Difuzer: Uncovering Suspicious Hidden Sensitive Operations in Android Apps

Jordan Samhi[1], Li Li[2], Tegawendé F. Bissyandé[1], Jacques Klein[1]
[1]SnT, University of Luxembourg, Luxembourg, firstname.lastname@uni.lu
[2]Monash University, Australia, firstname.lastname@monash.edu

ABSTRACT

One prominent tactic used to keep malicious behavior from being detected during dynamic test campaigns is *logic bombs*, where malicious operations are triggered only when specific conditions are satisfied. Defusing logic bombs remains an unsolved problem in the literature. In this work, we propose to investigate Suspicious Hidden Sensitive Operations (SHSOs) as a step towards triaging logic bombs. To that end, we develop a novel hybrid approach that combines static analysis and anomaly detection techniques to uncover SHSOs, which we predict as likely implementations of logic bombs. Concretely, Difuzer identifies SHSO entry-points using an instrumentation engine and an inter-procedural data-flow analysis. Then, it extracts trigger-specific features to characterize SHSOs and leverages One-Class SVM to implement an unsupervised learning model for detecting abnormal triggers.

We evaluate our prototype and show that it yields a precision of 99.02% to detect SHSOs among which 29.7% are logic bombs. Difuzer outperforms the state-of-the-art in revealing more logic bombs while yielding less false positives in about one order of magnitude less time. All our artifacts are released to the community.

ACM Reference Format:
Jordan Samhi[1], Li Li[2], Tegawendé F. Bissyandé[1], Jacques Klein[1]. 2022. Difuzer: Uncovering Suspicious Hidden Sensitive Operations in Android Apps . In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510135

1 INTRODUCTION

Security and privacy in Android have become paramount given its pervasive use in a wide range of user devices, be it handheld, at home, or in the office [34]. Yet, regularly, new threats are discovered, even in the official Google Play app store [18]. Typically, thousands of apps are regularly flagged by antivirus engines: for the year 2020 alone, the AndroZoo [4] repository has collected over 228 000 apps, among which over 10 000 apps are flagged by at least five antivirus engines hosted by VirusTotal. Addressing the spread of malware in app markets is therefore a prime concern for researchers and practitioners. In the last decade, several approaches have been proposed in the literature to automate malware identification. These approaches explore static analysis techniques [24, 25, 37, 58, 85], dynamic execution [60, 77, 86], or a combination of both [11, 17, 81], as well as the use of machine-learning [59, 65].

While the aforementioned techniques have been proven effective on benchmarks, attacks evolve rapidly with increasingly sophisticated evasion techniques. Typically, malware writers rely on code obfuscation [20] to bypass static analyses. To evade detection during dynamic analysis, attackers seek to hide malicious code behind triggering conditions. These are known as *logic bombs*, the triggering conditions of which being varied. For example, a logic bomb could execute malicious instructions only at a specific time that is not likely to be reached when market maintainers dynamically analyze the software before it is distributed.

Logic bombs can be used for any malicious activity such as adware [22], trojan [61], ransomware [83], spyware [64], etc. [89]. Furthermore, as the trigger and the malicious code are generally independent of the core application code, logic bombs can easily be added in legitimate apps and repackaged for distribution [27, 42, 44, 88]. Therefore, detecting logic bombs is of great importance, especially in mobile devices that carry much personal information. However, due to the undecidable nature of this detection problem in general [63], and the fact that dynamic analyses will likely fail to detect such behaviors [1], analysts explore static-analysis based heuristic or machine learning approaches to detect logic bombs.

A logic bomb is characterized by the fact that it implements a hidden sensitive operation. Therefore, recent works addressing logic bombs have focused on the identification of Hidden Sensitive Operations (HSOs) as a target [57]. However, not all HSOs are logic bombs. Indeed, an HSO may be neither **intentional** nor **malicious**, while logic bombs always are. In this work, we propose to identify **Suspicious HSOs** (SHSO) towards triaging logic bombs among HSOs. Indeed, we consider that an SHSO is an HSO that is likely implementing a logic bomb. Further note that, in this study, we do not attempt to address a binary classification problem of discriminating malware from benign apps (e.g., by using logic bombs as a key criteria of maliciousness). Instead, our ambition is to improve the detection of logic bombs, which are considered sweet spots for targeting the understanding of malware's malicious behaviors. Indeed, while the literature proposes a variety of approaches for predicting Android apps' maliciousness (i.e., malware detection), the community still seeks to make significant breakthroughs in the location of malicious code parts. Detecting logic bombs thus provides an opportunity to locate and characterize malicious code implemented as hidden sensitive operations.

Recent literature on Android has already approached the problem of detecting sensitive behavior triggered only when certain conditions are met. Such triggers are referred hereafter as *sensitive triggers*. TriggerScope [26] was proposed as a static analysis tool to detect logic bombs: its analyses are based on heuristics and are thus

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510135

limited to certain trigger types (i.e., time-related, location-related, and SMS-related triggers). TriggerScope further relies on symbolic execution, which reduces its capacity to scale to massive datasets. Unlike TriggerScope, HsoMiner [57] leverages a supervised learning approach with engineered features to reveal sensitive triggers. HsoMiner, however, does not specifically target malicious triggers: it flags up to 20% of apps, which makes it inefficient for isolating dangerous triggers in the wild; it also takes on average 13 min/app, which makes it challenging to exploit for large-scale experiments.

HSO triggering conditions are typically implemented by *if statements*. A given app code, however, may contain from hundreds to thousands of such conditional statements. Therefore, a major challenge in the research around HSO is to reduce the search space for accurately spotting suspicious sensitive triggers. Our core idea towards achieving this ambition is to model specific trigger characteristics to spot SHSOs.

In this work, we propose a novel approach to identify suspicious hidden sensitive operations where we rely on an unsupervised learning technique to perform anomaly detection. We intend to detect suspicious triggers deviating from the normality of the myriads of conditional checks performed in typical apps. To do so, we explore specific trigger/behavior features to guide our detection system towards enumerating truly suspicious triggers and thus refine the search space for uncovering logic bombs. We propose Difuzer, a novel hybrid approach that combines ❶ code instrumentation to insert particular statements required for taint analysis, ❷ inter-procedural static taint analysis to find suspicious sensitive triggers, and ❸ anomaly detection to reveal *Suspicious Hidden Sensitive Operations* in Android apps.

While the literature includes work [57] that proposed supervised learning techniques for detecting HSOs, Difuzer relies on unsupervised learning to spot "abnormal" triggers. Moreover, towards ensuring that the model is accurate in the detection of suspicious HSOs, Difuzer leverages features that are specifically-engineered to capture semantic properties of maliciousness.

The main contributions of our work are as follows:

- We propose Difuzer, a novel approach to detect SHSOs in Android apps. Difuzer combines code instrumentation, static inter-procedural taint tracking and anomaly detection techniques.
- We evaluate Difuzer and show its ability to reveal SHSOs with a 99.02% precision in less than 35 seconds on average per app, outperforming previous approaches.
- We demonstrate that the trigger- and behavior-specific features of Difuzer are relevant for triaging logic bombs among HSOs: 29.7% of detected SHSOs are indeed confirmed as logic bombs.
- We compare Difuzer against a state of the art logic bomb detector, TriggerScope: Difuzer reveals more logic bombs than TriggerScope while yielding less false positives.
- We further applied Difuzer on a dataset of "benign" apps from Google Play. By analysing the yielded SHSOs, Difuzer contributed to suspect 8 adware apps, which Google removed from Google Play after we have pointed them out.
- We release the Difuzer prototype in open-source and further make available to the research community the first Android logic bomb dataset, called DataBomb: https://github.com/Trustworthy-Software/Difuzer

2 BACKGROUND AND DEFINITIONS

In this section, we first introduce *Taint Analysis* and *Anomaly Detection*, two techniques used in our approach. Then, we carefully define important concepts and finally succinctly give the context for our study.

Taint analysis: Taint analysis is a dataflow analysis that follows the flow of specific values within a program. A variable V is tainted when it gets a value from specific functions called *sources*. The taint is propagated to other variables if they receive a derivation of the value in V. If a tainted variable is used as a parameter of specific functions called *sinks*, it means that during execution, the value derived from a *source* can be used as a parameter of a *sink*. In this paper's context, we rely on taint analysis to check if the conditional expression involves sensitive data value(s).

Anomaly detection: When analyzing data of the same class, several items can significantly differ from the majority. They are called *outliers* and can be viewed as abnormal. There are numerous techniques in the state-of-the-art for achieving this outlier detection in sets of data [13]. This paper relies on *One-Class Support Vector Machine* (OC-SVM) [69], an unsupervised learning algorithm that learns common behavior based on features extracted in an initial dataset. Once the model is learned, a prediction is performed by checking whether a new sample features make it more or less abnormal w.r.t. the common model. In this paper's context, an anomaly is computed by considering distances among vectors representing *triggers*, i.e., a condition along with the behavior triggered.

Definitions: We define terms that will be used and referred to throughout the paper. Figure 1 visually depicts our definitions.

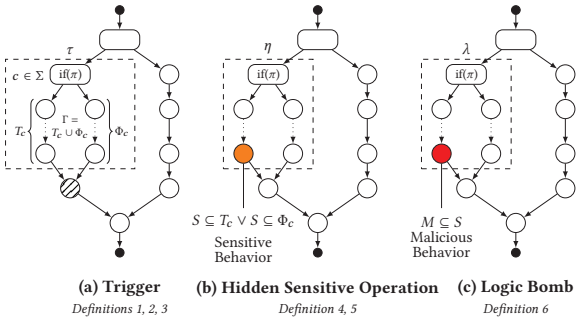

Figure 1: Definitions illustrations. The graphs represent the Control-Flow Graph of the same function.

Definition 1 (Trigger). A trigger is a piece of code that activates operations under certain conditions. In Figure 1a, the trigger τ (dashed rectangle) is represented by the condition c (rounded rectangle node), the true branch T_c and the false branch Φ_c. The true branch T_c represents all the statements (nodes) for which each path from the entry-point must go through c and are executed if and only if π is true. Note that every path from the entry-point to the hatched node must go through c. In other words, c strictly dominates the hatched node. However, the hatched node can be executed if π is true or false. Therefore it is not part of T_c nor Φ_c. The false branch Φ_c represents all the statements for which each path from the entry-point must go through c and are executed if and only if π is false.

More formally, let Σ be the set of statements of a function (nodes in Fig. 1). Let $c \in \Sigma$ be a conditional statement (i.e., an if statement, rectangle nodes in Fig. 1). Let π be c's predicate. Let ε be the conditional execution function such as $\varepsilon(\pi, \sigma)$ is true if $\sigma \in \Sigma$ is executed if and only if π is true. Let δ be the dominator function such as $\delta(d, \sigma)$ is true if $d \in \Sigma$ strictly dominates $\sigma \in \Sigma$, false otherwise. Let T_c and Φ_c be the *true* and the *false* branch [1] of c such as:

$$T_c = \{\sigma \mid \sigma \in \Sigma \wedge \delta(c, \sigma) \wedge \varepsilon(\pi, \sigma)\}$$
$$\Phi_c = \{\sigma \mid \sigma \in \Sigma \wedge \delta(c, \sigma) \wedge \varepsilon(\neg\pi, \sigma)\}$$

Then, a trigger τ is defined as a triplet: $\tau = (c, T_c, \Phi_c)$.

Definition 2 (Guarded code). Let τ be a trigger such as: $\tau = (c, T_c, \Phi_c)$. Then, the code guarded by c is: $\Gamma = T_c \cup \Phi_c$.

Definition 3 (Trigger entry-point). We define a trigger entry-point as the condition triggering the guarded code. More formally, given a trigger $\tau = (c, T_c, \Phi_c)$, c is defined as its entry-point.

Definition 4 (Hidden Sensitive Operation (HSO)). An HSO is a piece of code that represents a set of instructions, which (1) implement a security-sensitive operation and (2) are only executed when specific criteria are met (cf. Figure 1b). More formally, let $\eta = (c, T_c, \Phi_c)$ be a trigger and S a piece of sensitive behavior such as $S \subset \Sigma$. Then, η is a hidden sensitive operation if $S \subseteq T_c \vee S \subseteq \Phi_c$.

Definition 5 (Suspicious Hidden Sensitive Operation (SHSO)). An SHSO refers to an HSO that implements a sensitive operation that appears to be suspicious given the context of the app. For example, a navigation app may legitimately retrieve user location information (which is a sensitive operation), while a calculator is suspicious if it attempts to retrieve such sensitive data.

Definition 6 (Logic bomb). A logic bomb is a piece of malicious code triggered under specific circumstances. More formally, let $\lambda = (c, T_c, \Phi_c)$ be an SHSO, S its sensitive behavior, and M a piece of malicious code such as $M \subset \Sigma$. Then, λ is a logic bomb if $M \subseteq S$ (cf. Figure 1c). In other words, a logic bomb is an SHSO which suspicious sensitive behaviour is malicious.

```
1  // Example simplified for reading, with renamed methods
2  public static void m() {
3      m1();
4      performMaliciousActivity();
5  }
6  public static void m1() {
7      if (m2()){
8          System.exit(0);
9      }
10 }
11 public static boolean m2() {
12     String str1 = Build.MODEL;
13     String str2 = encryptedString(); // str2 = "Emulator"
14     return str1.contains(str2);
15 }
16 public static String encryptedString() {
17     String s1 = "cb6624dec24f889f4fcdf6c8cda99d4a";
18     return BYDecoder.decode(s1, "0ec47edd8db3a02b");
19 }
```

Listing 1: Logic bomb identified by Difuzer in "com.flyingbees.BrasilTvEnvivo" with emulator evasion.

In Listing 1, we summarize the general behavior of a concrete example of a logic bomb extracted from a real-world app. This logic bomb was detected by Difuzer. In this example, the different parts of the SHSO (including the triggering condition checks) are split across several methods ($m1, m2, m$). The actual triggering condition check is done in line 3: $m2$ will return `true` if the device runs in an emulator and the app execution will be halted. Otherwise, the malicious behavior (line 2) will be triggered.

The challenge in detecting the aforementioned logic bomb is that analysts cannot rely on rules or models to detect it due to the lack of a formal definition of malicious behavior. Therefore, we note that, with little coding effort, malware authors could push malicious code that will be missed in most dynamic analyses. Indeed, sandboxes and testing environments usually return default values for environment variables [60]. Besides the device's model, different environment values (e.g., sensors, settings, GPS, remote values, etc.) can be used to trigger malicious code.

Comparing to previous works, we note that the presented simple example of logic bomb detected by Difuzer would constitute a challenge to the existing state of the art. TriggerScope [26] cannot identify this logic bomb. Indeed, since its heuristics are limited to time-, location-, and SMS-related triggers, logic bombs with a new trigger (e.g., environment variable such as `Build` class fields) are missed. HsoMiner [57] could detect this logic bomb if its training set includes similar examples. Unfortunately, HsoMiner flags too many HSOs (e.g., ~20% of apps), making the manual check a cumbersome task. In contrast, Difuzer offers a reasonable number of warnings to be checked manually.

3 APPROACH

Goal: With Difuzer, we do not aim at detecting any HSOs, but only suspicious HSOs (SHSOs) for which the likelihood of being logic bombs is high.

Intuition: As shown in previous studies [57], the number of HSOs per app can be large, even in benign apps. This suggests that although HSOs are "sensitive" operations, most of them are legitimate, i.e., they are used to implement common behavior. In contrast, logic bombs are rare, especially in benign apps. The idea behind Difuzer is to use an anomaly detection approach, with specifically designed features, to triage logic bombs among SHSOs.

Overview: In Figure 2, we present an overview of our approach, which consists of two main modules: (1) identification of SHSO entry-point candidates via control flow analysis, instrumentation, and taint tracking (left dotted block); (2) From these entry-points, triggers are extracted, and the second module (right dotted block) extracts specifically designed features fed into an outliers predictor. This predictor is previously trained on a set of reference apps (i.e., apps considered benign) to learn legitimate usages of triggers.

3.1 Identifying SHSO candidate entry-points

Previous works [3, 19, 55, 60, 73] have shown that specific values, such as system inputs and environments variables, are often used to trigger HSOs. State-of-the-art approaches have thus proposed to check whether the conditions of *if statements* contain these sensitive data. To that end, they rely on symbolic execution [26] or backward data-dependency graphs [57] that could suffer from scalability problems. With Difuzer, we propose to use taint analysis to track sensitive data values and check if they are involved in conditional expressions.

[1] Note that in case there is no false branch, $\Phi_c = \emptyset$.

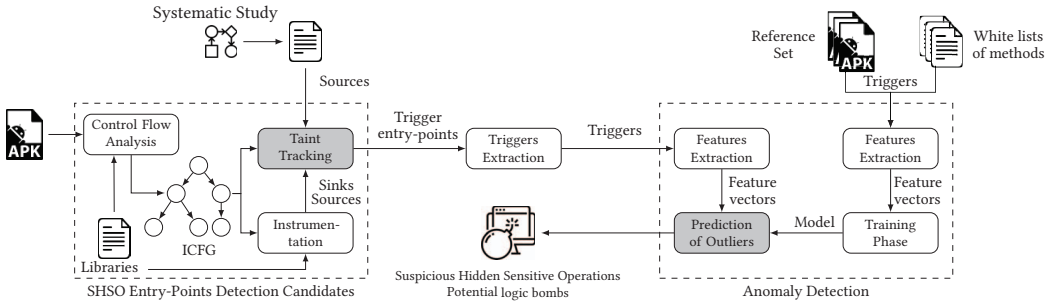

Figure 2: Overview of the DIFUZER approach.

Taint analysis tools generally track data from sources to sinks. The implementation of FLOWDROID, a popular taint analysis framework for tracking sensitive information, considers sources and sinks at the method level. In our case however, sinks are fine-grained code locations, which are conditional expressions of *if statements*. This requires for DIFUZER to instrument apps in order to insert dummy method calls that will make the apps ready for analysis by FLOWDROID (cf. Section 3.1.2). Moreover, sources can be method calls or data field accesses. To build the set of source and sinks we propose to make a systematic mapping (cf. Section 3.1.1) that explores internal and external system properties and their associated APIs as well as environment variables.

3.1.1 Systematic mapping toward defining sources. As already explained, a first step is to track sensitive values. In this work, these values are derived from particular source methods. Then, if a sensitive value falls into an *if statement*, we consider the condition as a potential SHSO entry-point. This section will describe how we gathered a comprehensive list of source methods used for the taint tracking phase. Note that we did not rely on the reference sources list produced by SUSI [5] since it has been shown that most of the methods are inappropriate for tracking sensitive data, and lead to a high amount of false-positives (e.g., >80%) [36, 51, 56].

In general, decisions on whether to trigger SHSOs or not are taken on system properties [19, 57, 71, 73]. Hence, we performed a systematic mapping of the Android framework from SDK version 3 to 30 (versions 1 and 2 were unavailable) to gather a comprehensive list of source methods. In particular, since in the case of Android apps, system properties can be derived from the device's *internal* and *external* properties, we inspect the successive versions of the framework to identify various means to access these properties.

	Device					
	Internal			**External**		
	System	**Content**	**Build**	**SIM**	**Internet**	**GPS**
Examples	Sensors, Camera	Call Logs, Contacts	Model, Hardware	Phone call, SMS	Parameters, Content	Latitude, Longitude

Table 1: Examples of sensitive sources

In Table 1, we enumerate the different property types (with examples) on which we reasoned to retrieve sensitive sources, which are classically focused on in the literature [19, 57, 71, 73]. We follow a systematic process to perform the retrieval of sources from the given property types: we first extracted patterns from the different ways to access the aforementioned properties. Then, we used those patterns to automatically discover the sensitive sources that we make available to the research community in the DIFUZER project's repository. In the following, we further detail the internal and external properties that we consider.

Internal: In the case of internal properties, a developer can get sensitive information of the device from three main channels: 1) System properties, 2) Content in internal databases, and 3) Information from `BUILD` class (see Table 1). In the following, we describe how we obtain a list of sources for those three channels:

❶ *System properties*: While developing an Android app, developers have access to several useful APIs. In this case, the most interesting is `android.content.Context.getSystemService(java.lang.String)` [30] which returns the system-level handler for a given service. The service is described by a string given as parameter to `getSystemService` method. The `Context` class gives developers access to pre-defined constants (e.g., `SENSOR_SERVICE`).

In fact, every constant contains the name of the service with `"_SERVICE"` appended to it. The return value type of the `getSystemService` method call is derived from the constant name (e.g., `SENSOR_SERVICE` will give a `SensorManager` [33]) which in turn can be used to get a object whose type is also derived from the constant name (e.g., a SensorManager object can be used to obtain a Sensor object [32]). We used this pattern to compile our list of sensitive sources for the System properties. More specifically, we verify if the class exists in at least one SDK version for each class obtained. If this is the case, we list the methods of the class and keep only the "getter methods", i.e., those starting by "get" or "is" (e.g., methods such as `getId()` or `isWifiEnabled()`).

❷ *Content in internal databases*: To access databases fields, one has to perform a query which returns a `android.database.Cursor` [31] object. This object is then used to iterate over the result of the query. Hence, to get sensitive source methods related to content in internal databases, we applied the same process as for system properties (i.e., to retrieve the "getter" methods) but on the `Cursor` class.

❸ `Build` *class*: The `Build` class [29] allows developers to access information about the current build of the device from its fields. For instance, one can get the brand associated with the device by accessing `Build.BRAND`. Note that our objective is to retrieve a list of source methods. However, the information a developer can get from the `Build` class can only be retrieved from class fields, not method calls. Consequently, in Section 3.1.2, we will explain how we instrument the app under analysis to add method call statements representing `Build` field accesses.

We gathered a list of 618 unique methods for internal values.

External: In the case of external properties, a developer can get sensitive information from three channels: 1) SIM card, 2) Internet Connection, and 3) GPS chip. The process to collect the source methods is similar to the one followed with `Cursor` class, except we do not know in advance the name of the classes to inspect. Therefore we relied on a heuristic to identify such classes: for each SDK version, we listed all the classes and kept only those with class names containing the following words: "Sms, Telephony, Location, Gps, Internet, and Http". Once the classes were retrieved, we listed the methods for each class and kept those starting by "get" or "is". The intuition is the same as in the case of internal sources.

We gathered a list of 794 unique methods for external values. Finally, after combining sensitive sources from internal and external values, our list contains 1285 unique methods (127 duplicates).

3.1.2 Instrumentation. Performing taint tracking, as briefly described in Section 2, consists of a data-flow algorithm that propagates the taint from a source method to a sink method.

Sinks related challenge: We remind that one objective of Difuzer is to identify SHSOs' trigger entry-points. Consequently, the taints that Difuzer tracks are supposed to fall into *if statements*. However, being not a method call, an *if statement* cannot be considered as a sink when using state-of-the-art static taint analyzers [6, 28, 79]. A concrete example of what Difuzer tracks is given in Listing 2. On line 7, *countryCode* variable is tainted from *getNetworkCountryIso()* source. This value is then used (line 9) to perform a test and trigger malicious activity (line 9). As an *if statement* is not considered a sink, a flow cannot be found.

```
1   public void method() {
2     String b = Build.BRAND;
3   + b = BuildClass.getBRAND(); // dummy method call for field access
4     String p = Context.TELEPHONY_SERVICE;
5     Object o = this.getSystemService(p);
6     TelephonyManager tm = (TelephonyManager) o;
7     String countryCode = tm.getNetworkCountryIso();
8   + IfClass.ifMethod(countryCode, "RU"); // dummy method call for if statement
9     if(countryCode.equals("RU")){ performMaliciousActivity(); }
10  }
```

Listing 2: Example of app instrumentation performed by Difuzer (Lines with "+" represent added lines).

Our approach overcomes this limitation by instrumenting apps. To accomplish this, the app code is first transformed into Jimple [76], the internal representation of Soot [75]. Then, Difuzer iterates over every condition of the app, and for each condition, Difuzer inserts a dummy method `ifMethod` with the variables involved in the condition as parameters. This `ifMethod()` is static and declared in a dummy class `IfClass` that contains all instrumented methods related to conditions. See line 8 in Listing 2.

Once the instrumentation is over, we dynamically register every newly generated method calls as sinks to FlowDroid.

Sources related challenge: As described in Section 3.1.1, we consider, in this study, `Build` class' fields as sources. Since field accesses are not method calls, we follow the same process as for *if statements* to insert dummy methods. More specifically, Difuzer generates a static method call on-the-fly representing a field access from the `Build` class. Listing 2 depicts an example of this instrumentation process, where the dummy method `getBRAND()` of the dummy class `BuildClass` is inserted in line 3. Furthermore, newly generated method calls are registered as sources for taint tracking.

3.2 Anomaly detection

This section presents Difuzer's second module, which relies on anomaly detection. In particular, we detail the unsupervised machine learning technique used to detect abnormal triggers.

3.2.1 Why a One-Class SVM?. A classical classification problem requires samples from positive and negative classes to build a model, which is then used to assign labels to test instances [39]. This induces possessing a reasonable amount of samples from two classes, which is not the case in our study. Indeed, the SHSO detection problem is challenging, and to the best of our knowledge, there is no ground truth made publicly available. Thus, using supervised learning in our study is not practical and present limited feasibility.

Therefore, we decided to rely on an unsupervised learning technique to detect SHSOs, particularly on a One-Class Support Vector Machine (OC-SVM) machine learning technique. An SVM algorithm was chosen due to its ability to generalize [78] and its resistance to over-fitting [80]. The general idea of OC-SVM is to identify the smallest hyper-sphere to include most of the samples of the positive samples [84]. A sample considered as an outlier by the model means the data-point is not in the hyper-sphere.

3.2.2 Features extraction. As already said, the second Difuzer module's objective is to detect abnormal triggers with the intuition that these triggers are HSOs for which the likelihood to be a logic bomb is high, namely SHSOs. This module implements an OC-SVM algorithm which takes as input feature vectors computed from the triggers previously extracted from the entry-points yielded by the first module of Difuzer (cf. Figure 2).

To engineer anomaly detection features, we reviewed surveys [47, 89] and related-papers [2, 57, 62, 87] discussing Android malware and investigated the techniques used by malware writers to hide malicious code within apps. Eventually, we identified nine unique trigger/behavior features that are described in the following.

In the remainder of this section, we consider a trigger $\tau = (c, T_c, \Phi_c)$ and its guarded code $\Gamma = T_c \cup \Phi_c$ (cf. Section 2).

Difuzer builds a feature vector $v =< S, N, D, R, B, P, M_1, S_1, J >$ for a given trigger where:

S: Number of sensitive methods used in guarded code. Intuitively, this feature represents how much a trigger controls the execution of sensitive methods. Indeed, while HSOs guard the execution of sensitive operations for performing sensitive activities [25], benign triggers, in the general case, perform benign activities, i.e., invoke few sensitive methods, or not at all. To retrieve this value, Difuzer iterates over every statement of Γ and recursively checks whether a sensitive method is called or not. For this purpose, we gathered a list of sensitive APIs constructed in previous work [7].

N: Is native code used in guarded code? Since analyzing native code is more challenging than Java bytecode [46], Android malware developers tend to hide malicious code from automated analyses in native code [2, 62]. Hence, this feature is a boolean value that, when set to 1, means native code is used in Γ, 0 otherwise.

D: Is dynamic loading used in guarded code? Dynamic class loading is not exclusively used in malware. However, as malware is becoming increasingly sophisticated, they use built-in capabilities like dynamic loading to hide from automated analyses [87]. Consequently, likewise native code, this feature is a boolean value set to 1 if dynamic loading is used in Γ, 0 otherwise.

R: Is reflection used in guarded code? Android malware writers tend to use more and more reflection-based code [87] since most of the state-of-the-art techniques overlook this property due to the challenging task of resolving it. Therefore, this feature is set to 1 if reflection is used in Γ, 0 otherwise.
B: Does guarded code trigger background tasks? Android apps rely on the Service component to run background tasks. Hence, with this feature, we aim at capturing the fact that the app under analysis performs stealthy operations without user knowledge. The intuition here is that SHSOs' role is to hide code both from security analysts and end-users (e.g., in the case of a logic bomb). This feature is set to 1 if background services are triggered in Γ, 0 otherwise.
P: Are parameters of condition used in guarded code? This feature captures the dependency of a condition to its guarded code. The hypothesis is that, in the case of SHSOs, the guarded code does not use values used in the condition since they represent different behaviors. To achieve this, Difuzer performs a def-use analysis of the guarded code to verify if any variable used in the condition is used before being assigned a new value. If this is the case, the feature is set to 1, 0 otherwise.
M_1: Number of app methods called only in guarded code. With this attribute, we attempt to uncover the number of methods defined in the app called only in the guarded code of a trigger. The rationale is that app methods that are only used under a specific circumstance are likely to be defined only for this specific circumstance, representing hidden behavior [26]. To retrieve this number, Difuzer queries the call-graph (built using SPARK [40] algorithm) for each method call in the guarded code to verify if it has only one incoming edge (i.e., it is only called within the current method).
S_1: Number of sensitive methods called only in guarded code. In the same way as M_1, we aim to capture the number of sensitive methods only used in the guarded code of a given trigger.
J: Behavior difference between branches. Intuitively, two branches of an SHSO should be noticeably different. Indeed, of the two branches, one is considered the normal behavior (no or few sensitive operations) if the condition is not satisfied and the other as the sensitive behavior (sensitive operations) if the condition is satisfied [57]. Therefore, to compute this difference, Difuzer first inter-procedurally retrieves sensitive method calls in both branches of a given trigger. Let X_{T_c} and X_{Φ_c} respectively be the sets of sensitive methods in the true and the false branch of a trigger. Therefore, to compute this difference of the two branches, Difuzer relies on the Jaccard distance: $D_j(X_{T_c}, X_{\Phi_c}) = 1 - \frac{|X_{T_c} \cap X_{\Phi_c}|}{|X_{T_c} \cup X_{\Phi_c}|}$, which characterizes the behavior difference of the two branches. A value close to 1 means that both branches are dissimilar.

3.2.3 Training phase. To train our OC-SVM model, we need samples of a positive set, i.e., triggers considered normal. Therefore, we randomly chose 10 000 goodware (i.e., VirusToal [74] score = 0) from AndroZoo [4]. Then, for each of these apps, we applied Difuzer to extract a feature vector for each app's condition.

Afterward, we randomly chose 10 000 feature vectors[2] from those yielded by Difuzer, which we labeled as positive (i.e., part of the normal behavior). We then trained our One-Class Classification based anomaly detector, leveraging LibSVM [14]. To ensure that the selected training set does not bias the trained model's performance, we split it and compute Accuracy in 10-fold cross-validation. Overall, we achieve a stable Accuracy of 99.91% on average.

[2]The number of extracted vectors is orders of magnitude higher. However, for efficiency, we validated that a random set of 10 000 vectors yields an acceptable performance.

4 EVALUATION

To evaluate Difuzer, we address the following research questions:
RQ1: What is the performance of Difuzer for detecting Suspicious Hidden Sensitive Operations (SHSOs) in Android apps?
RQ2: Can Difuzer be used to detect logic bombs? We address this question by considering three sub-questions:

- **RQ2.a:** Are SHSOs detected by Difuzer likely logic bombs?
- **RQ2.b:** How does Difuzer compare against TriggerScope, a state of the art logic bomb detector?
- **RQ2.c:** From a qualitative point of view, does Difuzer lead to the detection of non-trivial triggers/logic bombs?

RQ3: Can SHSO detection in goodware reveal suspicious behavior?

4.1 RQ1: Suspicious Hidden Sensitive Operations in the wild

In this section, we assess the efficiency of Difuzer to find SHSOs on a dataset of malicious applications.
Dataset. To the best of our knowledge, there is no SHSO ground-truth available in the literature. Consequently, in this study, we considered 10 000 malicious Android apps as our malicious dataset. These apps were released in 2020, collected from the AndroZoo [4] repository, and have been flagged as malware by at least five antivirus scanners in VirusTotal.

We contacted the authors of state of the art approaches (e.g., HsoMiner [57], and TriggerScope [26]) to get their artifacts (datasets and tools) for comparative assessment. Unfortunately, no artifact was made available to us.
Libraries. It has been shown in the literature [15, 45] that library code can affect analyses performed over Android apps since it often accounts for a larger part than the app's core code. Consequently, in this study, we considered two cases: (1) with-lib analysis (i.e., we consider the entire app code including library code); (2) without-lib analysis (i.e., we consider only developer code). To rule out libraries, we rely on the state-of-the-art list available in [45].
Post-Filter. As a precaution, before analyzing the results without libs, we listed the classes in which Difuzer found potential sensitive triggers to search for redundant classes that could indicate libraries. We were able to filter out 19 additional libraries that were not listed in the list we used and provided by [45].

In the following, when referring to the analysis without libraries, we consider the 19 libraries previously presented as well as the libraries of the list in [45] as filtered. It accounts for a total of 5982 library classes and packages filtered.

4.1.1 Efficiency of Detecting SHSOs. We recall that Difuzer is targeted at detecting SHSOs. While in RQ2 we investigate the likelihood for these SHSOs to be logic bombs, we first investigate the efficiency (with RQ1) of Difuzer in the detection of SHSOs. We further perform an ablation study to highlight the performance of the anomaly detection module.

In Table 2, we report the results of applying Difuzer (with the anomaly detection step activated) on our 10 000 malware dataset. When analyzing the entire apps, Difuzer detects at least one SHSO in 339 apps (3.39%). Overall, Difuzer detects 5575 SHSOs in these 339 apps leading to an average number of 16.4 SHSOs per app. In comparison, when only the app developers' code is considered, Difuzer detects at least one SHSO in 259 apps (2.59%), with a total number of 2435 SHSOs detected and an average number of 8.2 SHSOs per app. We note that the 3437 (5575-2435) SHSOs that are not in the app developer code, are actually detected in 68 libraries suggesting that only a few libraries contain SHSOs . Figure 3 further details the distribution of detected SHSOs per apps.

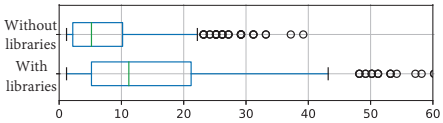

Figure 3: Distribution of the number of SHSO(s) per app in analyses with and without libraries (only apps with at least one SHSO are considered).

These first results show that SHSOs indeed exist in malicious apps, but in relatively low number (in around 3% of the apps). However, when SHSOs are present in an app, they are not rare (on average, about 8 SHSOs per app in the developer code). Finally, SHSOs are more prevalent in library code than in app developer code, but only a few libraries contain SHSOs.

Table 2 also reports the average numbers of triggers before and after applying the anomaly detection step (i.e., the second module of Difuzer). Interestingly, we can see that this anomaly detection drastically reduces the number of triggers that are considered as SHSOs. Indeed, when considering the 10 000 apps, there are on average $174336/10000 \approx 17.43$ and $146018/10000 \approx 14.60$ triggers per apps (with or without libraries respectively) generated by the first module of Difuzer, i.e., by the taint analysis step. After the anomaly detection step, these numbers drop to $5575/10000 \approx 0.56$ and $2435/10000 \approx 0.24$ respectively, corresponding to a decrease of 96% and 98% respectively.

These results show that the anomaly detection step has a significant impact on the number of detected SHSOs by significantly reducing the search space of triggers by up to 98%. This search space reduction is key when the ultimate goal is to detect malicious code and to support security analysts manual inspection (cf. Section 4.2).

Table 2: Results of the experiments executed on 10 000 malware with and without taking into account libraries.

	Analysis with libs	Analysis without libs
Number of apps with SHSO(s)	339	259
Number of SHSOs	5575	2435
Number of SHSOs/app	16.4	8.2
Average # triggers (i.e., before Anomaly detection)	17.43	14.60
Average # SHSOs (i.e., after Anomaly detection)	0.56	0.24
Mean analysis time	35.63 s	33.54 s

We further inspect the SHSOs detected by Difuzer by focusing on the app developer code only (we do not consider library code). Table 3 lists the top 10 types of trigger that Difuzer was able to discover. The second column gives some examples of methods considered sources for the taint tracking to uncover SHSO entry-points. We note the diversity of types of triggers that developers use. For instance, a developer can decide to trigger (or not) the sensitive code if: (Database trigger type) specific values are present in databases (e.g., contacts, messages); (Internet trigger type) external orders say so; (Build, Telephony, and Camera trigger types) the device is not an emulator; (Connectivity, and Wi-Fi trigger types) the device has Internet access; (Location rigger type) the user is in a pre-defined location; Note that the methods in Row 3 have been dynamically generated by Difuzer during instrumentation to track the Build class's field values.

Table 3: Top ten trigger types discovered by Difuzer in the developer code. (T. = Triggers)

Trigger Type	Examples of methods	# T.	Trigger Type	Examples of methods	# T.
Database	getString, getInt, getCount	785	Location	getLastKnownLocation, getLongitude	84
Internet	getResponseCode, getResponseMessage	715	Wi-Fi	isWifiEnabled, getConnectionInfo	76
Build	getMODEL, getMANUFACTURER	374	Power	isScreenOn, isInteractive	47
Telephony	getDeviceId, getNetworkOperatorName	97	Audio	getStreamVolume, isMusicActive	37
Connectivity	getActiveNetworkInfo, getNetworkInfo	88	Camera	getCameraIdList	28

Regarding the component types in which Difuzer found SHSOs, 90% of SHSOs are in methods of "normal" classes, i.e., not Android components. SHSOs are found in `Activities` in 9% of the cases. However, they are rarely found in `Services` and `Broadcast Receivers` (less than 1%).

4.1.2 Manual Analyses. Since static analysis approaches often suffer from false alarm issues, i.e., they report a large proportion of false-positive results, we decided to verify the detection capabilities of Difuzer manually. To that end, the authors of this paper randomly selected a statistically significant sample of 102 apps out of the 259 apps in which SHSOs exist in developer code, with a confidence level of 99% and a confidence interval of ± 10%. Only one sample was found to be a false-positive result. Indeed this app verifies if it is running in an emulator by comparing `Build.PRODUCT`, `Build.MODEL`, `Build.MANUFACTURER`, and `Build.HARDWARE` against well-known strings such as "generic", "Emulator", "google_sdk", etc. This test seems sensitive, but the guarded code displays the following message to the user: "Scooper Warning: App is running on emulator.". Therefore, Difuzer achieves a precision of 99.02 % to find *Suspicious Hidden Sensitive Operations* on this dataset. We release the annotated list of 102 apps that were manually checked for transparency in the project's repository.

4.1.3 Analysis Time. The last row in Table 2 reports Difuzer analysis time. Difuzer outperforms state-of-the-art trigger detectors with an average of 33.54 s per app (35.63 s for the analysis with libraries, with an average DEX size of 7.03 MB per app), making Difuzer suitable for large-scale analyses. In comparison, state-of-the-art tools such as TriggerScope [25] and HsoMiner [57]) require 219.21 s and 765.3 s per app respectively. Note that 85.42% (i.e., 28.65 seconds on average) of this time is reserved for the taint analysis. Also, 24 apps (0.24%) reached the timeout (i.e., 1 hour) before the end of the analysis.

RQ1 answer: Difuzer detects SHSOs in Android malware with high precision, i.e., 99.02 % in less than 35 seconds on average. Among the average 14.6 HSOs identified in an app based on triggers spotted by static taint analysis, only 2% are suspicious according to anomaly detection, which shows that Difuzer is effective in reducing the search space for manual analysis.

4.2 RQ2: Can Difuzer detect logic bombs?

In this section, we ❶ evaluate Difuzer's efficiency in detecting logic bombs (RQ2.a), ❷ compare it against TriggerScope (RQ2.b), and ❸ discuss logic bomb use cases in real-world apps (RQ2.c).

4.2.1 RQ2.a: Are SHSOs detected likely to be logic bombs?

Until now, we have shown that Difuzer is effective in detecting SHSOs. From a security perspective, however, we must further show that these SHSOs are actually malicious. In other words, are these SHSOs likely to be logic bombs. Unfortunately, such assessment is challenged by the lack of ground truth in the literature. We therefore require extra manual analysis effort of reported results.

Initial Manual Analysis: In previous Section 4.1.2, we present our manual analysis of SHSOs detected in 102 apps. During this analysis, we further checked if the detected SHSOs contain malicious code. In particular, for each app under analysis, we gathered information about the reason it was flagged by antiviruses (e.g., on VirusTotal). Then, in the guarded code of the potential SHSO found by Difuzer, we looked for malicious behavior matching our information previously gathered. For instance, if: (1) an app is labeled as being a trojan stealing the device's information; (2) the potential SHSO is performing emulator detection (e.g., calling `System.exit()` method if the device is running in an emulator); and (3) the behavior exhibited in the code guarded by the condition detected by Difuzer is gathering the device's information (e.g., unique identifier, current location, etc.) and sending it outside the device, the SHSO is considered a logic bomb.

Eventually, 30 apps (i.e., 29.7%) were manually confirmed to be logic bombs, i.e., the SHSOs were triggering malicious code.

Semi-Automated further Analysis: Manual investigation is time-consuming. This is the reason why we inspected 102 apps and not all 259 apps reported to having at least one SHSOs within the developer code parts. To quickly enlarge the set of identified logic bombs, we decided to follow a simple but efficient process. It is known that malicious developers often reuse the same piece of code in different apps [47]. Therefore, for each already identified logic bomb, we search for similarities (i.e., SHSOs found in the same class name, same method name, and the same type of trigger used) in SHSOs contained in the 157 (259 − 102) remaining apps. Our analysis yielded 16 additional apps containing logic bombs that were manually verified and confirmed. Eventually, our logic bomb dataset, called DataBomb, contains 46 Android apps, each with an identified logic bomb. We believe this dataset to be useful to the community to further improve logic bomb detection in Android apps. We made it publicly available in the project's repository.

Discussion about HSO, SHSO and Logic Bomb: In the literature [26, 57], HSO is consistently defined as a sensitive operation that is hidden by specific triggering conditions. Nevertheless, the notion of "sensitive operation" is not clearly delineated, which challenges comparison across approaches. In our work, we postulate that while detecting HSOs is an important first step, it is not enough to help security analysts. Indeed, as shown by our manual analysis, a large proportion of HSOs are indeed sensitive but not necessarily suspicious. As a result, most of the detected HSOs are legitimate and do not require any inspection effort from security analysts.

In this context, if the goal is to detect real security issues and reduce the burden of security analysts, a tool such as HsoMiner [57] which detects *HSOs* in 18.7% of apps within a set of over 300 000 apps (including malicious and benign apps) appears to be unpractical. In contrast, Difuzer detects *suspicious HSOs* in 3.39% of the analyzed apps (when libraries are considered), and our manual analyses confirm that in about 30% of the apps, these SHSOs are logic bombs, making the work of security analysts easier. Though both HsoMiner dataset and our dataset are different (we were not able to get the HsoMiner's authors dataset), if we compare the 18.7% of apps with HSOs reported by HsoMiner, with the 3.39% reported by Difuzer, we can say that Difuzer reduces the search space by up to 81.9% ($(18.7 - 3.39) \times \frac{100}{18.7} = 81.9$) to accelerate the identification of logic bombs.

RQ2.a answer: By triaging HSOs to focus on suspicious ones based on anomaly detection, Difuzer was able to reveal 30 logic bomb instances in a sampled subset of malware apps having SHSOs. Besides, we release DataBomb, an annotated dataset of 46 Android apps confirmed to be using logic bombs.

4.2.2 RQ2.b: How does Difuzer compare against TriggerScope, a state of the art logic bomb detector?

In the absence of a public ground-truth for Android logic bomb instances, we perform experimental comparisons against the TriggerScope state-of-the-art detector in the literature that relies on static analysis. Although TriggerScope is not publicly available, we are able to build on a replication based on technical details provided in TriggerScope paper [26].

Overall, our approach differs from TriggerScope's by three major differences: ❶ **Technique**: TriggerScope uses symbolic execution to tag variables with a limited number of values, we use static data flow analysis; ❷ **Target**: TriggerScope detects hidden sensitive operations (i.e., whether at least one sensitive method is called within the guarded code of a trigger), whereas Difuzer 's goal is to detect suspicious hidden sensitive operations (i.e., the guarded code is sensitive and implements an abnormal behavior); and ❸ **Approach**: TriggerScope maintains a list of sensitive methods and uses the occurrence of any of them as the sole criterion, Difuzer implements an anomaly detection scheme where the presence of sensitive methods is one feature among many others. While TriggerScope and Difuzer both rely on list of sources to find triggers of interest, TriggerScope handpicks a limited set of methods, whereas Difuzer 's list is based on a systematic mapping (cf. Section 3.1.1 - we leverage patterns to systematically search for sources).

Does TriggerScope identify as logic bombs the SHSOs flagged by Difuzer?

We applied TriggerScope on the subset of 102 apps where Difuzer identified a SHSO (cf. Section 4.2.1). The objective is to check whether TriggerScope is more or less accurate than Difuzer. Typically, among the 30 logic bombs that have been manually verified as true positives, how many are detected by TriggerScope. Similarly, does TriggerScope detect logic bombs (manually verified as true positives) that Difuzer could not. Figure 4 illustrates the differences in logic bomb detection (left figure). Overall:

- TriggerScope did not flag any logic bomb that Difuzer did not.
- TriggerScope could only detect 2 logic bombs among the 30 logic bombs that Difuzer correctly identified.

- As reported in the literature [67], TriggerScope exhibits a very high false positive rate at 94.6%: 35 among its 37 detections are false positives (the rate for Difuzer is 70.6%, 72/102).

Does Difuzer fail to flag as SHSOs the logic bombs detected by TriggerScope?

We recall that, contrary to Difuzer, which builds on anomaly detection, TriggerScope is restricted to detect only logic bombs where the trigger involves location-, time-, and SMS-related properties. Aligning with the assessment of Difuzer, we applied TriggerScope on our set of 10 000 malware. TriggerScope reported 591 logic bombs in 149 apps (~4/app): 98.6% of the reported cases are time-related. In the absence of ground truth, we again propose to manually verify a random sample set of reported logic bombs. To facilitate comparison with Difuzer, we sample 102 apps (we simply considered the same number of apps as in the previous question), and manually confirmed that for 97 (95.1%) apps, the reported logic bombs are false positives. In 5 (4.9%) apps, we found at least one reported logic bomb to be a true positive.

We further check whether on these 102 apps where TriggerScope reported a logic bomb, Difuzer also flags any case of SHSO: Difuzer flagged 68 apps as containing SHSOs, among which 7 are manually confirmed to be logic bombs. The details of the comparison between TriggerScope and Difuzer are presented in the Venn Diagram in Figure 4 (right figure). We note that:

- 2 logic bombs are detected by both Difuzer and TriggerScope.
- 5 SHSOs detected by Difuzer are actual logic bombs, but not detected by TriggerScope. Indeed, TriggerScope is limited by its focus on time, location and SMS-related triggers.
- 3 logic bombs are detected by TriggerScope, but not detected by Difuzer. Our prototype implementation considers a limited list of sources, which do not cover those 3 logic bomb cases.

Although we do not have a complete ground truth (with information about all cases of logic bombs), confirming and comparing detection reports by Difuzer and TriggerScope offers an alternative to assess to what extent each may be missing some logic bombs. The results described above suggest that Difuzer suffers significantly less from false-negative results than TriggerScope.

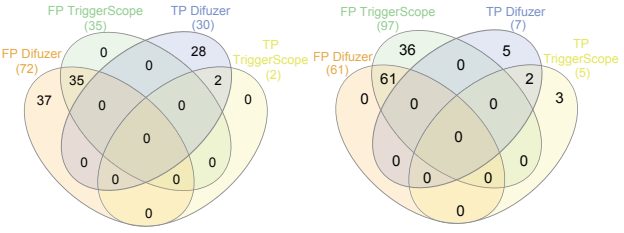

Figure 4: Venn Diagram representing results of TriggerScope and Difuzer on 102 apps originally detected by Difuzer on the left, and TriggerScope on the right. (FP = False Positive, TP = True Positive)

RQ2.b answer: Overall, Difuzer outperforms TriggerScope by detecting more logic bombs more accurately (wrt. false positives), and by missing less logic bombs (wrt. false negatives).

4.2.3 RQ2.c: From a qualitative point of view, does Difuzer lead to the detection of non-trivial triggers/logic bombs?

In this section, we discuss two real-world apps in which Difuzer revealed logic bombs that cannot be detected by TriggerScope.

Advertisement Triggering. Difuzer revealed an interesting logic bomb in "com.walkthrough.knife.assassin.hunter.baoer" app which is an adware app of the HiddenAd family. The app uses the `android.app.job.JobService` class of the Android framework to schedule the execution of jobs (the developer can handle the code of the job in `onStartJob` method). In the `onStartJob` method, the app takes advantage of the `PowerManager` of the Android framework to check if the device is in an interactive state (i.e., the user is probably using the device) with method `isScreenOn()`. If this is the case, the app displays advertisements to the user and schedules the same class's execution after a certain time.

Data Stealer. Logic bombs can also be used to trigger data theft under the condition that the data is available. For instance, in app "com.magic.clmanager", which is a Trojan (hidden behind a cleaning app) capable of stealing data on the device, Difuzer found a logic bomb related to the device unique identifier. Indeed, in method `d(Context c)` of the class `c.gdf`, a check is performed against the value returned by method `getDeviceId()` to verify if the value matches specific values (emulator detection) in a given file named "invalid-imei.idx". In the case the app considers that the device is not an emulator, it triggers the stealing of sensitive information about the device such as the current location, phone number, information on the camera, information about the Bluetooth, disk space left, whether the device is rooted or not, the current country, the brand, the model, information about the Wi-Fi, etc. Afterward, this information is written in a file and sent to a native method for further processing.

4.3 RQ3: SHSOs in benign apps

Until now, we have focused on malware. However, SHSOs are not exclusively found in malicious apps [57]. Therefore, in this section, we intend to conduct a study on benign applications.

Results. As confirmed in Section 4.1.1 and in previous studies [26, 57], benign libraries and benign Android apps implement HSOs. Our study confirms this finding. Even more, 354 benign apps (3.54%) were flagged by Difuzer to contain suspicious HSOs. We further manually analyzed 20 apps randomly selected from our results and confirmed that they all contain at least one SHSO. Table 4 shows the different trigger types used in benign apps to trigger SHSOs. A significant result here is that benign apps use considerably less the "Build" trigger type (see Table 3 for comparison) than malicious apps. Similarly, the "Telephony" trigger type is less used in benign apps than in malicious apps. This induces that, in benign apps, decisions are less taken depending on values derived from methods like: `getDeviceId()`, `getNetworkOperatorName()`, `getPhoneType()`, `getMODEL()`, `getMANUFACTURER()`, or `getFINGERPRINT()`. A hypothesis would be that benign apps are less prone to recognize an emulator environment (and use this information to set triggering conditions).

Besides, we can see in Figure 5 that, in comparison with malicious apps, benign apps tend to have significantly fewer triggers per app.

Table 4: Top ten trigger types used by benign Android apps.

Database	Internet	Location	Connectivity	Audio	Telephony	Wi-Fi	View	Activity	Build
897	283	264	74	63	58	25	21	19	19

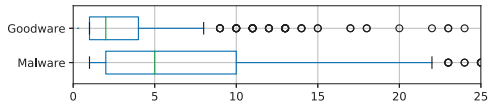

Figure 5: Distribution of the number of SHSO(s)/app in goodware and malware (apps with at least 1 SHSO are considered).

4.3.1 Case Study. This section presents an SHSO of a benign app.
Benign App. The app we consider in this case study is "no.apps.dnbnor". DIFUZER detected an SHSO in method <bom.∮: java.lang.String ···()> which tests if the value of `Build.CPU_ABI` or `Build.CPU_ABI2` is equal to pre-defined values stored in a file. In the case a match is found, it triggers the copy of a native code file into a second file. The native code file name is in the form: "lib/" + `str` + "/lib" + `f9` + ".so". The `str` variable represents a `CPU_ABI` value and the `f9` variable represents a string to designate the file. This file is then opened and eventually copied in the user data directory of the running app.

Although not malicious in this case, this behavior is suspicious, and DIFUZER was able to reveal it.

4.3.2 Malicious activities in Google Play. We now illustrate how DIFUZER contributed to removing 8 apps whose behavior was potentially harmful to users (in the form of aggressive, unsolicited, and intrusive ads) in Google Play. Developers of such *adware apps* managed to evade classical checks performed in Google Play.

During our manual analyses of benign apps, we stumbled upon an app with an SHSO flagged by DIFUZER. Our inspection of the code suggested that the SHSO is not a logic bomb per se since it does not trigger the malicious code. However, during this manual analysis, we noticed that the app was apparently mainly designed to display advertising content aggressively. To confirm our hypothesis, we downloaded the sample and executed it in an emulator. First, we noticed poor app design, poor quality, and low content. Then in nearly every screen (i.e., `Activity` component), we received embedded ads and full-screen ads. This behavior is characteristic of adware apps. After verification, we found that the app was still in Google Play with a relatively high number of downloads (a few thousands) but with negative comments. In fact, the app pretended to provide users with a "walkthrough" version of an existing game to display a profusion of ads on each screen.

We then search in our analyzed apps if DIFUZER detected similar SHSOs. Eventually, DIFUZER detected three apps with the exact same SHSO and the exact same service proposed to the user (walkthrough games). We tested these apps to confirm they were adware. They were also still in Google Play.

We then checked if similar "walkthrough games" were also still in Google Play and not in our initial dataset. Therefore, we searched for apps made by the same developers of the three previous apps detected by DIFUZER. We also searched for "walkthrough games" in Google Play and browsed the resulting apps. We inspected the newly collected apps and confirmed they were adware apps. Eventually, we identified 8 apps with the same adware behavior.

We contacted Google to report these 8 apps. They were removed in less than two weeks from Google Play. We make available the samples in the project's repository.

RQ3 answer: Our experiments show that SHSOs are present in benign apps and in widely-used libraries. We have seen through real-world examples that DIFUZER can reveal potentially harmful applications (PHA) and raise alarms concerning some apps' potential maliciousness. Overall, DIFUZER contributed to removing 8 adware apps from Google Play.

5 LIMITATIONS AND THREATS TO VALIDITY

An essential step in our approach is the identification of SHSOs entry-points. To do so, DIFUZER relies on state-of-the-art tool FLOWDROID [6]. Therefore, it carries the analysis limitations of FLOWDROID, i.e., unsoundness regarding reflective calls [43], dynamic loading [82], multi-threading [53] and native calls [48].

Although our approach proved to be efficient to detect SHSOs and logic bombs, feature selection can impact the performances. Indeed, feature engineering is a challenging task and can be prone to unsatisfactory selection since it does not capture *everything*.

Besides, our approach is based on SHSO entry-points detection using taint analysis, which relies on sources and sinks methods. Sinks are not an issue in our approach since they always represent *if conditions*. However, sources selection is at risk since they have been selected systematically, using heuristics and human intuitions. Therefore, our list of sources might not be complete.

Although, we have implemented TRIGGERSCOPE by strictly following the description in the original paper, our implementation might not be exempt from errors.

In the absence of a-priori ground truth, some of our assessment activities rely on manual analysis based on our own expertise. While we follow a consistent process (e.g., we carefully verify the hidden behaviour implementation against the antivirus report), our conclusions remain affected by human subjectivity. Nevertheless, we mitigate the threat to validity by sharing all our artefacts to the research community for further exploitation and verification.

6 RELATED WORK

Logic bombs in general. Hidden code triggered under specific conditions is a concern in many programming environments. The literature includes studies of the logic bomb phenomenon in programming prior to the Android era [11, 16] and targeting the Windows platform for example. Since then, various approaches have been proposed to tackle the challenging task of trigger-based behavior detection [9, 35, 38, 49, 70]. State-of-the-art techniques for the detection of trigger-based behaviour are varied and leverage fully-static analyses [26, 58, 85], dynamic analyses [86], hybrid analyses [10, 11], and machine-learning-based analyses [57].
Trigger-based behavior detection for Android DIFUZER combines static taint analysis and unsupervised machine learning techniques. Our closest related work is thus HSOMINER [57], which relies on static analysis and automatic classification to detect *HSOs*.

Contrary to our work, however, HsoMiner is not targeting suspicious HSOs and therefore does not focus on logic bombs.

Fratantonio et al. [26] proposed TriggerScope, an automated static-analysis tool that can detect logic bombs in Android apps. TriggerScope leverages a symbolic execution engine to model specific values (i.e., SMS-, time-, location-related variables). TriggerScope models conditions using *predicate recovery*. It combines symbolic execution results and path predicate recovery results to infer suspicious triggers. Finally, potential suspicious triggers undergo a control dependency step to verify if it guards sensitive operations. Nevertheless, the whole approach relies on static analysis to check defined properties of suspiciousness. In contrast, Difuzer takes advantage of unsupervised learning to discover abnormal (hence suspicious) trigger-based behavior.

Anomaly detection for security. We note that the idea of using anomaly detection to detect malware has been presented in the Avdiienko et al.'s paper [8]. Indeed, they present MudFlow that relies on anomaly detection to spot malware for which sensitive data flows deviate from benign data flows. It proved to be efficient by detecting more than 86% malware. While our approach is also based on anomaly detection to triage *abnormal* triggers (i.e., suspicious sensitive behavior) that deviate from normality (i.e., normal triggers/conditions), the end goal of both approaches is different. Indeed, MudFlow addresses a binary classification problem to discriminate malware from goodware. In contrast, Difuzer addresses the problem of detecting and locating *Suspicious Hidden Sensitive Operations* that are likely to be logic bombs in Android apps.

Malicious behavior detection in Android apps. Malware detection does not only focus on trigger-based malicious behavior. Indeed, the Android security research community worked on tackling general security aspects [12, 50, 52, 72, 89]. In the literature, numerous approaches have been proposed to detect Android hostile activities. Among which, machine-learning techniques [66], deep-learning techniques [54], static analyses through semantic-based detection [23], privacy leaks detection [6, 41, 68], as well as dynamic analyses[21, 60, 77]. Each of these approaches tackles a particular aspect of Android security. Therefore, analysts could combine our approach with the aforementioned techniques to detect a wide variety of Android malicious behavior more efficiently.

7 CONCLUSION

We proposed Difuzer, a novel approach for detecting *Suspicious Hidden Sensitive Operations* in Android apps. Difuzer combines bytecode instrumentation, static inter-procedural taint tracking, and anomaly detection for addressing the challenge of accurately spotting relevant SHSOs, which are likely logic bombs. After empirically showing that our prototype implementation can detect SHSOs with high precision (i.e., 99.02 %) in less than 35 seconds per app, we assessed its capabilities to reveal logic bombs and demonstrate that up to 30% of detected SHSOs were logic bombs. We therefore improve over the performance of the current state of the art, notably TriggerScope, which yields significantly more false positives, while detecting less logic bombs. Finally, we apply Difuzer on goodware to investigate potential SHSOs: Difuzer eventually contributed to removing 8 new adware apps from Google Play.

8 DATA AVAILABILITY

For the sake of Open Science, we provide to the community all the artifacts used in our study. In particular, we make available the datasets used during our experimentations, the source code of our prototype, the executable used for our experiments, the annotated list of our manual analyses, and a dataset of logic bombs.

The project's repository including all artefacts (tool, datasets, etc.) is available at:

https://github.com/Trustworthy-Software/Difuzer

9 ACKNOWLEDGMENT

This work was partly supported (1) by the Luxembourg National Research Fund (FNR), under projects Reprocess C21/IS/16344458 the AFR grant 14596679, (2) by the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 830892, (3) by the Luxembourg Ministry of Foreign and European Affairs through their Digital4Development (D4D) portfolio under project LuxWAyS, and (4) by the INTER Mobility project Sleepless@Seattle No 13999722.

REFERENCES

[1] Hira Agrawal, James Alberi, Lisa Bahler, Josephine Micallef, Alexandr Virodov, Mark Magenheimer, Shane Snyder, Vidroha Debroy, and Eric Wong. 2012. Detecting hidden logic bombs in critical infrastructure software. In *International Conference on Information Warfare and Security. Academic Conferences International Limited*, Vol. 1.

[2] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and optimizing detection of Android native code malware variants. *Computers & Security* 65 (2017), 230 – 246. https://doi.org/10.1016/j.cose.2016.11.011

[3] Scott Alexander-Bown. [n. d.]. *Android Security: Adding Tampering Detection to Your App.* https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app#4-1-emulator Accessed February 2021.

[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[5] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114* (2013).

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[7] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2382196.2382222

[8] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 426–436. https://doi.org/10.1109/ICSE.2015.61

[9] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware.. In *NDSS*. Citeseer.

[10] Luciano Bello and Marco Pistoia. 2018. Ares: triggering payload of evasive android malware. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2–12.

[11] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 65–88.

[12] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. 15–26.

[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (July 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[14] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 3, Article 27 (May 2011), 27 pages. https://doi.org/10.1145/1961189.1961199

[15] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *2016 IEEE Symposium on Security and Privacy (SP)*. 357–376. https://doi.org/10.1109/SP.2016.29

[16] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN)*. IEEE, 177–186.

[17] M. Choudhary and B. Kishore. 2018. HAAMD: Hybrid Analysis for Android Malware Detection. In *2018 International Conference on Computer Communication and Informatics (ICCCI)*. 1–4. https://doi.org/10.1109/ICCCI.2018.8441295

[18] Catalin Cimpanu. [n. d.]. *Play Store identified as main distribution vector for most Android malware.* https://www.zdnet.com/article/play-store-identified-as-main-distribution-vector-for-most-android-malware/ Accessed February 2021.

[19] Hitesh Dharmdasani. [n. d.]. *Android.HeHe: Malware Now Disconnects Phone Calls.* https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html Accessed December 2020.

[20] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *Security and Privacy in Communication Networks*, Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu (Eds.). Springer International Publishing, Cham, 172–192.

[21] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[22] Emre Erturk. 2012. A case study in open source software security and privacy: Android adware. In *World Congress on Internet Security (WorldCIS-2012)*. IEEE, 189–191.

[23] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 576–587.

[24] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti. 2016. ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. https://doi.org/10.1109/NTMS.2016.7792435

[25] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. 377–396. https://doi.org/10.1109/SP.2016.30

[26] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 377–396.

[27] Olga Gadyatskaya, Andra-Lidia Lezza, and Yury Zhauniarovich. 2016. Evaluation of Resource-Based App Repackaging Detection in Android. In *Secure IT Systems*, Billy Bob Brumley and Juha Röning (Eds.). Springer International Publishing, Cham, 135–151.

[28] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Trust and Trustworthy Computing*, Stefan Katzenbeisser, Edgar Weippl, L. Jean Camp, Melanie Volkamer, Mike Reiter, and Xinwen Zhang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–307.

[29] Google. [n. d.]. Build Class, https://developer.android.com/reference/android/os/Build. Accessed February 2021.

[30] Google. [n. d.]. *Context Class, https://developer.android.com/reference/android/content/Context#getSystemService(java.lang.String).* Accessed February 2021.

[31] Google. [n. d.]. Cursor Class, https://developer.android.com/reference/android/database/Cursor. Accessed February 2021.

[32] Google. [n. d.]. Sensor Class, https://developer.android.com/reference/android/hardware/Sensor. Accessed February 2021.

[33] Google. [n. d.]. SensorManager Class, https://developer.android.com/reference/android/ hardware/SensorManager. Accessed February 2021.

[34] IDC. [n. d.]. *Smartphone Market Share, https://www.idc.com/promo/smartphone-market-share/os.* Accessed January 2021.

[35] Xiaoqi Jia, Guangzhe Zhou, Qingjia Huang, Weijuan Zhang, and Donghai Tian. 2017. Findevasion: an effective environment-sensitive malware detection system for the cloud. In *International Conference on Digital Forensics and Cyber Crime*. Springer, 3–17.

[36] Mohsin Junaid, Donggang Liu, and David Kung. 2016. Dexteroid: Detecting malicious behaviors in Android apps using reverse-engineered life cycle models. *computers & security* 59 (2016), 92–117.

[37] Hyunjae Kang, Jae wook Jang, Aziz Mohaisen, and Huy Kang Kim. 2015. Detecting and Classifying Android Malware Using Static Analysis along with Creator Information. *International Journal of Distributed Sensor Networks* 11, 6 (2015), 479174. https://doi.org/10.1155/2015/479174 arXiv:https://doi.org/10.1155/2015/479174

[38] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 287–301.

[39] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. 2007. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering* 160, 1 (2007), 3–24.

[40] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction*, Görel Hedin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169.

[41] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[42] L. Li, T. F. Bissyande, and J. Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2901679

[43] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 318–329.

[44] L. Li, T. F. Bissyandé, and J. Klein. 2017. SimiDroid: Identifying and Explaining Similarities in Android Apps. In *2017 IEEE Trustcom/BigDataSE/ICESS*. 136–143. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.230

[45] L. Li, T. F. Bissyandé, J. Klein, and Y. L. Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 403–414.

[46] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67 – 95. https://doi.org/10.1016/j.infsof.2017.04.001

[47] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro. 2017. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics and Security* 12, 6 (2017), 1269–1284. https://doi.org/10.1109/TIFS.2017.2656460

[48] Cheng-Min Lin, Jyh-Horng Lin, Chyi-Ren Dow, and Chang-Ming Wen. 2011. Benchmark dalvik and native code for android system. In *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*. IEEE, 320–323.

[49] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 338–357.

[50] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. IEEE, 3–17.

[51] Linghui Luo, Eric Bodden, and Johannes Späth. 2019. A Qualitative Analysis of Android Taint-Analysis Results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 102–114. https://doi.org/10.1109/ASE.2019.00020

[52] Arvind Mahindru and Paramvir Singh. 2017. Dynamic permissions based android malware detection using machine learning techniques. In *Proceedings of the 10th innovations in software engineering conference*. 202–210.

[53] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. *ACM SIGPLAN Notices* 49, 6 (2014), 316–325.

[54] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 301–308.

[55] Trend Micro. [n. d.]. *Hacking Team Spying Tool Listens to Calls.* https://www.trendmicro.com/en_us/research/15/g/hacking-team-rcsandroid-spying-tool-listens-to-calls-roots-devices-to-get-in.html Accessed February 2021.

[56] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. 2018. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps.. In *NDSS*.

[57] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps.. In *NDSS*.

[58] Dorottya Papp, Levente Buttyán, and Zhendong Ma. 2017. Towards semi-automated detection of trigger-based behavior for software security assurance. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 1–6.

[59] N. Peiravian and X. Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 300–305. https://doi.org/10.1109/ICTAI.2013.53

[60] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security* (Amsterdam, The Netherlands) *(EuroSec '14)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2592791.2592796

[61] Heloise Pieterse and Martin S Olivier. 2012. Android botnets on the rise: Trends and characteristics. In *2012 information security for South Africa*. IEEE, 1–5.

[62] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. 2015. How Current Android Malware Seeks to Evade Automated Code Analysis. In *Information Security Theory and Practice*, Raja Naeem Akram and Sushil Jajodia (Eds.). Springer International Publishing, Cham, 187–202.

[63] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[64] Mustafa Hassan Saad, Ahmed Serageldin, and Goda Ismaeel Salama. 2015. Android spyware disease and medication. In *2015 second international conference on information security and cyber forensics (InfoSec)*. IEEE.

[65] J. Sahs and L. Khan. 2012. A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference*. 141–147. https://doi.org/10.1109/EISIC.2012.34

[66] Justin Sahs and Latifur Khan. 2012. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*. IEEE, 141–147.

[67] Jordan Samhi and Alexandre Bartel. 2021. On The (In)Effectiveness of Static Logic Bomb Detector for Android Apps. arXiv:2108.10381 [cs.CR]

[68] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein. 2021. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1398–1409. https://doi.org/10.1109/ICSE43902.2021.00126

[69] Bernhard Schölkopf, John C. Platt, John Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. 2001. Estimating the Support of a High-Dimensional Distribution. *Neural Computation* 13, 7 (2001), 1443–1471. https://doi.org/10.1162/089976601750264965 arXiv:https://doi.org/10.1162/089976601750264965

[70] Dawei Shi, Xiucun Tang, and Zhibin Ye. 2017. Detecting environment-sensitive malware based on taint analysis. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE.

[71] Maddie Stone. [n. d.]. *The Path to the Payload: Android Edition, 2019*. https://cfp.recon.cx/reconmtl2019/talk/TMHQGV/ Accessed December 2020.

[72] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. Copperdroid: Automatic reconstruction of android malware behaviors.. In *Ndss*.

[73] Oguzhan Topgul. [n. d.]. *Android Malware Evasion Techniques - Emulator Detection*. https://www.oguzhantopgul.com/2014/12/android-malware-evasion-techniques.html Accessed December 2020.

[74] Virus Total. 2020. *Virus total free online virus, malware and url scanner*. https://www.virustotal.com/en

[75] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[76] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations.

[77] Victor Van Der Veen, Herbert Bos, and Christian Rossow. 2013. Dynamic analysis of android malware. *Internet & Web Technology Master thesis, VU University Amsterdam* (2013).

[78] V. N. Vapnik. 1999. An overview of statistical learning theory. *IEEE Transactions on Neural Networks* 10, 5 (1999), 988–999. https://doi.org/10.1109/72.788640

[79] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[80] Huan Xu, Constantine Caramanis, and Shie Mannor. 2009. Robustness and Regularization of Support Vector Machines. *Journal of machine learning research* 10, 7 (2009).

[81] Lifan Xu, Dongping Zhang, Nuwan Jayasena, and John Cavazos. 2018. HADM: Hybrid Analysis for Detection of Malware. In *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*, Yaxin Bi, Supriya Kapoor, and Rahul Bhatia (Eds.). Springer International Publishing, Cham, 702–724.

[82] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. 2017. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security* 12, 7 (2017), 1529–1544.

[83] Tianda Yang, Yu Yang, Kai Qian, Dan Chia-Tien Lo, Ying Qian, and Lixin Tao. 2015. Automated detection and analysis for android ransomware. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 1338–1343.

[84] Yunqiang Chen, Xiang Sean Zhou, and T. S. Huang. 2001. One-class SVM for learning in image retrieval. In *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*, Vol. 1. 34–37 vol.1. https://doi.org/10.1109/ICIP.2001.958946

[85] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1106–1120.

[86] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*.

[87] M. Zheng, M. Sun, and J. C. S. Lui. 2014. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*. 128–133. https://doi.org/10.1109/IWCMC.2014.6906344

[88] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. 2013. AppInk: Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (Hangzhou, China) *(ASIA CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2484313.2484315

[89] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.