, "Hashing IMEI numbers does not protect privacy," http://blog.dasient.com/2011/07/hashing-imei-numbers-does-not-protect.html, 2011.

[19] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," in *Proc. of the International Conference on Trust and Trustworthy Computing (TRUST)*, 2012.

[20] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and Touch-Sensitive Record and Replay for Android," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2013.

[21] Google, "AdMob Behavioral Policies," https://support.google.com/admob/answer/2753860?hl=en, 2016.

[22] 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 *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[23] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[24] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou, "AntMonitor: A System for Monitoring from Mobile Devices," in *Proc. of the ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big Internet Data (C2BID)*, 2015.

[25] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2015.

[26] 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 *Proc. of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[27] Y. Liu, H. H. Song, I. Bermudez, A. Mislove, M. Baldi, and A. Tongaonkar, "Identifying Personal Information in Internet Traffic," in *Proc. of the ACM Conference on Online Social Networks (COSN)*, 2015.

[28] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android System," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[29] S. B. Needleman and C. D. Wunsch, "A General Method Applicable to Search for Similarities in Amino Acid Sequence of Two Proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[30] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware," in *Proc. of the European Workshop on System Security (EuroSec)*, 2014.

[31] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," 2014.

[32] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[33] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, Y. Chen, W. Zhu, and W. Chen, "Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android," in *Proc. of the International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2015.

[34] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson, "Haystack: In Situ Mobile Traffic Analysis in User Space," *arXiv preprint arXiv:1510.01419*, 2015.

[35] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic," in *Proc. of the International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2016.

[36] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar, "On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices," in *Proc. of the International Conference on Security and Cryptography (SECRYPT)*, 2013.

[37] X. Shu, J. Zhang, D. D. Yao, and W. C. Feng, "Fast Detection of Transformed Data Leaks," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 3, pp. 528–542, March 2016.

[38] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a Framework for Detecting Privacy Policy Violations in Android Application Code," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2016.

[39] Y. Song and U. Hengartner, "PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices," in *Proc. of the Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2015.

[40] M. Sun, T. Wei, and L. John, "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

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

[42] O. Tripp and J. Rubin, "A Bayesian Approach to Privacy Enforcement in Smartphones," in *Proc. of the USENIX Security Symposium*, 2014.

[43] T. Vidas and N. Christin, "Evading Android Runtime Analysis via Sandbox Detection," in *Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[44] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[45] G. Wondracek, P. Milani Comparetti, C. Kruegel, and E. Kirda, "Automatic Network Protocol Analysis," in *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2008.

[46] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective Real-time Android Application Auditing," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

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

[48] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[49] A. R. Yumerefendi, B. Mickle, and L. P. Cox, "TightLip: Keeping Applications from Spilling the Beans," in *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[50] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services," in *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.