Improving Logic Bomb Identification in Android Apps via Context-Aware Anomaly Detection

Marco Alecci, Jordan Samhi, *Member, IEEE*, Li Li, *Senior Member, IEEE*, Tegawendé F. Bissyandé, and Jacques Klein, *Member, IEEE*

***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 toward triaging logic bombs. To that end, we develop a novel hybrid approach that combines static analysis and context-aware 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 conducting an inter-procedural data-flow analysis. Subsequently, it extracts trigger-specific features to characterize SHSOs. To detect abnormal triggers, we utilize multiple One-Class SVM models, each trained on distinct sets of similar apps to more effectively capture normal behavior patterns. To assess the added value of the context-aware analysis, we compare DIFUZER++ against a baseline approach with no context (that we name DIFUZER). We show that the context-aware analysis leads to a significant improvement in both the precision and F1 score. Furthermore, the probability of successfully triaging logic bombs among SHSOs increases from 29.7% to 58.8%. All our artifacts are released to the community.**

***Index Terms*—Logic bomb, malware, android security, static analysis, clustering, anomaly detection.**

I. 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 [1]. Yet, regularly, new threats are discovered, even in the official Google Play app store [2]. Typically, thousands of apps are regularly flagged by antivirus engines: for the year 2022 alone, the ANDROZOO [3] repository has collected over 3825000 apps, among which over 145000 apps are flagged by at least five antivirus engines hosted by VirusTotal [4]. 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 [5], [6], [7], [8], [9], [10], dynamic execution [11], [12], [13], or a combination of both [14], [15], [16], as well as the use of machine-learning [17], [18].

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 [19] 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 [20], trojan [21], ransomware [22], spyware [23], etc. [24]. Furthermore, as the trigger and the malicious code are generally independent of the core application code (i.e., their context differ), logic bombs can easily be added in legitimate apps and repackaged for distribution [25], [26], [27], [28]. 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 [29], and the fact that dynamic analyses will likely fail to detect such behaviors [30], 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 [31]. 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. We hypothesize that logic bomb code is decoupled from apps' code, since these apps can be infected with pre-existing logic bomb code, which makes the logic bomb code more suspicious than any piece of code in the app. As an example, suppose a logic bomb's triggering condition relies on location data, implemented through the `getLastKnownLocation()` Android API method, and the infected app is a calculator. In that case, both the triggering condition and the code executed would be regarded as highly suspicious and even

Manuscript received 19 April 2023; revised 11 January 2024; accepted 22 January 2024. Date of publication 26 January 2024; date of current version 4 September 2024. This work was supported in part by the Luxembourg National Research Fund (FNR), under Grant NCER22/IS/16570468/NCER-FT and in part by REPROCESS under Grant C21/IS/16344458. *(Corresponding author: Marco Alecci.)*

Marco Alecci, Tegawendé F. Bissyandé, and Jacques Klein are with the SnT, University of Luxembourg, L-1359 Esch-sur-Alzette, Luxembourg (e-mail: marco.alecci@uni.lu; tegawende.bissyande@uni.lu; jacques.klein@uni.lu).

Jordan Samhi is with the CISPA Helmholtz Center for Information Security, 66123 Saarbrücken, Germany (e-mail: jordan.samhi@cispa.de).

Li Li is with the School of Software, Beihang University, Beijing 100191, China (e-mail: lilicoding@ieee.org).

Digital Object Identifier 10.1109/TDSC.2024.3358979

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

abnormal in the context of a calculator app. Consequently, we suggest utilizing a context-aware detection technique to identify suspicious HSOs, thereby improving the probability of detecting logic bombs.

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 localization of malicious code parts. Detecting logic bombs thus provides an opportunity to localize 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 [32] was proposed as a static analysis tool to detect logic bombs: its analyses are based on heuristics and are thus 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 [31] 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 (including a large portion of benign 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 context-aware anomaly detection to reveal *Suspicious Hidden Sensitive Operations* in Android apps.

While the literature includes work [31] 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++, on the one hand, utilizes specifically-engineered features that capture the semantic properties of maliciousness. On the other hand, it groups apps based on their context using clustering techniques to ensure that anomaly detection is performed with a contextual approach, i.e., on multiple sets of similar apps rather than a single set of unrelated apps. Previous research has shown the advantages of grouping similar apps to identify malicious behavior [33] and to profile malicious apps based on their data flow signatures [34].

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 context-aware anomaly detection techniques.
- We evaluate DIFUZER++ and show its ability to reveal SHSOs with a 98.56% precision in less than 48 seconds on average per app, outperforming previous approaches.
- We demonstrate that the trigger- and behavior-specific features are relevant for triaging logic bombs among HSOs.
- We demonstrate that the context of apps is relevant to triage logic bombs among SHSOs:
  while DIFUZER (i.e., a version of DIFUZER++ with not context information) uncovers 29.7% of logic bombs among the detected SHSOs, DIFUZER++ uncovers 58.82% of logic bombs.
- We show that DIFUZER, our baseline approach is enough to outperform the state-of-the-art logic bomb detector, TRIGGERSCOPE. Indeed, DIFUZER reveals more logic bombs than TRIGGERSCOPE while yielding fewer false positives.
- We release the DIFUZER++ prototype in open-source and further make available to the research community a new Android logic bomb dataset, called DATABOMB++: https://github.com/Trustworthy-Software/DifuzerPlusPlus

Extension Disclaimer: This paper is an extension of our previous work [35] which was published at the 44th International Conference on Software Engineering 2022 (ICSE 2022). In our previous work, we presented DIFUZER, a novel hybrid approach that employs a combination of data flow analysis techniques and anomaly detection to discriminate logic bombs among SHSOs within Android apps. This extension expands upon our previous work by incorporating contextual information about apps to enhance the training of anomaly detectors and improve the distinction between normal and abnormal behavior. Our new approach will be referred to as DIFUZER++, in contrast to our baseline approach, which we will simply refer to as DIFUZER.

II. BACKGROUND AND DEFINITIONS

In this section, we first introduce *Taint Analysis* and *Anomaly Detection*, two techniques used in our approach. Then, we briefly present the two algorithms used to incorporate the context in our approach, categorizing the apps into groups of similar apps: *Latent Dirichlet Allocation (LDA)* and *K-Means*. In the last part of the section, 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 [36]. This paper relies on *One-Class Support Vector Machine* (OC-SVM) [37], 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.

Apps Categorization: Applications available on the Google Play Store are sorted into specific categories to provide users with an idea of their functionality. However, alternative methods can also be used to group apps together based on similarities, such as analyzing the app's description or other kinds of data. *Latent Dirichlet Allocation* (LDA) [38] is a probabilistic topic modeling algorithm that discovers hidden topics within a large corpus of text data. It assumes that each document is a mixture of topics, and each topic is a probability distribution over a set of words. LDA works by iteratively assigning words to topics and updating the topic distributions until convergence. The resulting topic distribution for each document and word distribution for each topic can be used for analysis and classification. *K-means* [39] is an unsupervised machine learning method that partitions a dataset into k clusters. It randomly selects k initial centers, assigns data points to their nearest center, and computes new centers as the mean of their assigned points until convergence or a maximum number of iterations is reached. Both LDA and k-means are unsupervised machine learning algorithms that group similar data together; LDA groups similar text documents into topics based on word distributions, while k-means groups data points into clusters based on similarity or distance measures.

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

Definition 1 (Trigger): A trigger is a piece of code that activates operations under certain conditions. In Fig. 1(a), 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. 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. 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.

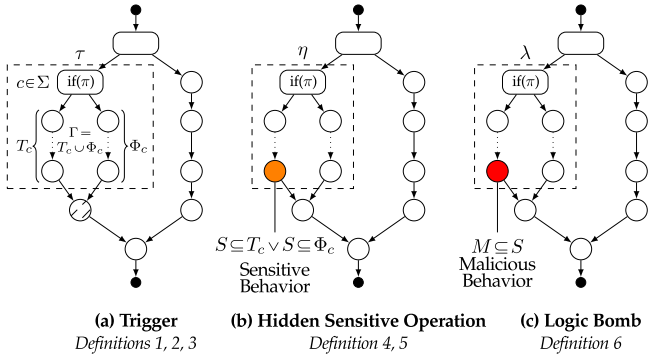

Fig. 1. Definitions illustrations. The graphs represent the Control-Flow Graph of the same function.

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. Fig. 1(b)). 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,

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

```
1  // Example simplified for reading, with renamed methods
2  public static String m1() {
3    int phoneType = telephonyManager.getPhoneType();
4    if (phoneType == 1) {
5      GsmCellLocation gsmCellLocation
       ↪  = telephonyManager.getCellLocation();
6      int a = gsmCellLocation.getCid();
7      int b = gsmCellLocation.getLac();
8      String str1 = a + b;
9    else {  String str1 = ""; }
10   return str1;
11 }
12
13 public static void m2() {
14   if (m1().isEmpty()){
15     performSomeActivity(str2);
16   }
17   else{
18     performMaliciousActivity(str2);
19   }
20 }
```

Listing 1: Logic bomb identified by DIFUZER++ in "com.xxooapp-.bubbleshot" app.

and M a piece of malicious code such as $M \subset \Sigma$. Then, λ is a logic bomb if $M \subseteq S$ (cf. Fig. 1(c)). In other words, a logic bomb is an SHSO which suspicious sensitive behavior is malicious.

Listing 1 provides an overview of a real-world example of a logic bomb that DIFUZER++ detected in an application called "com.xxooapp.bubbleshot." This application is a member of the "Bubble Shooter" game family and has a straightforward gameplay. However, an analysis of its code revealed that it attempts to retrieve the Cell ID (CID) and Location Area Code (LAC) using the `getCid()` and `getLac()` methods, respectively. In the context of mobile network communication, the CID and LAC are used to identify the specific cell tower to which a mobile device is connected, which can help determine its approximate location. This is highly unusual behavior for a simple arcade game, highlighting the need for context-aware analysis.

In this example, the different parts of the SHSO, including triggering condition checks, are split across methods $m1$ and $m2$. The triggering condition check occurs in line 4, where $m1$ returns a string with the CID and LAC information only if the `getPhoneType()` method returns 1, which corresponds to the phone type of GSM (Global System for Mobile Communications). If `getPhoneType()` returns any other value, an empty string will be returned instead. In $m2$, the malicious behavior will be activated only if the string returned by $m1$ contains the CID and LAC values.

The challenge in detecting the logic bomb described above is that traditional methods, such as rules or models, are not reliable due to the absence of a formal definition of malicious behavior. As a result, malicious code can easily evade most dynamic analyses with little effort from malware authors. This is because testing environments and sandboxes often return default values for environment variables making it difficult to detect the logic bomb [11]. For example, testing environments may always return the same value for `getPhoneType()`, thus failing to identify the malicious behavior. Besides the device's phone type, different environment values (e.g., sensors, settings, GPS, remote values, etc.) can be used to trigger malicious code. DIFUZER++ found a logic bomb that would constitute a challenge to the existing state of the art. TRIGGERSCOPE [32] cannot identify this logic bomb, as its heuristics are limited to time-, location-, and SMS-related triggers (e.g., GSM Cell values such as the value returned by `getCid()` or `getLac()` are missed). Although HSOMINER [31] could detect this logic bomb if its training set includes similar examples, its tendency to flag a large number of HSOs ($\sim$20% of apps) makes manual checking a cumbersome task. In contrast, DIFUZER++ offers a reasonable number of warnings to be checked manually. Moreover, by taking into account the category of an app, and thus, by flagging "abnormal" behavior wrt. the context of the app, we expect that DIFUZER++ can further reduce the number of false alarms (i.e., wrongly detected logic bomb) than our initial tool DIFUZER.

III. 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 [31], 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 primary objective of DIFUZER++ is to identify abnormal instances of HSOs (i.e., SHSOs), for which the likelihood of being logic bombs is high. This is achieved through a context-aware anomaly detection approach, utilizing specifically designed features.

Overview: In Fig. 2, we provide an overview of the DIFUZER++'s approach. The upper part illustrates the Application Phase of DIFUZER++, which includes all the steps executed whenever an application is given as input. In contrast, the lower section of Fig. 2 represents the DIFUZER++ Training Phase, which is performed only once to train multiple context-aware anomaly detector models. The DIFUZER++ approach comprises three key modules.

- **(1):** SHSO entry-point candidates Identification.
- **(2):** Clustering.
- **(3):** Anomaly Detection.

These modules will be explained in detail in the following subsections.

A. SHSO Entry-Point Candidates Identification

Previous works [11], [40], [41], [42], [43] 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 [32] or backward data-dependency graphs [31] 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.

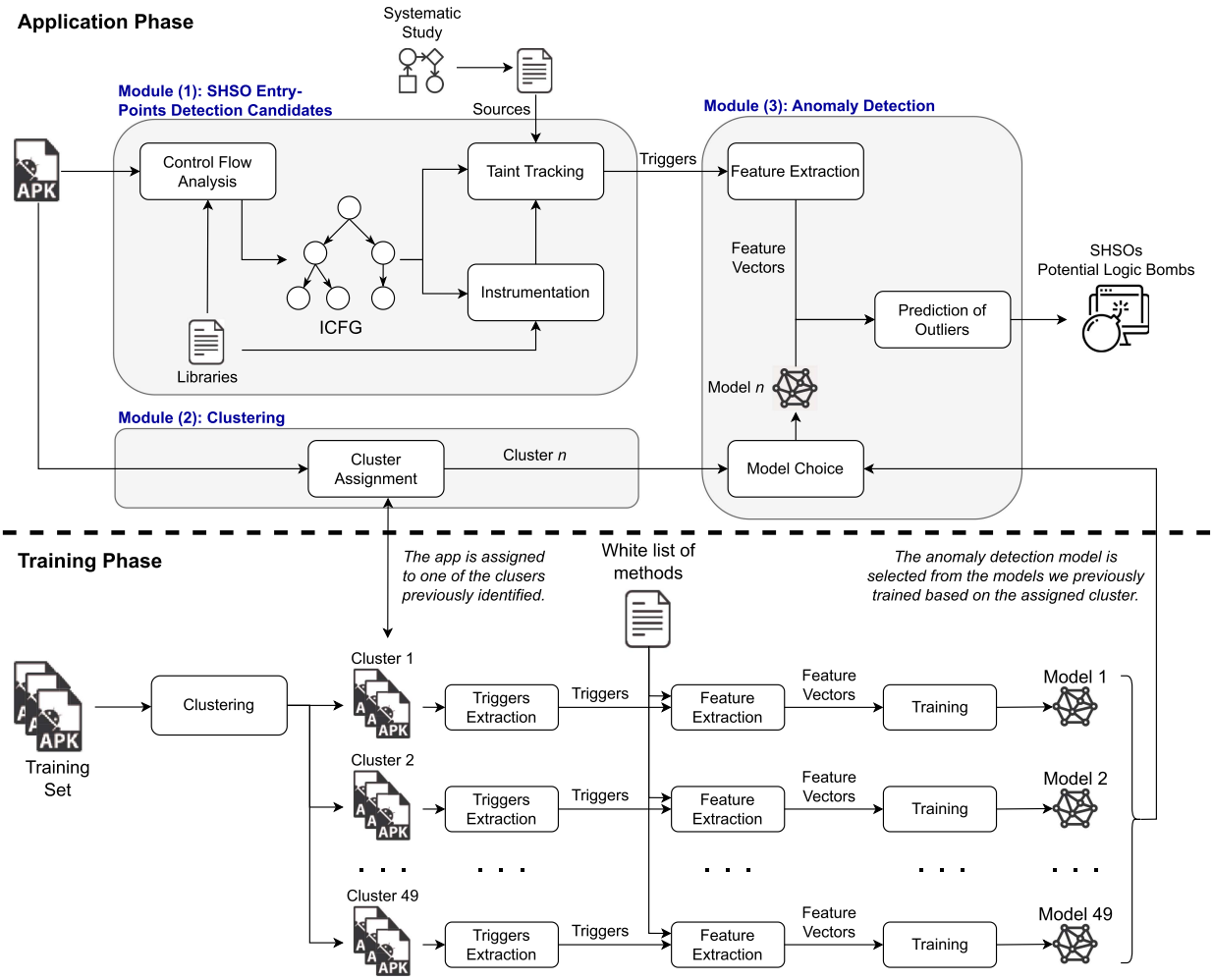

Fig. 2. Overview of the DIFUZER++ approach on a given APK file.

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 III-A2). 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 III-A1) that explores internal and external system properties and their associated APIs as well as environment variables.

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 [44] 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%) [45], [46], [47].

TABLE I
EXAMPLES OF SENSITIVE SOURCES

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

In general, decisions on whether to trigger SHSOs or not are taken on system properties [31], [40], [42], [48]. 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.

In Table I, we enumerate the different property types (with examples) on which we reasoned to retrieve sensitive sources, which are classically focused on in the literature [31], [40], [42], [48]. 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 I). 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)` [?] 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., `SENSORSERVICE` will give a `SensorManager` [?]) 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 [?]). 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` [?] 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 [?] 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 III-A2, 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).

```
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).

2) Instrumentation: Performing taint tracking, as briefly described in Section II, 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 [49], [50], [51]. 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.

Our approach overcomes this limitation by instrumenting apps. To accomplish this, the app code is first transformed into Jimple [52], the internal representation of Soot [53]. 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 III-A1, 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.

B. Module (2): Clustering

This section introduces DIFUZER++'s second module, namely the clustering module. As our final objective is to train multiple context-aware anomaly detection models on sets of similar apps, we began by forming clusters of apps (see Section III-B4), for each of which an anomaly detection model will be trained (see Section III-C3, enabling the engines to learn legitimate behavior while considering the app's context. Once the clusters have been formed, the trained clustering model is saved for future use during the application phase. Indeed, when a new application needs to be analyzed, it will be fed to the saved clustering model to determine the cluster to which it is most closely related. This identification will be crucial in the subsequent third module, as it will enable the selection of the most appropriate model for the anomaly detection phase.

1) Why a Context-Aware Analysis?: Providing context can be essential in enhancing the accuracy of Anomaly Detection models as it better helps distinguish normal from abnormal behavior. A specific behavior can be considered normal for one app but very unusual for another. For instance, a navigation app's use of the `getLastKnownLocation()` method to access position data is normal, whereas the same behavior would be considered unusual for a calculator app. Previously, in Section II, we presented a concrete example of a logic bomb related to mobile network communication that we discovered within a simple arcade game. Seeking out such contextually unusual behavior can enhance anomaly detection performance, emphasizing the necessity of context-aware analysis. Moreover, previous research has demonstrated the benefits of grouping similar apps to detect malicious behavior [33] and characterize malicious apps using their data flow signatures [34]. So, we decided to employ the same approach for DIFUZER++ by clustering apps into groups of similar apps and training an anomaly detection model for each group.

2) Categorization Techinques: The most straightforward method for grouping apps based on their similarity is to consider their assigned Google Play Category. However, several research papers have consistently highlighted the inadequacy of Google Play's current app categorization system. [54], [55], [56], [57]. As a result, we have opted to explore and compare alternative categorization methods instead of solely relying on the Google Play Category. In our extensive study on Android app categorization [54], we conducted a comprehensive evaluation of various categorization methodologies present in the existing literature. Our analysis underscored the remarkable superiority of approaches that utilize app descriptions, in contrast to those exclusively reliant on data extracted from the APK file, such as code information or XML values.

In addition, our paper [54] introduced a novel description-based approach called G-CatA, demonstrating its substantial advantages in improving tools reliant on app categorization. G-CatA, an abbreviation for **G***PT-based* **CAT***egorization of* **A***ndroid apps*, leverages OpenAI's powerful GPT-based text embedding models [58] to effectively process and represent app descriptions, using the `cl100k_base` tokenizer i.e., the same tokenizer employed in ChatGPT 3.5 and ChatGPT 4 [59].

TABLE II
APPS FILTERED FOR EACH STEP OF THE DATASET CREATION

	#Apps
Filtering from ANDROZOO *dataset*	905 929
Retrieving categories and descriptions	476 302
Removing non English descriptions	375 135

As a result, to implement context-aware anomaly detection in DIFUZER++, we opted to compare well-established strategies, such as applying ❶ LDA and ❷ K-Means to the app descriptions, along with our innovative ❸ G-CatA approach and ❹ Google Play Categories.

3) Dataset Creation: Since anomaly detection models are designed to comprehend the "normal" behavior of apps, a set of "normal" apps is necessary. To achieve this, we rely on goodware apps, aligned with the literature [35]. As mentioned earlier, in our approach to clustering apps, we rely on both their descriptions and Google Play categories. This implies the necessity of having apps with available descriptions and categories. To accomplish this, we collected all the goodware apps, defined as those with a VirusTotal score of 0, from the ANDROZOO [3] dataset over the past five years, specifically those from Google Play. (Since Google Play displays apps' descriptions and categories). In total, this resulted in 905930 apps. We used the `google-play-scraper` library [60] to obtain the Google Play category and description of each app. Furthermore, we retained only apps with English descriptions, using the `langdetect` library [61]

Table II provides a breakdown of the app count at different stages of our dataset creation process. Our final dataset comprises 375135 apps spanning across 49 distinct Google Play categories. A comprehensive list of these 49 categories is available on our repository in a file named `googlePlayCategories.csv`.

On average, each category contains approximately 7655 apps, although there is substantial variation, reflected in a significant standard deviation of 7185. For instance, the `BUSINESS` category has the most apps (33330), while the `COMICS` category has the fewest (409). This underscores the importance of not relying solely on the Google Play category to cluster apps into similar groups, as it can introduce some bias.

4) Training Phase: After assembling the dataset of goodware apps, our initial step involved preprocessing their descriptions using standard NLP techniques, such as removing non-textual items, stop-words (common words such as 'the,' 'is,' 'at,' etc.), and stemming (a process of identifying the root of a word, such as 'fishing,' 'fished,' and 'fisher,' to match the common root 'fish') [33], [62], [63]. Following the preprocessing of app descriptions, we utilized the LDA and K-means implementations from the `scikit-learn` library [64] in addition to the G-CatA approach (which is described in detail in our paper [54]) with an input of 49 as the number of clusters. We matched the number of clusters to the same number of Google Play categories to better compare the four approaches. However, further investigation into the optimal number of clusters may be considered for future work.

After categorizing the apps into 49 distinct groups, the clustering model is saved using the `joblib.dump` method from the `joblib` library [65]. When a new app is analyzed during the DIFUZER++ Application Phase, it will be possible to reload the model using the `joblib.load` method to determine which of the 49 clusters the analyzed app belongs to. This will be done after preprocessing its description in the same way as the apps in the training set.

C. Module (3): Anomaly Detection

This section presents DIFUZER++'s third module, which performs anomaly detection. After grouping the applications into clusters based on their similarities, as detailed in Section III-B, the next step involves the training of multiple anomaly detection models, with one dedicated to each cluster. The trained models are stored for future use in the analysis of new applications. Specifically, during the application phase, a single model is selected from the saved models based on the output of DIFUZER++'s second module, to ensure a context-aware analysis. After selecting the appropriate model, the features extracted from the analyzed app will be fed to the model, which will output a list of potential logic bombs.

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 [66]. 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 presents 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 [67] and its resistance to over-fitting [68]. The general idea of OC-SVM is to identify the smallest hyper-sphere to include most of the samples of the positive samples [69]. A sample considered as an outlier by the model means the data-point is not in the hyper-sphere.

2) Features Extraction: As already said, the third DIFUZER++ module's objective is to detect abnormal triggers with the intuition that these triggers are HSOs for which the likelihood of being 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. Fig. 2).

To engineer anomaly detection features, we reviewed surveys [24], [70] and related-papers [31], [71], [72], [73] 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 II).

For a given trigger, DIFUZER++ builds a feature vector $v =< S, N, D, R, B, P, M_1, S_1, J >$ 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 [7], 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 [74].

N: Is native code used in guarded code? Since analyzing native code is more challenging than Java bytecode [75], Android malware developers tend to hide malicious code from automated analyses in native code [71], [72]. 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 [73]. 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 [73] 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 [32]. To retrieve this number, DIFUZER++ queries the call-graph (built using SPARK [76] algorithm) for each method call in the guarded

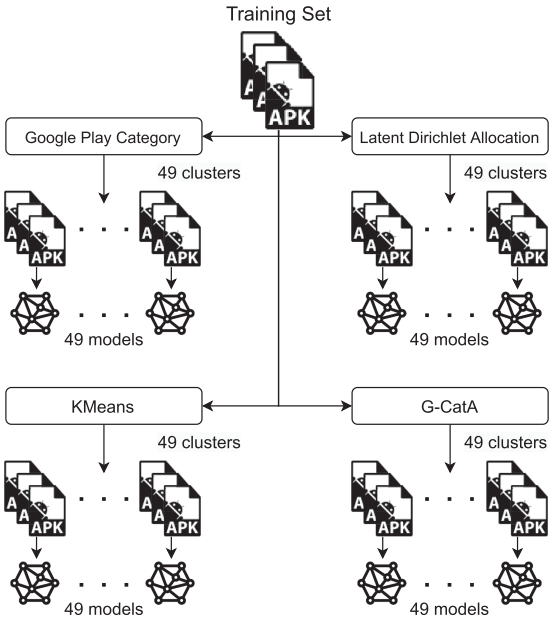

Fig. 3. Building of the anomaly detection models.

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 [31]. 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) Training Phase: As depicted in Fig. 3, we trained a total of 196 models, 49 for each of the four approaches we used for clustering (see Section III-B2). The first step consisted of extracting the feature vectors from all the apps contained in our dataset of goodware apps, i.e., the one described in Section III-B3. Then, for each group of apps, we randomly selected 10000 feature vectors from the ones extracted from apps belonging to the same group. These feature vectors were then fed into a One-Class SVM model to learn what constitutes *normal* behavior, using the implementation provided by the `scikit-learn` library [64]. 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 98.56% on average.

IV. EVALUATION

We aim to answer the following research questions to assess the efficiency of DIFUZER++ and demonstrate that context-aware analysis provides superior precision in detecting logic bombs compared to our baseline approach DIFUZER.

RQ1: How does DIFUZER, our baseline approach **without** context-aware anomaly detection, perform? We address this question in 4 sub-questions:

- *RQ1.a:* What is the performance for detecting SHSOs in Android apps?
- *RQ1.b:* Are SHSOs detected by DIFUZER likely logic bombs?
- *RQ1.c:* How does DIFUZER compare against TRIGGERSCOPE, a state-of-the-art logic bomb detector?
- *RQ1.d:* From a qualitative point of view, does DIFUZER lead to the detection of non-trivial triggers/logic bombs?

RQ2: How does DIFUZER++, our novel approach **with** context-aware anomaly detection, perform? We address this question in 2 sub-questions:

- *RQ2.a:* What is the performance for detecting SHSOs in Android apps?
- *RQ2.b:* Can DIFUZER++ find more logic bombs in the wild when the context is considered?

A. RQ1: How Does DIFUZER, Our Baseline Approach Without Context-Aware Anomaly Detection, Perform?

In this section, we evaluate our approach to detect SHSOs and logic bombs without context-aware analysis. Up until now, as outlined in Section III-B4, DIFUZER has been trained using a context-aware methodology. Therefore, to evaluate the effectiveness of DIFUZER without employing a context-aware approach, it is necessary to train an OC-SVM anomaly detector on a dataset of unrelated apps. Therefore, we randomly chose 10000 goodware (i.e., VirusToal [4] score = 0) from ANDROZOO [3]. Then, for each of these apps, we applied DIFUZER to extract a feature vector for each app's condition. Afterward, we randomly chose 10000 feature vectors[2] from those yielded by DIFUZER, which we labeled as positive (i.e., part of the normal behavior). We then trained a One-Class Classification-based anomaly detector. 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.

1) RQ1.a. 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 10000 malicious Android apps as our malicious dataset. These apps were released in 2020, collected from the ANDROZOO [3] repository, and have been flagged as malware by at least five antivirus scanners in VirusTotal.

[2]The number of extracted vectors is orders of magnitude higher. However, for efficiency, we validated that a random set of 10000 vectors yields an acceptable performance.

TABLE III
RESULTS OF THE EXPERIMENTS EXECUTED ON 10000 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

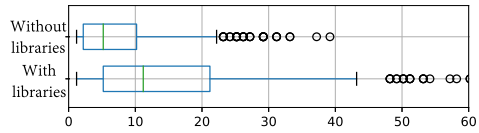

Fig. 4. Distribution of the number of SHSO(s) per app in analyses with and without libraries (only apps with at least one SHSO are considered).

We contacted the authors of state of the art approaches (e.g., HSOMINER [31], and TRIGGERSCOPE [32]) 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 [77], [78] 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 [77].

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 [77].

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 [77] as filtered. It accounts for a total of 5982 library classes and packages filtered.

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

In Table III, we report the results of applying DIFUZER (with the anomaly detection step activated) on our 10000 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. Fig. 4 further details the distribution of detected SHSOs per apps.

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 IV
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

Table III 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 10000 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 IV-A2).

We further inspect the SHSOs detected by DIFUZER by focusing on the app developer code only (we do not consider library code). Table IV 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.

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%).

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 $\pm$ 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.

Analysis Time: The last row in Table III 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 [7] and HSOMINER [31]) 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 h) before the end of the analysis.

RQ1.a answer: DIFUZER, without a context-aware anomaly detector, 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.

2) RQ1.b. 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 IV-A1, 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 [70]. 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 [31], [32], 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 [31] which detects *HSOs* in 18.7% of apps within a set of over 300000 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.

RQ1.b 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 the 46 apps in which we found logic bombs in an annotated dataset of Android apps confirmed to be using logic bombs, called DATABOMB.

3) RQ1.c. 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 [32]. As TRIGGERSCOPE does not consider the context of analyzed apps, we have chosen to compare it solely against our baseline approach DIFUZER, rather than DIFUZER++, which also takes into account contextual information.

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 III-A1 - 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 IV-A2). 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. Fig. 5 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 [79], 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).

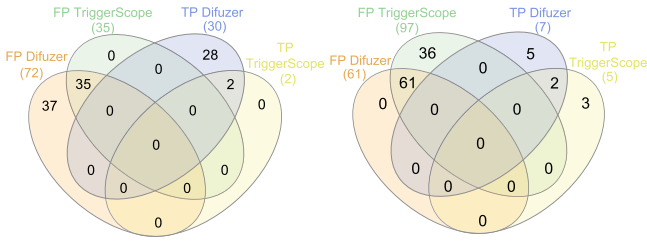

Fig. 5. Venn Diagram representing results of TRIGGERSCOPE and DIFUZER on 102 apps originally detected by DIFUZER on the left, and TRIGGERSCOPE on the right. (FP = False Positive, TP = True Positive).

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 10000 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 Fig. 5 (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.

RQ1.c 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) RQ1.d. 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.

B. RQ2: How Does DIFUZER++, Our Novel Approach **With** Context-Aware Anomaly Detection, Perform?

In this section, we evaluate our approach to detect SHSOs and logic bombs with context-aware analysis. However, we cannot reuse the initial dataset used in RQ1, as the 10000 malicious apps sourced from the ANDROZOO repository do not contain the necessary metadata for context-aware anomaly detection. For this reason, in the first sub-question RQ2.a we compare DIFUZER++ against DIFUZER on DATABOMB, i.e., the 46 Android apps containing logic bombs that have been manually verified as true positives. However, we acknowledge that DATABOMB is biased if our goal is to compare DIFUZER vs. DIFUZER++ as it only includes logic bombs that were previously identified by DIFUZER. To address this limitation, in RQ.2b, we evaluate the performance of DIFUZER++, and compare it against DIFUZER, on a new dataset of 3743 malicious apps that were never analyzed by DIFUZER.

1) RQ2.a. Incorporate Context Into DIFUZER++: With this Research Question, we aim to evaluate the performance of DIFUZER++ when incorporating context through the use of anomaly detection models trained on groups of similar apps. More specifically, we compare the results of the four clustering variant approaches (i.e., clustering with either the Google Play Categories, LDA, K-Means, and G-CatA) against our baseline DIFUZER approach.

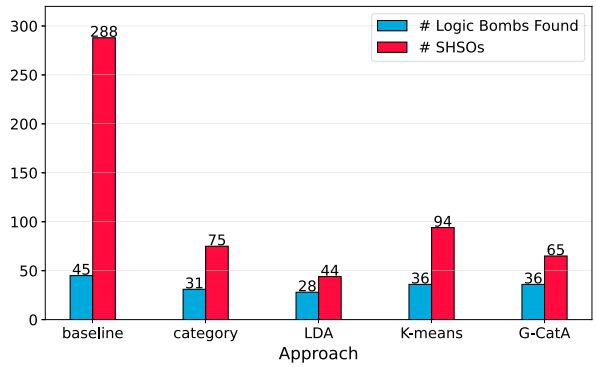

Fig. 6. Number of Logic Bombs found (blue) compared to the number of SHSOs (red) among all the different approaches.

TABLE V
EVALUATION OF CONTEXTUALIZATION APPROACHES

Approach	#SHSOs	#Logic Bombs found	Precision	Recall	F1 Score
Baseline	288	45	15.63%	100.0%	27.02%
Category	75	31	41.33%	68.89%	51.56%
LDA	44	28	63.63%	62.22%	62.92%
K-Means	94	36	38.29%	80.00%	51.79%
G-CatA	65	36	55.38%	80.00%	65.45%

Dataset: As stated before, we performed our evaluation over the 46 Android apps containing logic bombs from DATABOMB. We attempted to gather the category and description of all the apps manually but could not obtain this information for one of them (after searching extensively for different versions of the app, it seems that it has been removed from all Android app stores publicly available, including unofficial ones), resulting in a reduced dataset of 45 apps.

Evaluation: In Fig. 6, we present the results of using DIFUZER++ on our dataset of 45 apps that were confirmed to have a logic bomb. We find that when contextual information is included, DIFUZER++ fails to detect some logic bombs. However, using context-aware analysis highly reduces the number of SHSOs produced by DIFUZER++. Across all four approaches, the average reduction in the number of SHSOs is 75%.

Table V presents the Precision for each approach, defined as the ratio of the number of logic bombs to the total number of SHSOs, while the Recall indicates how many logic bombs were found. Then the F1 Score is presented and computed as

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{1}$$

The Precision increased in all scenarios, reaching 63.63% (4 times compared to our baseline approach without the context information) when using LDA to cluster the apps. These results show that, by being more precise, DIFUZER++ can speed up the identification of logic bombs. The results also show that, while being more precise, DIFUZER++ still keeps a respectable level of recall (i.e., only a limited number of logic bombs are missed). Finally, based on the F1 score, the G-CatA approach

TABLE VI
LOGIC BOMBS FOUND BY DIFUZER++ ACROSS GOOGLE PLAY CATEGORIES

Category ID	# Logic Bombs	# Logic Bombs Found
COMMUNICATION	3	0
EDUCATION	1	0
ENTERTAINMENT	9	9
GAME_ACTION	1	0
GAME_ARCADE	1	1
GAME_CASUAL	3	2
GAME_SIMULATION	1	0
GAME_SPORTS	1	0
GAME_STRATEGY	1	1
MUSIC_AND_AUDIO	3	0
TOOLS	21	18
TOTAL	45	31

has demonstrated its effectiveness as the best method for incorporating context into DIFUZER++.

Considerations About Missed Logic Bombs: While DIFUZER++ offers enhanced precision, it does come with a trade-off: it detected, on average, 27% fewer logic bombs than DIFUZER across all four approaches. However, it is crucial to consider two significant factors. First, due to the absence of ground-truth data, the evaluation was based on a biased dataset consisting only of apps that DIFUZER had previously correctly identified, inherently favoring DIFUZER in any comparison against alternative approaches. Second, a potential explanation for this variance in performance between DIFUZER and DIFUZER++ could be attributed to the context-aware analysis. Indeed, some categories may be too "heterogeneous" to improve the performance of our baseline approach, while others, characterized by more consistent app behavior, may be better suited for anomaly detection. For instance, as reported in Table VI, DIFUZER++ failed to detect logic bombs in all apps from the `COMMUNICATION` category, while successfully identifying all logic bombs in the apps belonging to the `ENTERTAINMENT` category. Similarly, the same reasoning can be applied to the LDA, K-Means, and G-CatA approaches. To address potential biases arising from the limited categories (11) in our ground truth dataset, it is essential to evaluate DIFUZER++ with a broader range of real-world apps in RQ2.b.

RQ2.a answer: Although DIFUZER++ fails to detect some of the logic bombs identified by DIFUZER, the incorporation of context improved the Precision by up to 48%. This improvement can speed up logic bomb identification.

2) RQ2.b. Logic Bombs Detection Incorporating Context: In RQ2.a, we assessed the performance of DIFUZER++ when incorporating context information. However, our evaluation was restricted to a small dataset of 45 apps with confirmed logic bombs, which may have resulted in biased results due to the limited categories present. To overcome these limitations, we conducted a comprehensive manual inspection to compare the performance of our contextual approaches DIFUZER++ against our baseline approach DIFUZER on a larger, more diverse set of applications.

Dataset: As previously stated, we cannot rely anymore on the initial dataset used in RQ1, as the 10000 malicious apps lack of metadata. Hence, we collected all malicious apps that were available on the Google Play over the past two years using the same techniques described in Section III-B3 to retrieve the Google Play Category and app description. This resulted in a final dataset of 3743 apps, categorized into 49 different categories.

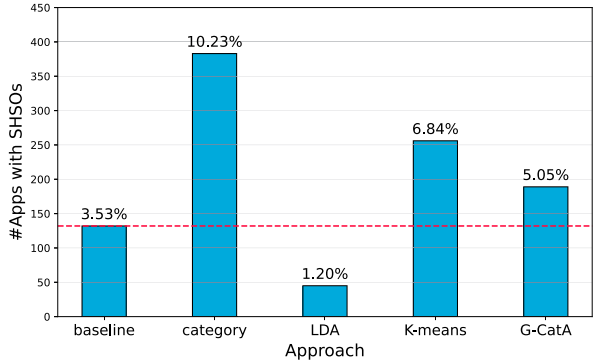

Fig. 7. Number of apps with at least one SHSO.

TABLE VII
NUMBER OF APPS FLAGGED BY DIFUZER++ AND MANUALLY INSPECTED

Approach	Apps with at least one SHSO	Apps Inspected	Apps Inspected with confirmed logic bombs	Logic Bomb Detection Rate
Baseline	132	46	6	13.04%
Category	383	58	20	34.48%
LDA	45	28	15	53.57%
K-Means	256	54	13	24.53%
G-CatA	189	51	30	58.82%
Total (with duplicates)	1005	237	83	
Total (without duplicates)	794	204	51	

Detecting SHSOs: We ran DIFUZER++ on our new dataset of 3743 malicious apps to obtain all potential SHSOs. Fig. 7 illustrates the number of apps with at least one SHSO that DIFUZER++ detected, along with the percentage of these apps relative to the total number of apps in the dataset. The results show that using the Google Play category, K-means or G-CatA leads to an increase in the number of apps flagged, while LDA is the only approach that results in a decrease in the number of flagged apps.

Although considering context may not seem effective in reducing the search space for manual analysis, we still need to assess the potential of these apps to contain logic bombs. To that end, in the next paragraph, we perform a manual analysis to check whether the flagged apps contain actual logic bombs.

Manual Analysis: As previously discussed, we aimed to assess the potential of the SHSOs found by DIFUZER++ to be logic bombs through manual analysis. DIFUZER++ identified a total of 1005 apps with at least one SHSO across all five approaches (including the baseline). While some overlap between the approaches might exist, manually analyzing so many apps would be overly time-consuming. To address this, we randomly selected a statistically significant sample for each approach with a confidence level of 90% and a confidence interval of $\pm 10\%$, reducing the number of apps to be analyzed from 1005 to 237. Table VII provides the detailed number of apps inspected for each approach.

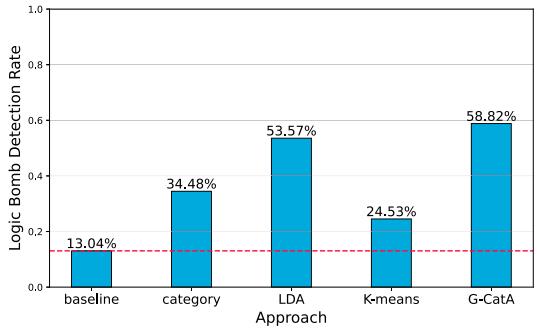

Fig. 8. Percentage of apps with confirmed Logic Bombs over the number of apps inspected for each approach.

Evaluation: The main outcome of our manual analysis is displayed in Fig. 8. For each approach, we present the Logic Bomb Detection Rate, which represents the ratio of the number of apps that were manually verified to have a logic bomb to the total number of apps we manually inspected. Table VII provides the detailed number of apps manually verified to have a logic bomb. Our findings indicate that apps identified by DIFUZER++ as potentially containing a logic bomb are more likely to indeed have one when utilizing a contextual approach. This is especially apparent when using LDA and G-CatA, where over half of the examined apps were found to contain a logic bomb. These results validate our previous intuition that while our baseline approach DIFUZER may identify fewer apps and thus reduce the scope of the search, these apps are less likely to have a logic bomb. Conversely, utilizing the Google Play Category, K-means, or G-CatA methods may require the analysis of more apps, but these have a higher probability of containing a logic bomb. Finally, using LDA has a dual impact: it not only narrows down the search space when compared to the baseline method but also increases the probability of detecting logic bombs, almost reaching the top score of 58.82% achieved by G-CatA.

DATABOMB++. Considering all the methods we employed in testing DIFUZER++, we have successfully identified a total of 83 apps that were manually confirmed to contain a logic bomb. We eliminated the logic bombs that were identified by more than one approach, resulting in 51 remaining applications that were then used to construct a new dataset of apps infected with logic bombs. We named this dataset DATABOMB++ and have made it publicly available in the project's repository as a valuable resource for the research community.

Considerations Regarding Logic Bombs Found: In Section II, we presented an example of a logic bomb that determined a device's location using methods associated with the context of mobile network communication. At the time of writing, the simple arcade game app containing the logic bomb, namely "com.xxooapp.bubbleshot," is no longer available on Google Play, along with 11 other apps that we manually confirmed to contain logic bombs. Since we did not report these apps to Google Play, we cannot definitively confirm that their removal was a direct result of the logic bombs we discovered. However, there is a high probability that these apps were removed for security reasons.

RQ2.b answer: Through our empirical study and our manual analysis, we showed that the apps detected by DIFUZER++ have a higher probability of actually containing a logic bomb when the context is taken into account. However, this can result in a wider search area when compared to the baseline. The G-CatA approach achieves the highest Logic Bomb Detection Rate of 58.82%, which means that out of 10 apps flagged by DIFUZER++, almost 6 are likely to contain a logic bomb. Furthermore, we release DATABOMB++, a dataset consisting of 51 apps that were identified by DIFUZER++ and verified to contain a logic bomb through manual analysis.

V. 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 [50]. Therefore, it carries the analysis limitations of FLOWDROID, i.e., unsoundness regarding reflective calls [80], dynamic loading [81], multi-threading [82] and native calls [83].

Although our approach proved to be efficient in detecting SHSOs and logic bombs, feature selection can impact the performance. 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.

Moreover, as we conducted a systematic mapping of the Android framework across SDK versions 3 to 30, we acknowledge that certain APIs may be deprecated (e.g., `getDeviceId` was deprecated in API level 26) or do not exist in the recent versions of Android (versions 31 to 34). However, the initial module of Difuzer++ will simply not consider them when analyzing the ICFG. It will still consider all possibly identified APIs within those specific apps as sources. Therefore, even though deprecated APIs are considered as sources, they do not impact the performance of DIFUZER++.

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.

For context-aware anomaly detection, we exclusively used apps from the Google Play for model training and testing due to its convenient access to categories and descriptions. However,

it is important to acknowledge that Google Play, while the official Android app store, is not the sole available market. Other third-party app markets may offer different selections, potentially biasing our models and limiting their representation of the full range of Android apps available.

During the Training Phase of DIFUZER++, when performing app clustering, we employed four different techniques. As previously explained in Section III-B4, we opted to match the number of clusters to the number of Google Play categories, which is 49. We acknowledge that 49 may not be the optimal number of clusters for the LDA, K-Means, and G-CatA approaches. However, we made this choice to ensure a fair comparison of DIFUZER++'s outcomes when using Google Play Categories, without introducing any bias related to the number of clusters as no ground-truth data is available for this problem.

VI. 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 [16], [84] and targeting the Windows platform for example. Since then, various approaches have been proposed to tackle the challenging task of trigger-based behavior detection [85], [86], [87], [88], [89]. State-of-the-art techniques for the detection of trigger-based behaviour are varied and leverage fully-static analyses [8], [9], [32], dynamic analyses [13], hybrid analyses [16], [90], and machine-learning-based analyses [31].

Trigger-Based Behavior Detection for Android DIFUZER++ combines static taint analysis and unsupervised machine learning techniques. Our closest related work is thus HSOMINER [31], 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. [32] 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 [91]. 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 [24], [92], [93], [94], [95]. In the literature, numerous approaches have been proposed to detect Android hostile activities. Among which, machine-learning techniques [96], deep-learning techniques [97], static analyses through semantic-based detection [98], privacy leaks detection [50], [99], [100], as well as dynamic analyses [11], [12], [101]. 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.

Context-Aware Analysis Clustering similar mobile apps together and considering the context of each app can actively improve the accuracy of anomaly detection in mobile apps. For instance, CHABADA by Gorla et al. [33] uses anomaly detection to identify malicious apps by comparing their behavior with their descriptions. This work has been extended by Ma et al. [62], who used an active semi-supervised approach, and Zhang et al. [63], who detect apps that use suspicious third-party libraries or exhibit behavior inconsistent with their descriptions. Another approach proposed by Yang et al. [34] involves characterizing malicious Android apps based on their data flow signatures, analyzing the topics of their data flows, and identifying patterns indicative of malicious behavior. Previous research has explored the benefits of context-aware analysis in detecting malicious behavior, but these studies have generally focused on identifying threats in a broad sense. In contrast, DIFUZER++ focuses on detecting logic bombs, combining static inter-procedural taint tracking with context-aware anomaly detection and leveraging features that are specifically designed for this task.

VII. 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 context-aware anomaly detection for addressing the challenge of accurately spotting relevant SHSOs, which are likely logic bombs. Our empirical evaluation of DIFUZER++ shows that it can detect SHSOs with high precision in less than 48 seconds per app. DIFUZER++ can detect up to 58.82% of logic bombs among SHSOs, which is a significant improvement over our baseline approach, DIFUZER, which only detects 29.7% of logic bombs among SHSOs and does not rely on context-aware anomaly detection. We, therefore, improve over the performance of the current state of the art, notably

TRIGGERSCOPE, which yields significantly more false positives while detecting fewer logic bombs.

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 artifacts (tool, datasets, etc.), is available at: https://github.com/Trustworthy-Software/DifuzerPlusPlus

REFERENCES

[1] IDC, Smartphone market share. Accessed: Jan., 2021. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[2] C. Cimpanu, Play store identified as main distribution vector for most android malware. Accessed: Feb., 2021. [Online]. Available: https://www.zdnet.com

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[4] V. Total, "Virus total free online virus, malware and url scanner," 2020. [Online]. Available: https://www.virustotal.com/en

[5] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "Anastasia: Android malware detection using static analysis of applications," in *Proc. 8th IFIP Int. Conf. New Technol. Mobility Secur.*, 2016, pp. 1–5.

[6] H. Kang, J. wook Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *Int. J. Distrib. Sensor Netw.*, vol. 11, no. 6, 2015, Art. no. 479174. [Online]. Available: https://doi.org/10.1155/2015/479174

[7] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 377–396.

[8] D. Papp, L. Buttyán, and Z. Ma, "Towards semi-automated detection of trigger-based behavior for software security assurance," in *Proc. 12th Int. Conf. Availability Rel. Secur.*, 2017, pp. 1–6.

[9] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1106–1120.

[10] J. Samhi et al., "Jucify: A step towards android code unification for enhanced static analysis," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, 2022, pp. 1232–1244. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3512766

[11] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *Proc. ACM 7th Eur. Workshop Syst. Secur.*, 2014. [Online]. Available: https://doi.org/10.1145/2592791.2592796

[12] V. Van Der Veen, H. Bos, and C. Rossow, "Dynamic analysis of android malware," *Internet Web Technol.*, Master thesis, VU Univ. Amsterdam, 2013. [Online]. Available: https://www.vvdveen.com/publications/MSc.pdf

[13] C. Zheng et al., "Smartdroid: An automatic system for revealing UI-based trigger conditions in android applications," in *Proc. 2nd ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2012, pp. 93–104.

[14] L. Xu, D. Zhang, N. Jayasena, and J. Cavazos, "HADM: Hybrid analysis for detection of malware," in *Proc. SAI Intell. Syst. Conf.*, Y. Bi, S. Kapoor, and R. Bhatia, Eds. Berlin, Germany, Springer International Publishing, 2018, pp. 702–724.

[15] M. Choudhary and B. Kishore, "Haamd: Hybrid analysis for android malware detection," in *Proc. Int. Conf. Comput. Commun. Inform.*, 2018, pp. 1–4.

[16] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*. Berlin, Germany: Springer, 2008, pp. 65–88.

[17] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Proc. Eur. Intell. Secur. Inform. Conf.*, 2012, pp. 141–147.

[18] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and API calls," in *Proc. IEEE 25th Int. Conf. Tools Artif. Intell.*, 2013, pp. 300–305.

[19] S. Dong et al., "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Security and Privacy in Communication Networks*, R. Beyah, B. Chang, Y. Li, and S. Zhu Eds., Berlin, Germany: Springer International Publishing, 2018, pp. 172–192.

[20] E. Erturk, "A case study in open source software security and privacy: Android adware," in *Proc. IEEE World Congr. Internet Secur.*, 2012, pp. 189–191.

[21] H. Pieterse and M. S. Olivier, "Android botnets on the rise: Trends and characteristics," in *Proc. IEEE Inf. Secur. South Afr.*, 2012, pp. 1–5.

[22] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for android ransomware," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun., IEEE 7th Int. Symp. Cyberspace Saf. Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, 2015, pp. 1338–1343.

[23] M. H. Saad, A. Serageldin, and G. I. Salama, "Android spyware disease and medication," in *Proc. IEEE 2nd Int. Conf. Inf. Secur. Cyber Forensics*, 2015, pp. 118–125.

[24] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.

[25] W. Zhou, X. Zhang, and X. Jiang, "Appink: Watermarking android apps for repackaging deterrence," in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2484313.2484315

[26] L. Li, T. F. Bissyande, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Trans. Softw. Eng.*, vol. 47, no. 4, pp. 676–693, Apr. 2021.

[27] L. Li, T. F. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 136–143.

[28] O. Gadyatskaya, A.-L. Lezza, and Y. Zhauniarovich, "Evaluation of resource-based app repackaging detection in android," in *Secure IT Systems*, B. B. Brumley and J. Röning, Eds., Berlin, Germany: Springer International Publishing, 2016, pp. 135–151.

[29] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. Amer. Math. Soc.*, vol. 74, no. 2, pp. 358–366, 1953. [Online]. Available: http://www.jstor.org/stable/1990888

[30] H. Agrawal et al., "Detecting hidden logic bombs in critical infrastructure software," in *Proc. Int. Conf. Inf. Warfare Secur. Academic Conf. Int. Limited*, 2012, pp. 1–11.

[31] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[32] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 377–396.

[33] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. ACM 36th Int. Conf. Softw. Eng.*, 2014, pp. 1025–1035. [Online]. Available: https://doi.org/10.1145/2568225.2568276

[34] X. Yang, D. Lo, L. Li, X. Xia, T. F. Bissyandé, and J. Klein, "Characterizing malicious android apps by mining topic-specific data flow signatures," *Inf. Softw. Technol.*, vol. 90, pp. 27–39, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491730366X

[35] J. Samhi, L. Li, T. F. Bissyande, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, 2022, pp. 723–735. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3510135

[36] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1541880.1541882

[37] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Comput.*, vol. 13, no. 7, pp. 1443–1471, 2001. [Online]. Available: https://doi.org/10.1162/089976601750264965

[38] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=944937

[39] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probability*, 1967, pp. 281–297.

[40] O. Topgul, Android malware evasion techniques - emulator detection. Accessed: Dec., 2020. [Online]. Available: https://www.oguzhantopgul.com/2014/12/android-malware-evasion-techniques.html

[41] S. Alexander-Bown, Android security: Adding tampering detection to your app. Accessed: Feb., 2021. [Online]. Available: https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app#4--1-emulator

[42] H. Dharmdasani, Android.hehe: Malware now disconnects phone calls. Accessed: Dec., 2020. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html

[43] T. Micro, Hacking team spying tool listens to calls. Accessed: Feb., 2021. [Online]. Available: https://www.trendmicro.com

[44] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," Univ. of Darmstadt, Tech. Rep. TUDCS-2013–0114, 2013.

[45] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 102–114.

[46] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[47] M. Junaid, D. Liu, and D. Kung, "Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models," *Comput. Secur.*, vol. 59, pp. 92–117, 2016.

[48] M. Stone, "The path to the payload: Android edition," 2019. Accessed: Dec., 2020. [Online]. Available: https://cfp.recon.cx/reconmtl2019/talk/TMHQGV/

[49] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Trust and Trustworthy Computing*, S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, Eds., Berlin, Germany: Springer, 2012, pp. 291–307.

[50] S. Arzt et al., "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[51] F. Wei, S. Roy, X. Ou, and D. Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1329–1341. [Online]. Available: https://doi.org/10.1145/2660267.2660357

[52] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," McGill University, Tech. Rep., 1998.

[53] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. CASCON 1st Decade High Impact Papers*, 2010, pp. 214–224. [Online]. Available: https://doi.org/10.1145/1925805.1925818

[54] M. Alecci, J. Samhi, T. F. Bissyandé, and J. Klein, "Revisiting android app categorization," 2023, *arXiv:2310.07290*.

[55] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Trans. Softw. Eng.*, vol. 43, no. 9, pp. 817–847, Sep. 2017.

[56] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 292–302.

[57] D. Surian, S. Seneviratne, A. Seneviratne, and S. Chawla, "App miscategorization detection: A case study on Google play," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 8, pp. 1591–1604, Aug. 2017.

[58] OpenAI, 2022. [Online]. Available: https://openai.com/blog/introducing-text-and-code-embeddings

[59] OpenAI, 2023. [Online]. Available: https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb

[60] Google play scraper library, 2019. [Online]. Available: https://pypi.org/project/google-play-scraper/

[61] Langdetect library, 2015. [Online]. Available: https://pypi.org/project/langdetect/

[62] S. Ma, S. Wang, D. Lo, R. H. Deng, and C. Sun, "Active semi-supervised approach for checking app behavior against its description," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, 2015, pp. 179–184.

[63] C. Zhang, H. Wang, R. Wang, Y. Guo, and G. Xu, "Re-checking app behavior against app description in the context of third-party libraries," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, 2018, pp. 665–710.

[64] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[65] J. D. Team, "Joblib: Running python functions as pipeline jobs," 2020. [Online]. Available: https://joblib.readthedocs.io/

[66] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerg. Artif. Intell. Appl. Comput. Eng.*, vol. 160, no. 1, pp. 3–24, 2007.

[67] V. N. Vapnik, "An overview of statistical learning theory," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 988–999, Sep. 1999.

[68] H. Xu, C. Caramanis, and S. Mannor, "Robustness and regularization of support vector machines," *J. Mach. Learn. Res.*, vol. 10, no. 7, pp. 1485–1510, 2009.

[69] Y. Chen, X. S. Zhou, and T. S. Huang, "One-class SVM for learning in image retrieval," in *Proc. Int. Conf. Image Process.*, 2001, pp. 34–37.

[70] L. Li et al., "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1269–1284, Jun. 2017.

[71] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Automating and optimizing detection of android native code malware variants," *Comput. Secur.*, vol. 65, pp. 230–246, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016740481630164X

[72] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, "How current android malware seeks to evade automated code analysis," in *Information Security Theory and Practice*, R. N. Akram and S. Jajodia, Eds., Berlin, Germany: Springer International Publishing, 2015, pp. 187–202.

[73] M. Zheng, M. Sun, and J. C. S. Lui, "DroidTrace: A ptrace based android dynamic analysis system with forward execution capability," in *Proc. Int. Wirel. Commun. Mobile Comput. Conf.*, 2014, pp. 128–133.

[74] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 217–228. [Online]. Available: https://doi.org/10.1145/2382196.2382222

[75] L. Li et al., "Static analysis of android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917302987

[76] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Compiler Construction*, G. Hedin, Ed., Berlin, Germany: Springer, 2003, pp. 153–169.

[77] L. Li, T. F. Bissyandé, J. Klein, and Y. L. Traon, "An investigation into the use of common libraries in android apps," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 403–414.

[78] K. Chen et al., "Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and IoS," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 357–376.

[79] J. Samhi and A. Bartel, "On the (in)effectiveness of static logic bomb detector for android apps," in *IEEE Trans. Dependable Secure Comput.*, Aug. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9524530

[80] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 318–329.

[81] Y. Xue et al., "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 7, pp. 1529–1544, Jul. 2017.

[82] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 316–325, 2014.

[83] C.-M. Lin, J.-H. Lin, C.-R. Dow, and C.-M. Wen, "Benchmark dalvik and native code for android system," in *Proc. IEEE 2nd Int. Conf. Innov. Bio-Inspired Comput. Appl.*, 2011, pp. 320–323.

[84] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC*, 2008, pp. 177–186.

[85] D. Shi, X. Tang, and Z. Ye, "Detecting environment-sensitive malware based on taint analysis," in *Proc. IEEE 8th Int. Conf. Softw. Eng. Serv. Sci.*, 2017, pp. 322–327.

[86] X. Jia, G. Zhou, Q. Huang, W. Zhang, and D. Tian, "FindEvasion: An effective environment-sensitive malware detection system for the cloud," in *Proc. Int. Conf. Digit. Forensics Cyber Crime*, Springer, 2017, pp. 3–17.

[87] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, Springer, 2011, pp. 338–357.

[88] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 287–301.

[89] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient detection of split personalities in malware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Citeseer, 2010, pp. 1–16.

[90] L. Bello and M. Pistoia, "Ares: Triggering payload of evasive android malware," in *Proc. IEEE/ACM 5th Int. Conf. Mobile Softw. Eng. Syst.*, 2018, pp. 2–12.

[91] V. Avdiienko et al., "Mining apps for abnormal usage of sensitive data," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 426–436.

[92] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.

[93] 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. IEEE 3rd Int. Workshop Building Anal. Datasets Gathering Experience Returns Secur.*, 2014, pp. 3–17.

[94] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[95] A. Mahindru and P. Singh, "Dynamic permissions based android malware detection using machine learning techniques," in *Proc. 10th Innov. Softw. Eng. Conf.*, 2017, pp. 202–210.

[96] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Proc. IEEE Eur. Intell. Secur. Inform. Conf.*, 2012, pp. 141–147.

[97] N. McLaughlin et al., "Deep android malware detection," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 301–308.

[98] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 576–587.

[99] L. Li et al., "IccTA: Detecting inter-component privacy leaks in android apps," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 280–291.

[100] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein, "RAICC: Revealing atypical inter-component communication in android apps," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1398–1409. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00126

[101] W. Enck et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, 2014.