On The (In)Effectiveness of Static Logic Bomb Detection for Android Apps

Jordan Samhi and Alexandre Bartel

Abstract—Android is present in more than 85% of mobile devices, making it a prime target for malware. Malicious code is becoming increasingly sophisticated and relies on *logic bombs* to hide itself from dynamic analysis. In this article, we perform a large scale study of TSOpen, our open-source implementation of the state-of-the-art static logic bomb scanner TriggerScope, on more than 500k Android applications. Results indicate that the approach scales. Moreover, we investigate the discrepancies and show that the approach can reach a very low false-positive rate, 0.3%, but at a particular cost, e.g., removing 90% of sensitive methods. Therefore, it might not be realistic to rely on such an approach to automatically detect *all* logic bombs in large datasets. However, it could be used to speed up the location of malicious code, for instance, while reverse engineering applications. We also present TrigDB a database of 68 Android applications containing trigger-based behavior as a ground-truth to the research community.

Index Terms—Logic bombs, trigger analysis, static analysis, android applications security

1 Introduction

Android is the most popular mobile operating system with more than 85% of the market share in 2020 [1], which undeniably makes it a target of choice for attackers. Fortunately, Google set up different solutions to secure access for applications in their *Google Play*. It ranges from fully-automated programs using state-of-the-art technologies (e.g., Google Play Protect [2]) to manual reviews of randomly selected applications. The predominant opinion is that the *Google Play* market is considered relatively malware-free. However, as automated techniques are not entirely reliable and manually reviewing every submitted application is not possible, they continuously improve their solutions' precision and continue to analyze already-present-in-store applications.

Consequently, the main challenge for attackers is to build malicious applications that remain under the radar of automated techniques. For this purpose, they can obfuscate the code to make the analysis more difficult. Example of obfuscation includes code manipulation techniques [3], use of dynamic code loading [4] or use of the Java reflection API [5]. Attackers can also use other techniques such as packing [6] which relies on encryption to hide their malicious code. In this paper, we focus on one type of evasion technique based on logic bombs. A logic bomb is code logic which executes malicious code only when particular conditions are met.

- *Jordan Samhi is with the University of Luxembourg, 4365 Esch-sur-Alzette, Luxembourg. E-mail: jordan.samhi@uni.lu.*
- *Alexandre Bartel is with the University of Luxembourg, 4365 Esch-sur-Alzette, Luxembourg, and with the University of Copenhagen, 1165 København, Denmark, and also with the Umeå University, 90736 Umeå, Sweden. E-mail: alexandre.bartel@cs.umu.se.*

Manuscript received 24 July 2020; revised 3 Dec. 2020; accepted 23 Aug. 2021. Date of publication 27 Aug. 2021; date of current version 11 Nov. 2022. (Corresponding Author: Jordan Samhi.)
Digital Object Identifier no. 10.1109/TDSC.2021.3108057

A classic example would be malicious code triggered only if the application is not running in a sandboxed environment or after a hard-coded date, making it invisible for dynamic analyses. This behavior shows how simple code logic can defeat most dynamic analyses leading to undetected malicious applications.

In the last decade, researchers have developed multiple tools to help detecting logic bombs [7], [8], [9]. Most of them are either not fully automated, not generic or have a low recall. However, one approach, TriggerScope [10] stands out because it is fully automated and has a false positive rate close to 0.3%. In this paper, we try to replicate TriggerScope and perform a large-scale study to show that the approach scales. Furthermore, we identify specific parameters that have a direct impact on the false positive rate.

As TriggerScope is not publicly available and the authors cannot share the tool, we implement their approach as an open-source version called TSOpen. Although TSOpen has been implemented by faithfully following the details of the approach given in TriggerScope paper, we did not use the same programming language, i.e., C++. We used Java to reuse publicly available and well tested state-of-the-art solutions. Indeed, our solution relies on the so-called Soot framework [11] to convert the Java bytecode into an intermediate representation called Jimple [12] and to automatically construct control flow graphs. Also, to model the Android framework, the life-cycle of each component and the inter-component communication, TSOpen relies on algorithms from FlowDroid [13].

We use TSOpen to conduct a large-scale analysis to see if such a static approach is scalable. We ran TSOpen over a set of 508122 applications from a well-known database of Android applications named Androzoo [14]. This experiment shows that the approach is scalable but yields a high false-positive rate. Hence, because of this high false-positive rate, the approach might not be suitable to detect *all* logic bombs automatically. More than 99651 applications were

flagged with 522300 triggers supposedly malicious, yielding a false-positive rate of more than 17%.

Since we obtain a false-positive rate which is much higher than in the literature, we investigate the discrepancies. We construct multiple datasets to consider the concept drift effect, which could affect the results shown by Jordaney *et al.* [15]. Moreover, we also investigate multiple aspects of the implementation, such as the list of sensitive methods, the call-graph construction algorithm or the timeout threshold. Furthermore, we applied additional two filters not mentioned in the literature: (1) Purely symbolic values removal and (2) Different package name removal. Results indicate that to get close to a false positive rate of 0.3%, either aggressive filters should be put in place or a short list of sensitive methods should be used. In both cases, the impact on the false-negative rate is considerable. This means that if the approach is usable in practice with a low false-positive rate, it might miss many applications containing logic bombs.

We have sent the paper to the TRIGGERSCOPE's authors, who gave us positive feedback and did not see any significant issue regarding our approach nor on TSOPEN's design.

In summary, we present the following contributions :

- We implement TSOPEN, the first open-source version of the state-of-the-art approach for detecting logic bombs and show that this approach might not be appropriate for *automatically* detecting logic bombs because it yields too many false-positives.
- We conduct a large-scale analysis over a set of more than 500000 Android applications. While the approach is theoretically not scalable because it relies on NP-hard algorithms, we find that, in practice, 80% of the applications can be analyzed.
- We conduct multiple experiments on the approach's parameters to see the impact on the false positive rate and identify that a low false-positive rate can be reached but at the expense, for instance, of missing a large number of sensitive methods.
- We experimentally show that TRIGGERSCOPE's approach might not be usable in a realistic setting to detect logic bombs with the information given in the original paper. We empirically show that using TRIGGERSCOPE's approach, *trigger analysis* is not sufficient to detect logic bombs.
- We publicly release a database of Android apps containing logic bombs as a ground-truth for future research.

We make available our implementation of TSOPEN and TRIGDB with datasets to reproduce our experimental results: https://github.com/JordanSamhi/TSOpen

The remainder of the paper is organized as follows. First, a motivation example is given in Section 2 in order to clarify the reason we are studying logic bombs. Afterward comes the overview of TRIGGERSCOPE in Section 3, in which we give an overview of its structure. We evaluate TSOPEN in Section 4. Subsequently, in Section 5, we discuss the limitations of our work and the reference paper [10]. Section 6 relates similar state-of-the-art works in the context of detecting anti-reverse-engineering practices. Finally, in Section 8 we present the direction in which we will continue our research.

2 MOTIVATION

We consider two motivating examples. The first one is an application that seems legit but embeds a time-bomb that is triggered at a specific date: let's say two weeks after the installation date. This would mean that the malicious code will remain silent for some time before being triggered. Listing 1 is a concrete example of such a logic bomb. It is located at line 4 within the `onStart()` method, and it triggers the malicious code when a user opens the application and the date is reached.

We can see that with a minimum of effort, a malware developer can bypass most of the dynamic analyses which study the behavior of applications by monitoring them. Petsas & al. [16] give interesting results regarding simple solutions to bypass state-of-the-art dynamic analyses like Andrubis [17] and CopperDroid [18]. The idea is to keep the malicious code dormant during dynamic analyses by hiding it behind unexplored branches. Nevertheless, recent works show that dynamic analyses improve code coverage during the execution of an application by forcing path exploration. They do so by instrumenting the application to modify the control flow. However, these approaches face several limitations, e.g., GRODDDROID [19] does not force all branches but only those that guard malicious code which they have to detect beforehand, reducing the problem to detecting malicious code. Likewise, X-FORCE [20] does not force all branches and can force unfeasible paths, making it unsound for analyses.

The first example in Listing 1 was not part of a targeted attack but more in the idea of a widespread malicious application. Let's now consider a State-Sponsored Attack or an Advanced Persistent Threat [21] which targets specific devices. Those devices could embed seemingly legitimate applications that contain, e.g., an SMS-bomb and remain undetected by most of the analysis tools.

Listing 1. Time-Bomb Example.

```
1 protected void onStart() {
2   Date now = new Date();
3   Date attackDate = installDate.plusDays(14);
4   if(now.after(attackDate)){attack();}|}
5 }
```

Listing 2. SMS-Bomb Example.

```
1 public void onReceive(Context c, Intent i) {
2     SmsMessage sms = getIncomingSms(i);
3     String body = sms.getMessageBody();
4     if(body.startsWith("!CMD:"))
5         {processCmd(getCmdFromBody(body));}|
6 }
```

Such a logic bomb could be used as a backdoor to steal sensitive user data. Indeed, if it is a well prepared targeted attack, the attacker could send an SMS with a specific string recognized by the application. Then the application would leak wanted information and stop the broadcast of the SMS to other applications supposed to receive it. This example shows how important it is to detect as many logic bombs as

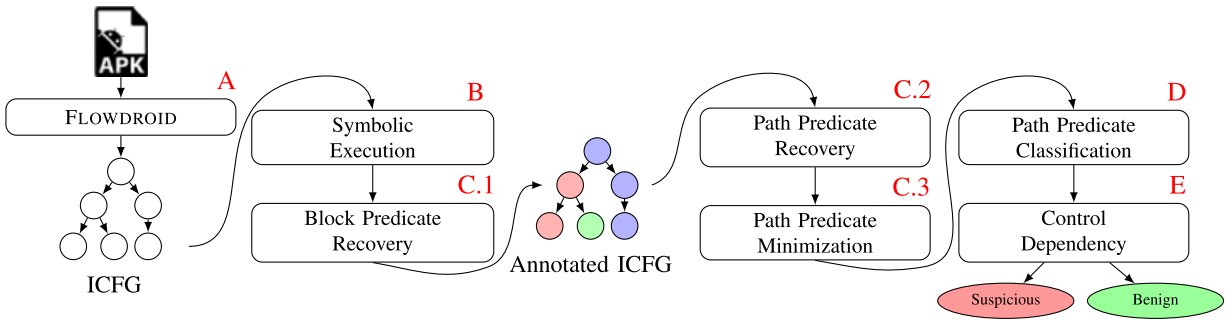

Fig. 1. Overview TSOpen. First, the application APK is processed by FLOWDROID to model the application. Then, every step of the analysis is applied to the ICFG until the final decision of the application's suspiciousness is taken by our tool.

possible in applications, especially in mobile devices, which nowadays store a lot of personal information.

Listing 2 shows an implementation of such a threat in the method `onReceive(Context c, Intent i)` of `BroadcastReceiver` class which is triggered when, in this case, an SMS is received by the device. First, the body of the SMS is retrieved in line 3. Then, at line 4, it is compared against "! CMD:" which is a hardcoded string matched against the body of an SMS. This check is considered suspicious because the application can match a hardcoded string against any incoming SMS. Hence it can wait for external commands to be executed by the malicious part of the application. Note that it could be harmless. That is why it is suspicious and not malicious. If the condition is satisfied, the command is retrieved from the SMS, and the malicious code is activated at line 5.

3 OVERVIEW

In this section, we explain TSOPEN, an open-source implementation of TRIGGERSCOPE, at the conceptual level. The approach is summarized in Fig. 1.

3.1 Applications Representation

Android apps do not have a single entry point like usual Java programs. They are made of components, each one having its life-cycle managed by the Android framework.

Modeling life-cycles and how components are connected is not trivial. That is why TSOPEN relies on FLOWDROID [13]. Indeed, FLOWDROID handles intra-component communications by introducing dummy main methods and opaque predicates to guarantee that any execution order would not influence any static analysis over the model. FLOWDROID also relies on ICCTA [22] to model the inter-component communications thanks to EPICC [23]. Using state-of-the-art solutions allowed us to avoid reimplementing the Android framework modeling reducing implementation errors. Thus, we retrieve an *interprocedural control flow graph* on which we can run static analysis algorithms (step A in Fig. 1).

Now that we have a model of the application that can be seen in Appendix D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2021.3108057., we can run our analysis, starting with the symbolic execution.

3.2 Symbolic Execution

When classifying predicates, the program has to make decisions depending on the type of objects in conditions, i.e., the condition's semantics. Therefore, this analysis models, using symbolic execution (step B in Fig. 1), the values and the operations performed over Java objects. More precisely, as we faithfully implemented TRIGGERSCOPE, we focus on modeling strings, integers, location, SMS, and time-related objects. Also, these interesting objects are annotated in order to ease the classification.

Furthermore, the classification cannot be done without retrieving the instructions guarded by a condition, that is why the next step, i.e., the predicate recovery, is essential.

3.3 Predicate Recovery

An essential step of the analysis is to construct the intra-procedural path predicate related to each instruction to build the logical formula leading to the instruction. For this, we operate as follows:

Let $ICFG = (I_r, E_r)$ the directed graph describing the interprocedural control flow graph given by FLOWDROID where, I_r represents the set of reachable instructions of the program and $E_r \subseteq I_r \times I_r$ corresponds to the set of reachable directed edges of the program represented by a pair of instructions (i_a, i_b) indicating that the flow goes from i_a to i_b. Let C_r the set of reachable conditions of the program and $\Gamma^-(i) = \{x | (x, i) \in E_r\}$ the predecessor function.

The algorithm to retrieve the full path predicate of each instruction is described as follows:

1) $\forall i \in I_r, \forall e = \{(x, i) | x \in \Gamma^-(i)\} \in E_r$, annotate e with the closest preceding condition $c \in C_r$ (step C.1 in Fig. 1).
2) $\forall i \in I_r$ annotate i with $p = getFormula(i) = \{\bigvee(getFormula(x) \wedge c) | x \in \Gamma^-(i), c$ the condition annotated on edge $(x, i)\}$ (step C.2 in Fig. 1).
3) $\forall i \in I_r$, simplify the formula p with the basic laws of Boolean algebra, p is the full intraprocedural path predicate annotated on i (step C.3 in Fig. 1).

The last step is essential in order to remove false dependencies of instructions. Indeed, consider the following formula: $(p \wedge q) \vee (\neg p \wedge q)$ which could have been calculated after step 2. The instruction annotated with this formula would have a false dependencies on predicate p because $(p \wedge q) \vee (\neg p \wedge q) = q \wedge (p \vee \neg p) = q \wedge 1 = q$ as defined by the distributive and complementation laws of boolean algebra. Hence, the elimination of false dependencies.

Now that we have retrieved path predicates and eliminated false dependencies, we can classify predicates.

3.4 Predicate Classification

In order to classify predicates, i.e., their potential suspiciousness, two essential characteristics are taken into account (step D in Fig. 1). First, we verify that the predicate involves a previously computed time-, SMS- or location-related object. Second, we verify the type of check performed over the object. The focus is set to comparisons with relevant previously modeled objects and hardcoded values/constants in the application. If a condition corresponds to these criteria, it is flagged as suspicious. Note that the Jimple intermediate representation of the Java bytecode is convenient for this stage as it allows analysts to access explicit object types. This step acts as a filter for the final control dependency step as it reduces the conditions to analyze by ruling out not suspicious conditions.

We can now perform the last step to check if a sensitive method is called within the guarded instructions of a suspicious condition.

3.5 Control Dependency

The last step of the approach consists in characterizing whether a condition is defined as a logic bomb (step E in Fig. 1). For this, every guarded instruction of a considered condition is checked to verify if it invokes a sensitive method. Also, TRIGGERSCOPE's developers had the idea to check whether a variable would be modified and later involved in another check, which, in turn, its guarded instructions would be similarly checked. This idea extends the range of possibilities regarding the search for logic bombs. Our implementation takes all these steps into account.

The interested reader can find further details about the implementation in Appendix A, available in the online supplemental material. Furthermore, Appendix H, available in the online supplemental material, shows an execution example of the analysis.

4 EVALUATION

In this section we evaluate TSOPEN and address the following research questions:

- RQ1: Does TSOPEN's approach scale?
- RQ2: What parameters can impact the false positive rate?
- RQ3: Is it possible to locate the malicious code with logic bomb detection?
- RQ4: Do benign and malicious applications use similar behavior regarding the approach under study and why?
- RQ5: Are TRIGGERSCOPE's results reproducible?

Our analyses were run on a server with an Intel Xeon E5-2430 2.20GHz processor with 24 cores, and 95GB of RAM and the *High-Performance Computing* [24] equipment available at the University of Luxembourg.

4.1 RQ1: Does TSOPEN's Approach Scale?

We perform the large scale analysis on a large dataset containing 508122 applications. This dataset has been created by randomly extracting applications from the 10 million applications of Androzoo [14]. This analysis is necessary to understand why it could or could not be deployed in real-world analyses, e.g., before being accepted into a store.

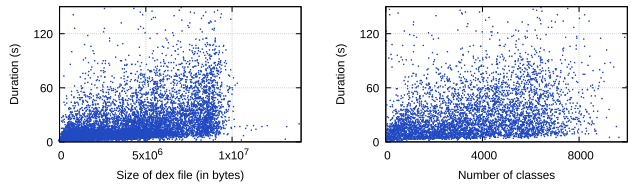

Fig. 2. Evolution of the duration of the analysis depending on the size of the dex file and the number of classes in the applications considered.

Analyzing millions of applications with a 1-hour timeout has to be parallelized to take as short a time as possible. For this purpose, we took advantage of the *High-Performance Computing* [24] equipment available at the University of Luxembourg.

We took into consideration 508122 benign and malicious applications. In fact, out of 508122 considered applications, 405810 (79.9%) were successfully analyzed with an average of 21 seconds per analysis (e.g., timeout). The proportion of applications with detected triggers with this approach is 19.6% (99651) and 34.9% (177112) without library filter (The library filter is further explained in Section 4.2.5). Also, 522300 (791364 without library filter) triggers are detected by TSOPEN among which 0.48% of SMS-related triggers, 1.35% location-related triggers, and 98.17% of time-related triggers.

Our default threshold for the timeout is one hour. Some applications cannot be analyzed within one hour. A malware developer could simply use techniques, such as obfuscation, to slow down static analysis tools to prevent the application from being analyzed. To understand how an attacker could bypass the analysis, we measured the execution time of the analysis in function of four features: (a) the size of dex files, (b) the number of classes, (c) the number of objects, and (d) the number of branches.

In Fig. 2, we can intuitively assume that there is no correlation between the size of the dex file or the number of classes in the app with the duration of the analysis. In fact, when measuring the *Pearson Correlation Coefficient (PCC)*, computed based on Equation (1), we can state that there is no correlation.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}. \quad (1)$$

In Equation (1), n is the number of data pair, x_i and y_i are data points, $\bar{x}$ and $\bar{y}$ respectively correspond to $\frac{1}{n}\sum_{i=1}^{n} x_i$ and $\frac{1}{n}\sum_{i=1}^{n} y_i$.

The correlation coefficient computed for the data corresponding to the duration as a function of the dex size is equal to 0.152. With its value close to 0, we can say that the size of the bytecode of an application does not influence the duration of the analysis, this means that even if an attacker naively introduces libraries or code to bring noise in the analysis, e.g., with dead code, it will not force the analysis to reach the timeout. Similarly, this type of analysis does not seem to be sensitive about the number of classes in an application as the *PCC* computed for the data corresponding to the duration as a function of the number of classes is equals to 0.153. The same conclusion can be done as for the dex file size, even with a lot of noise, meaning many classes

TABLE 1
Correlation Coefficients of the Data of Fig. 3 (PCC: Pearson Correlation Coefficient, SCC: Spearman Correlation Coefficient)

	PCC	SCC	PCC $(x_i, log(y_i))$
# of objects to time	0.359	0.908	0.795
# of branches to time	0.331	0.839	0.657

brought by obfuscation, for example, the analysis still stays efficient.

On the other hand, other features directly influence the duration of the analysis. In Table 1 representing the correlation coefficients of Fig. 3 we can see that the more objects in an application, the more time it will take to analyze the application. Indeed, while the *Pearson correlation coefficient* does not indicate any linear correlation (we can intuitively see the exponential correlation in Fig. 3) due to the *PCC* value of 0.359, the *Spearman Correlation Coefficient* computed based on Equation (2) assures us that the relationship between the variables observed can be represented using a monotonic function [25] due to a coefficient of 0.908.

$$r_s = \frac{cov(rg_x, rg_y)}{\sigma_{rg_x}, \sigma_{rg_y}}. \tag{2}$$

In Equation (2) rg_x and rg_y respectively represent the rank variables of x and y. Similarly, σ_{rg_x} and σ_{rg_y} respectively represent the standard deviations of rg_x and rg_y.

Better, as exponential functions can be approximated into linear functions by taking the logarithm of both sides, we can compute a linear correlation coefficient on $(x_i, log(y_i))$ for $i \in \{0, 1, \ldots, n\}$ (n being the number of pairs of data) and extrapolate the results for the original data. We obtain a score of 0.795, which is a strong linear correlation, assuring us that the original data is positively correlated following an exponential function.

The explanation for this exponential correlation is simple: to understand and detect logic bombs, this approach aims at retrieving the semantic of objects of interest. This means that the more objects to model, the more statements to take into account while modeling, therefore the more time the analysis will take. This can be problematic for an application with many objects or an application where the developer deliberately introduces useless objects to introduce noise in the analysis.

Additionally, we can see in Table 1 that the number of reachable branches and the duration of the analysis is, similarly to the number of objects and the duration of the analysis, positively correlated following an exponential function. Indeed, it introduces new paths, meaning many values to remember depending on the path during the symbolic

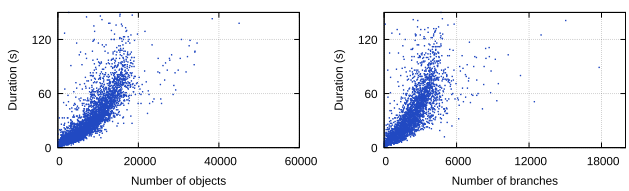

Fig. 3. Evolution of the duration of the analysis depending on the number of objects and branches in the applications considered.

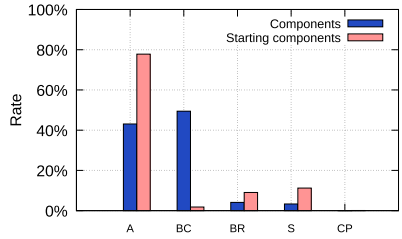

Fig. 4. Rate of components (A: Activities, BC: BasicClass, BR: BroadcastReceivers, S: Services, CP: ContentProviders).

execution. Regarding the approach used in this analysis, the most troublesome consequence of having many branches (path minimization is an NP-hard problem) is that it considerably slows down the analysis.

Furthermore, as the path predicate recovery is not necessary over the entire code of an application, we collected, afterward, on a subset of the large-scale study's applications the average time taken by the predicate recovery step. It revealed that it is responsible for 22.2% of the analysis time of an application on average and also responsible, in 35.3% of the cases, for reaching the timeout. In contrast, the symbolic execution is responsible for 61.7% of the analysis time on average, but it has to be performed over the entire application to decide to classify predicates. It must be taken into account for future work in order to optimize the number of successfully analyzed applications.

For understanding the general scheme in which the logic bombs, even false-positives, are triggered we extracted for each detected trigger the type of component in which the method triggering the logic bomb is located as well as the component where the call stack starts for reaching this method, referred to as starting component.

In Fig. 4 we can see that in a large number of cases, components containing the method triggering the logic bomb are non-Android classes (49.44%). Also, 43.1% are located in `Activities`, meaning that the trigger can also be directly embedded in the user code interface. It makes sense since many applications use time-related triggers for user interfaces (e.g., games).

If we take into account the starting components, it becomes more evident. In fact, almost 80% of the starting components are `Activites`. Many of these are likely to call a method of another class to trigger the logic bomb. Another interesting fact in starting components is that, despite the low proportion, the process of triggering often starts in a `BroadcastReceiver` or a `Service`. `BroadcastReveivers` are, in this case, mostly linked to SMS-related triggers. Regarding `Services`, we can assume that this method is likely to be used to monitor the device and trigger code at the right moment.

We also extracted two other features to understand the form of the check when detected. We wanted to know if the intra-procedural logic formula extracted during the analysis was complex or not in the general case. We can see in Fig. 5 that in the majority of the cases, there is only one predicate in the formula, which means that the triggered behavior, in the case of this particular analysis, is generally isolated not part of a multiple branch decision.

Furthermore, the density of instructions dominated by a trigger is interesting to study. Indeed, we can see that in the

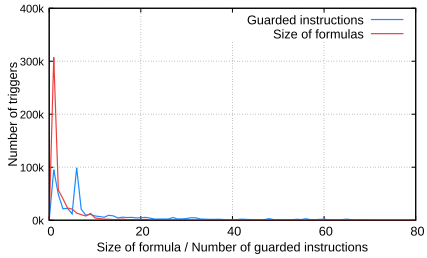

Fig. 5. Size of logical formula and count of guarded instructions.

majority of cases, the number of guarded instructions by a trigger is lesser than 10 (JIMPLE instructions). As the number of instructions is small, we can assume that those instructions represent different calls to other classes' methods to perform a useful action. This assumption correlates with the fact that in most cases, the component in which is the trigger is a basic class (see Fig. 4), that is to say, a non-Android component class. In fact, in 55.29% of the cases, one of the instructions is a method call, which confirms our previous data records of Fig. 4.

To retrieve the rate of false-positives among the 99651 detected applications, we based ourselves on VirusTotal [26]. However, the VirusTotal score is challenging to trust for qualifying an application as malware. That is why we decided to classify these applications by detection rate. Table 2 shows that the rate of false-positives reaches a lower bound of 17.2% and an upper bound of 24.6%.

We applied TSOPEN on 508122 Android applications with a success rate of 79.9%. Our experimentations show that the approach scales on large datasets. However, it also show that the approach has a high false-positive rate of 17% which would require much manual work (which the automated analysis was trying to prevent).

4.2 RQ2: What Parameters Can Impact the False Positive Rate?

The conclusion of RQ1 is surprising since we do not reach the false positive rate of the literature (0.3%). Thus, in this research question, we identify the main parameters that could significantly impact the false positive rate. Since we run many analyses, we cannot use the massive dataset of RQ1. We thus build a new smaller dataset. In order to build it, we operated as described in the literature. That is to say, we only considered benign applications from Google Play using the minimum score given by VirusTotal [26]. For this, we, again, used the Androzoo dataset [14]. Then, we analyzed the applications to check whether they contained the permission `android.permission.RECEIVE_SMS`, use a location API or a time/date library. Similarly to the literature, we selected 5803 time-related applications, 4135 location-related applications, and 1400 SMS-related applications. We ended up with a total of 11338 unique benign applications.

4.2.1 Control Experiment

The control experiment, in which we do not change any parameter, has been conducted in the same context with the timeout set to 1 hour per app. Our analysis was able to successfully analyze 7297 applications out of 11338 (i.e., 64.4%) with an average of 24 seconds per application. A success means that the analysis for an application did not reach the timeout nor crashed.

TABLE 2
VirusTotal (VT) Detection Rate of TSOPEN Flagged Applications (October 2019)

VT	>0	>10	>20	>30	>40	>50
Apps	29829	15861	7919	1237	55	1
FP Rate	17.2%	20.6%	22.6%	24.2%	24.5%	24.6%

The analysis found 9535 suspicious triggers, 4824 applications with a suspicious check, 3636 applications with suspicious triggered behavior and 3099 applications after post-filters (see Table 8 for more information) yielding a false-positive rate of 27.3%.

On a dataset with two orders of magnitude smaller than in RQ1, we find that the false positive rate still reaches a high value of more than 27%.

4.2.2 Sensitive Methods Filter

In this experiment, we randomly remove methods in the list of sensitive methods, one after the other, to observe the impact on the false positive rate. We perform this experiment 32 times to see if the results converge. Fig. 6 shows the results of this experiment. Each curve represents an experiment. We see that in order to reach a low false-positive rate (e.g., 0.38%, represented by the dotted line), we have to remove, on average, more than 11500 methods ($>$ 90%) from the list of sensitive methods, which will be missed during the analysis.

We can also see a curve diving fast on the leftmost side of the graph of Fig. 6. It represents the same filter for which the most used sensitive methods are removed first. The sensitives methods are ordered by their occurrence in logic bombs based on the results of the control experiment in Section 4.2.1.

We observe that to reach a low false-positive rate represented by the dotted line, removing the 68 most-used methods is enough. This means a concentrated number of sensitive methods are used to qualify a trigger as a logic bomb. Those methods mostly allow one to read device information, write into logs/files, and communicate with the external world.

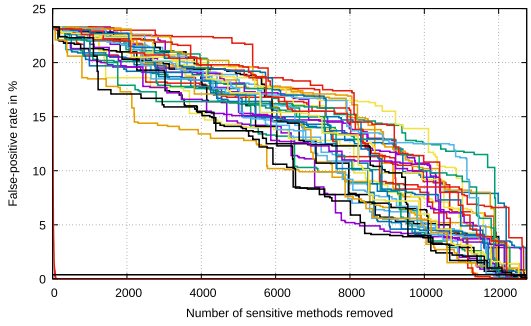

Fig. 6. Evolution of the false-positive rate as a function of the number of sensitive methods randomly removed from the list of sensitive methods considered.

TABLE 3
Experimental Results With Timeout Variation (cols. 2 and 3), Symbolic Filter (col. 4), Package Filter (col. 5)

	Original results	Timeout 2h	Timeout 3h	Symbolic values filter	Package filter
# apps analyzed	7297	9884	9897	9880	10133
Mean time of analysis	24	141.2s	146.5s	128.6s	13.2s
# of suspicious triggers	5391	7724	7727	1033	83
# of apps with triggers	1701 (23.3%)	2373 (20.9%)	2376 (21%)	381 (3.4%)	31 (0.3%)

TABLE 4
Top 15 Sensitive Methods Considered Order by Number of Occurrence in the Defaul Experiment of Section 4.2.1

Sensitive Methods	Occurences	Percentage	False-positive rate induced
TextView.setText	688	12.8%	1.6%
ConnectivityManager.getActiveNetworkInfo	600	11.1%	1.2%
File.< init >	544	10.1%	1.2%
Log.v	436	8.1%	1.6%
URL.openConnection	361	6.7%	1.3%
ContextWrapper.startActivity	361	6.7%	1.7%
Location.getLatitude	280	5.2%	0.9%
Log.println	241	4.5%	1.0%
Activity.finish	186	3.5%	0.9%
NotificationManager.notify	163	3.0%	0.6%
File.mkdirs	118	2.2%	0.5%
Handler.sendEmptyMessageDelayed	112	2.1%	1.3%
Handler.sendEmptyMessage	90	1.7%	0.6%
Handler.sendMessage	85	1.6%	0.6%
TelephonyManager.getDeviceId	77	1.4%	0.4%

We provide in Table 4 the list of sensitive methods, each present in at least 1% of logic bombs detected in the control experiment of Section 4.2.1. Those 15 methods represent 80.7% of the total of potential logic bomb yielded by our tool. Although some of these methods can be omitted in a definitive list, others like `TelephonyManager.getDeviceId`, which is considered sensitive as it can be leaked and deliver information to the attacker, have to appear in the list of sensitive methods considered. This method alone corresponds to 0.4% of the rate of false-positive. We observe that each method in the list can have a considerable impact on the false-positive rate.

From the definition of a false-positive in the literature, we can deduce that a false-negative, in this case, it is a malicious application not flagged by the tool. Therefore, to measure the rate of false-negative in our study, we use a dataset containing only malicious apps. As before, we randomly remove methods from the list of sensitive methods. Fig. 7 details our findings. We can first see that the false-negative rate starts from 45.7%, then we can see that removing sensitive method increases the rate of false-negative (while decreasing the rate of false-positive, see Fig. 6). We have seen previously that removing about 11500 sensitive methods could be helpful to reach a low false-positive rate close to 0.3%. In Fig. 7, we can see that doing so would set the false-negative rate between 70% and 95%, which would be unacceptable to detect malicious applications.

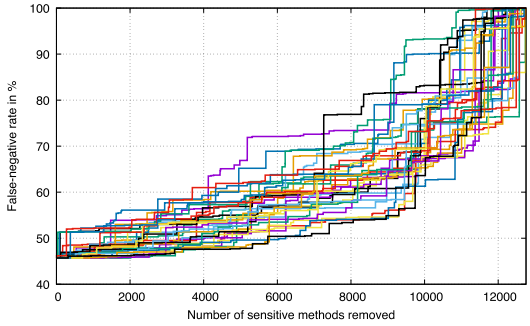

Fig. 7. Evolution of the false-negative rate as a function of the number of sensitive methods randomly removed from the list of sensitive methods considered.

> Changing the list of sensitive methods can significantly impact the false-positive rate and the false-negative rate, at least up to two orders of magnitude.

4.2.3 *Trigger Filter*

In this experiment, we modify TSOpen in order not to take into account any potential trigger if the values retrieved during the symbolic execution attributed to the test were purely symbolic or unknown. In Table 3 we can see that the number of suspicious triggers drops to 1033 and the number of applications with suspicious triggers to 381. This minor change produces results with a factor 5 change regarding the detection rate. Also, it allows our tool to get a false positive rate close to 3.4%. Unfortunately, the analysis misses all logic bombs where triggers are derived from purely symbolic values.

TABLE 5
Comparison Between TSOpen's Results Before and After Filtering Common Libraries (LB : Logic Bomb)

	Before Lib filter	After Lib filter
# Apps w/ LB	3099	1701 (-45.1%)
# Suspicious LB	9535	5391 (-43.5%)
# LB per App	3.1	3.2
# Time-related	9099	5034 (-44.6%)
# SMS-related	132	117 (-11.3%)
# Location-related	304	240 (-21%)

Using the trigger filter can have a significant impact on the false positive rate.

4.2.4 Package Filter

In this experiment, we modify TSOpen in order to only take into account methods that are in the same package as the application under analysis. This filter is stronger than the library filter of Section 4.2.5 as it constrains more the analysis. The results of Table 3 shows that the number of methods analyzed is significantly reduced. Indeed, the time taken for the analysis is shallow compared to the other analyses. Also, the number of triggers yielded by TSOpen reaches 83 in 31 different apps. The rate of false-positive is almost equal to the original one. This heuristic has a significant drawback, and an attacker could easily bypass this filter by changing the package name of classes implementing the triggered behavior. Unfortunately, the analysis does not take into account all the code outside of the package. In some applications this accounts for more than 93% of the code.

Using the package filter can have a significant impact on the false positive rate.

4.2.5 Library Filter

In this experiment, we filter out well-known libraries in order to remove noise from the results. For this, we used a list that was made in a study about common libraries [27] used in Android applications. We manually analyzed 35 of them and confirmed that they contain only false-positives.

After having filtered common libraries from the triggers found beforehand, our results reveal that a scaling approach with this analysis would still not be conceivable concerning the still high number of triggers detected. Indeed, Table 5 shows that even with a reduction of 43.5% of the number of suspicious triggers, there are 5391 suspicious triggers. Also, the number of applications flagged as containing a logic bomb goes from 3099 to 1701, a reduction of 45.1% for the tool but still greater by two orders of magnitude compared to the state-of-the-art. It means that among 11338 unique benign applications there are potentially 1701 false-positives (23.3%).

Note that, despite being conservative, the false-positive rate calculated during this experiment is obtained by counting the number of benign applications flagged by our tool containing a logic bomb. This can be explained since a logic bomb necessarily contains malicious code, otherwise, it is triggered behavior. We acknowledge that even being relatively free from malicious applications, picking applications from *Google Play* is not sufficient to qualify the dataset's applications as benign. Nevertheless, to stay in line with the literature, we use the same evaluation process.

TABLE 6
Experimental Results With Different List of Sensitive Methods

# apps analyzed	8285
Mean time of analysis	21.1s
# of suspicious triggers	2855
# of apps with triggers	956 (11.5%)

The majority of detected triggers filtered by the common libraries are time-related triggers. Out of the 4144 suspicious triggers filtered, 4065 (98.1%) are time-related whereas only 15 (0.36%) are SMS-related and 64 (1.54%) are location-related. It shows that common libraries make great use of time-related triggers. Besides, we have already said that suspicious time-related triggers definition was not narrow enough to detect it compared to SMS-related and location-related. We can say that even with an efficient library filter, time-related triggers are still commonly used in benign applications.

The library filter does not have a significant impact on the false positive rate.

4.2.6 Different List of Sensitive Methods

To build the list of sensitive methods, we reused the results of Pscout [28] and SuSi [29], as in the literature. The constructed list contains 12755 methods. Nevertheless, we have seen that with this list, we obtain a high false-positive rate. That is why we decided to verify the impact if we were to use another, shorter list of sensitive methods.

We started from the premise that a permission-based method is not necessarily sensitive. Therefore, we used a list of sink methods from FlowDroid [13] as they can leak data, which is considered sensitive. The new list features 130 methods.

Nevertheless, the number of triggers flagged by our tool (after re-running the experiment) stays relatively high by reaching 2855. They are distributed in 956 applications (11.5%, see Table 6). Even with a reduced list of methods considered for the control dependency step, the tool cannot make the difference between malicious and benign behavior. This shows the need for a more in-depth analysis of the guarded behavior of the triggers.

Using a reduced list of sensitive methods which are all involved in data leak does not have a significant impact on the false positive rate.

4.2.7 Concept Drift

Differences in results between TSOpen and existing experiments of the literature could be due to *concept drift* [15], i.e., the fact that applications used in the experiments of existing papers are older than the ones used in our paper. We launched experiments on multiple datasets of 10k apps

TABLE 7
Experimental Results With Different Call-Graph Construction Algorithms and a Reduced List of Sensitive Methods Considered

	CHA	RTA	VTA
# apps analyzed	2414	3724	7605
Mean time of analysis	37.4s	39.1s	37.4s
# of suspicious triggers	817	1887	2925
# of apps with triggers	275 (11.4%)	506 (13.6%)	957 (12.6%)

(CHA: Class Hierarchy Analysis, RTA: Rapid Type Analysis, VTA: Variable Type Analyis).

from 2013 to 2016 and have the following results for the false positive rate: 2013: 18.5%, 2014: 15.7%, 2015: 21.1% and 2016: 22.6%. We observe no significant impact on the results.

> Variation in the application release dates does not have a significant impact on the false positive rate.

4.2.8 Timeout Variation

Experiments in the literature could have been conducted a couple of years ago. To simulate the hardware available at the time, we performed the experiments with shorter timeouts. We launched experiments on multiple datasets of 10k applications and have the following results for the false-positive rate: 30min: 16.1%, 15min: 15.8%, and 5min: 15,7%. We observe no significant impact on the result.

> Reducing the timeout does not have a significant impact on the false positive rate.

4.2.9 Call-Graph Construction Algorithm

The literature might be imprecise and might not always provide all information regarding the implementation of the tool they developed. Mostly, a crucial part of performing inter-procedural analyses is the call-graph construction algorithm. Therefore, as we do not always know which call-graph algorithm is used, we renewed our previous experiment by varying the algorithm. For this, we used the following call-graph construction algorithms: SPARK [30], CHA [31], RTA [32] and VTA [33].

Table 7 reveals our experimental findings. First, we can see that none of these algorithms allow us to get a low false-positive rate close to 0.3%. It can be deduced that having the correct algorithm will not suffice to have a perfect implementation. Second, even though the results with Spark algorithm and VTA algorithm are close, we can see that changing the call-graph construction algorithm leads to different results (i.e., false-positive rates of 12.6% for VTA, 11.4% for CHA, 13.6% for RTA, and 11.5% for SPARK).

Besides, we run the same experiment by increasing the timeout to 2h and 3h. Table 3 shows the results of these experiments. We can see that there is almost no difference between a timeout of 2h and 3h. However, considering the initial timeout of 1h, the results obtained here are different. Indeed, the timeout of 1h yielded 1701 applications with triggers, against 2373 and 2376, respectively 2h and 3h. Likewise, the number of triggers detected increases from 5391 with 1h to 7724 and 7727 with respectively 2h and 3h.

> Changing the call-graph construction algorithm does not have a significant impact on the false positive rate.

> We have experimentally seen that minor changes in the implementation can have an important impact on the results. Using heuristics allows the approach to get a false-positive rate similar to the literature (0.3%). However, this result has a significant impact on the recall, the false-negative rate being raised between 70% and 95%.

4.3 RQ3: Is it Possible to Locate the Malicious Code With Logic Bomb Detection?

This is by far the most interesting question for this research area. Indeed, detecting malicious code is a difficult problem per se, that is why if this approach could help in this direction, it could be promising. In fact, TSOpen's approach is efficient for this purpose for applications taken individually. Indeed, we manually analyzed 200 apps, and we were able to locate quickly (i.e., in less than 2 minutes on average) the malicious code with the results yielded by TSOpen. When a true-positive is encountered, we were able to directly inspect the method in which the logic bomb was. Consequently, we had malicious code at hand. Note that these manual analyses allowed us to construct TrigDB, in Section IVD0c we give more details.

> During our numerous manual analyses, we were able to locate/track the malicious code easily. The condition of the logic bomb playing the role of the malicious code entry-point.

4.4 RQ4: Do Benign and Malicious Applications use Similar Behavior Regarding the Approach Under Study and Why?

In this section, we analyze randomly chosen malicious and benign Android applications containing a trigger.

a) Malicious. The first malicious application we present is called "LittlePhoto"[1] and allows a person with malicious intents to install third party applications and receive information about the device via HTTP by sending an SMS with `"$$@@&&$$"` or `"$$@@&&@@"` as the content. It can be viewed as a targeted attack which uses a logic bomb detected as `#sms/#body.equals('$$@@&&$$')`. This kind of SMS-related logic bomb is usual in remote administration tool (RAT) or SMS-based backdoors.

The second one is called "com.allen.mp"[2] and this time relies on a time-related triggered behavior : `"#now cmp 14400000L"`. After decompiling the application and analyzing it (see Appendix E, available in the online supplemental material, for an example of the code), we found that it

1. 9c92c2279a33de01561ce775c8beee9bbb58895a1f632d19f41ac2b286e12bb2.
2. 54f3c7f4a79184886e8a85a743f31743a0218ae9cc2be2a5e72c6ede33a4e66e.

checks if there is a ten days period between the current time and a pre-defined value. If the condition is satisfied, the application retrieves information about the type of operating system, the version of the Android framework, the model of the device, the number of the device, the operator, the type of network, and information about the storage. Then it sends all this information to a C&C server: `"http://search.gongfu-android.com:8511/search/sayhi.php"`.

b) Benign. The first benign application we present is called "Exam Tool"[3]. It allows students to cheat during an exam without looking at the phone but get the answer from a friend by the number of vibrations the phone would make depending on the received SMS. TSOpen flagged this application with this logic bomb : `#sms/#body.equals("zebinjo")`, it is clear that it is suspicious according to the definition of a logic bomb. However, is it malicious? Even if it calls a method categorized as sensitive (on the list of sensitive methods), the answer is no. Indeed, depending on the SMS received, the phone will vibrate according to a particular protocol defined in the application. But, a hard-coded response code is in the `onReceive` method of the `BroadcastReceiver` receiving the SMS. And when receiving exactly the string "zebinjo" it will trigger the vibration according to the following scheme : "abacdacdcadcdacdacadcacd–ca–c-da-dca-cda-c-ac-a-c-adc-a-c-a-dca-cac-a-dc-ad", the code can be seen in Appendix F, available in the online supplemental material.

This example shows why TSOpen's approach cannot be entirely reliable because of the trigger's focus and not the triggered behavior itself. It implies detecting malicious code, i.e., the problem reduces to detecting malicious code.

Another example of SMS-related triggered behavior is the family of tracker applications allowing users to get information on a device using SMS commands remotely. The application using this kind of triggered behavior we are going to analyze is "MyCarTracks"[4]. The trigger condition discovered by TSOpen is `#sms/#body.startsWith("GETPOS")` visible in Appendix G, available in the online supplemental material.

The next benign application named "TrackMe"[5] makes the use of a time-related trigger-based behavior, which is part of the definition of TSOpen's approach but is clearly legitimate. The detected logic bomb is `#now cmp 15L` which is a comparison between the current date and a numeric value of 15, see Appendinx C for the code. It appears that it is simply a check performed to verify the validity of the trial version of the application.

c) TrigDB . We have previously seen that the large-scale analysis produced a considerable number of false-positives. Therefore, manually analyzing all the applications flagged by TSOpen is unthinkable. Nevertheless, we did so on several hundred applications to verify if we could find real logic bombs.

We were able to gather 34 malicious applications containing logic bombs. We did so by reverse-engineering the applications and starting from the check found by TSOpen. That way, we had a starting point to track the malicious code. First, we checked if the application was a known malware in databases like VirusTotal [26]. It is not sufficient. Indeed we found many applications respecting this constraint, but the trigger found by the tool was not triggering the malicious code per se. Second, we tried to understand what was the purpose of the malware (also in databases like [26]) in order to compare it with the behavior of the code guarded by the trigger found by our tool. If the behaviors matched, obviously or otherwise, we kept the application for our database.

Applications are mainly *Trojans* and *Adwares*. Each application contains one or more triggers. These logic bombs are SMS-based and time-based triggers.

We also added 34 benign applications containing trigger behavior similar to malicious triggers. They mainly come from the Google Play market and are, for the majority, time-related.

For each application, we noted the triggers of interest. For each trigger, we noted the type and location in the smali code.

We believe this dataset to be useful for the community to encourage future research in finding and understanding logic bombs. Also, as this dataset is bound to grow, it can be used as a ground-truth to evaluate alternative approaches that aim at detecting logic bombs.

> Our manual investigations have shown that benign and malicious applications can use the same code for benign and/or malicious behavior. Therefore, in this case, the problem of qualifying malicious code remains. We make available a list of malicious/benign applications that make a similar usage of trigger-based behavior in TrigDB in the repository of TSOpen project.

4.5 RQ5: Are TriggerScope's Results Reproducible?

The experiments have been conducted on two datasets: the first is a dataset of malicious applications, and the second is a dataset of benign applications. To faithfully reproduce the experiments, we wanted to use the datasets of the experiments in the original paper. Unfortunately, the list of benign applications has been lost. Hence, we created a new dataset which has the same properties as the original dataset. Concerning malicious applications, 3 out of the 14 considered in their experiments were shared with us.

4.5.1 Malicious Applications

We have executed TSOpen over the three malicious applications TriggerScope's authors were willing to share to check if the same logic bombs described in their paper could be found. The first one is called *Holy Colbert*, the second one comes from the *Zitmo* malware family, and the last one is the *RCSAndroid* malware.

First, when executing TSOpen over the application called "Holy Colbert" coming from the so-called MalGenome dataset [34] we effectively find the same time-bomb as they did, but not only, we also discovered an SMS-bomb, see Appendix B, available in the online supplemental material, for more details. The SMS-bomb revealed by our tool is

3. c373d79960eadfd34fe56ad67051e2d39536c393513667b9d3c05bcd601dd874.

4. 17bf1d8afc22a40681f833cf442732795f134e3a4b287fd9f99d21db3fa07a81.

5. a22374fd3a2f6a91ab15d06be29b0a2134a1756d3aa8afe5d7cac8470b418c8d.

TABLE 8
Result of Our Analysis on the 11338 Benign Applications

Domain	# Apps	# w/ SC	# w/ STB	# After PF
Time	2967 (4950)	1719 (302)	1263 (30)	1094 (10)
Location	3305 (3430)	2366 (71)	1817 (23)	1516 (8)
SMS	1025 (1138)	739 (89)	556 (64)	489 (17)
Total	7297 (9518)	4824 (462)	3636 (117)	3099 (35)

The values in parenthesis represent TriggerScope's results for the original dataset. (SC: Suspicious Checks, STB: Suspicious Triggered Behavior, PF: Post-Filters).

`#sms/#body.matches("health")` which represents a suspicious narrow check against the body of an incoming SMS. It is triggered if the time-bomb is satisfied, meaning at a specific date, here the May, 21^{st} 2011. It triggers the deletion of data through the content resolver. This implies that our implementation is not entirely identical to the original. We do not claim that this additional finding makes our implementation more precise because it may introduce more false-positives in other applications.

Finally, regarding the "Zitmo" and "RCSAndroid" malicious applications they provided us, TSOpen was able to extract the same logic bombs as TriggerScope.

4.5.2 Reproducibility of TriggerScope's Results

In this section, we further investigate the discrepancies between TSOpen's results and TriggerScope's.

In the original paper, they had a total of 9582 unique benign applications due to overlap between categories. We made the list of the 11338 applications public in the project repository for reproducibility purposes.

First, we observe in Table 8 a considerable difference between our analysis and TriggerScope's: while TSOpen identified 3099 suspicious apps, TriggerScope identified only 35 [6]. While it is true that the two datasets are different, we did not expect to find a two order of magnitude difference between the results.

Second, we extracted, from each application, and each potential suspicious check, different features to understand and verify our results. Among them, we retrieved the class containing the suspicious check and the method in which it appears, and the sensitive method invoked to flag it. It appears that only 20 methods in the list of sensitive methods represent 89% of the sensitive methods considered to flag the suspicious checks. The list of sensitive methods that we used might explain why we have such a difference in our results. Nevertheless, according to us, it cannot explain the gap alone because other factors could impact the results. Consequently, we also analyzed the classes and methods containing suspicious triggers to verify if some distinct pattern might emerge.

We found 3165 different combinations of class/method among the 9535 suspicious triggers without any combination being overly represented. We manually analyzed the most used of them (i.e., the first 35) to verify if they were logic bombs. It seems that they enter the definition of a logic bomb according to the paper in question, but they are not. In Listing 3 we can see an example in the *card.io* library, which executes some code if some time has elapsed, then it executes the method called `android.hardware.Camera.Open()` which is considered suspicious in the list of sensitive methods. We chose this example to emphasize that most of them are based on time-related triggers. Better, we found that within the 9535 logic bombs found (among the 3099 applications flagged), only 3.1% were location-related and 1.4% were SMS-related.

We also note that most of the suspicious triggers (43.5%) are part of a library used in applications. Manual analyses revealed that they are not logic bombs, thus introduce noise in the analysis.

Listing 3. Trigger in io.card.Payment.CardScanner Class of Card.io Library (simplified). The `Camera.open()` Method (on the list of Sensitive Methods) is Triggered -Not Only- Under the Condition Triggered by the *while* Instruction.

```
1private Camera connectToCamera
(int checkInterval, int maxTimeout) {
2  long start = System.currentTimeMillis();
3  if (this.useCamera) {
4    do {try {return Camera.open();}
5  catch (RuntimeException e) {...}
6    } |\textcolor{red}{while (System.currentTi-
     meMillis() - start < maxTimeout);}|
7  } return null;
8 }
```

We observe that no filter mechanism is mentioned in TriggerScope's paper. We contacted the authors, but we could not get the information on whether a filter was used or not in their implementation. According to our results, to reach such a low rate of false-positives (0.38%), at least a library filter has to be used to rule out repeated false-positives.

The reader may have noticed the structure of the previous logic bombs, i.e., nested conditions. It raises the question of whether this type of structure is often used in trigger-based behavior. Also, we want to verify if considering nested conditions could have been treated as a single logic bomb by TriggerScope developers. Therefore, it could explain the gap between our results and theirs. We measured this and found that only 16.38% of detected triggers have a nested structure. According to this number, we can conclude that it does not impact the conclusion when comparing TriggerScope and TSOpen.

We were able to construct a dataset with the same properties as the one used in the original paper. However, TSOpen yielded a high false-positive rate by detecting 3099 potential apps with logic bombs (>27%). We see the importance of having the original list for reproducing this experiment as it can significantly impact the false-positive rate. Therefore, with the information that has been provided to us and the description of the approach made in the original paper, we conclude that the approach might be used in a realistic setting to detect logic bombs.

6. Unfortunately, TriggerScope authors were unable to run their tool on our dataset to compare the results, so we reused the results of their paper on their original dataset to compare our results.

5 Limitations

Trigger Types. Only three trigger types have been modeled, which is not representative of logic bombs, though expanding it to other types would be easy. Regarding other types of logic bombs, we recently found that a new banking Trojan named "Cerberus" made smart use of the accelerometer sensor for monitoring the device [35]. Indeed, it is based on the assumption that a real person would move with its device, hence changing the step counter's values. Only if this condition is satisfied would the malicious code be triggered.

In a recent analysis, Stone found a malicious application using multiple evading techniques [36]. The malware will first check if the device has a Bluetooth adapter and a name, which is important as emulators use default names. Then it would check if the device has a sensor and verify the content of `/proc/cpuinfo` to find both *intel* and *amd* strings. As most devices use ARM processors, those strings should not appear. It also checks the appearance of any *Bluestacks* files, which is an emulator solution and other emulation detections. Finally, this application would deliberately throw an exception and check the content to find any matching string that would show an emulator's existence.

Predicate Minimization. The next limitation lies in the fact that predicate recovery and predicate minimization are performed systematically, which increases the probability of running into a complicated formula for which the minimization step would never end. Besides, this step is responsible for 22.2% of the analysis time and responsible, in 35.3% of the cases, for reaching the timeout of the analysis. Unlike the symbolic execution step, which is responsible for the largest part of an application's analysis time (61.7%), the path predicate recovery and minimization are not necessary for the entire application. Indeed, the symbolic execution phase is necessary to decide on the suspiciousness of conditions. A countermeasure would be to locate interesting checks and then perform the full path predicate recovery and minimization.

Maliciousness. The most critical weakness of TriggerScope approach is the control dependency step, which compares method calls dominated by suspicious triggers with a list of sensitives methods. This phase requires more attention as it is used to qualify the maliciousness of a condition. Indeed, a suspicious check can be harmless due to the same usage benign and malicious applications make of trigger-based behavior. Nevertheless, we recognize the difficulty of this step, given the lack of a formal definition of malware. Despite having considerable resources, major companies also realize the difficulty to automatically qualify malicious code, e.g., Google still accepts malicious applications in its PlayStore [37].

Implementation Errors. Even though we reproduced faithfully the approach described in TriggerScope paper and reused available and well-tested state-of-the-art code when possible, we are not immune from implementation errors.

Implementation Unknowns. Given the details in the original paper, we are not able to reproduce the results. Therefore, we tried to vary parameters and implementation details to get as close as TriggerScope's results. However, it is challenging to test all the combinations of implementation/parameters to get the original results.

6 Related Work

In 2008, Brumley & al. [7] developed MineSweeper which is an interesting approach to assist an analyst. Their solution worked directly at the binary level of an executable application. Their goal was to uncover trigger-based behavior by constructing conditional paths and input values to execute the application. The next step was to ask a solver whether the path is feasible or not. If not, they would not explore this path. On the contrary, they would explore this path and ask the solver to construct -if possible- input values to satisfy the formula. They would then execute the application with the computed trigger input values to inspect the behavior, and if a malicious behavior were encountered, they would know the conditional path leading to this behavior, thus detecting if a logic bomb exists. They conducted their experiments on four real-world applications and succeeded in finding trigger-based behavior in less than 30 minutes per application with less than 14 potential logic bombs per application. Unlike TSOpen, the process is not entirely static nor fully automatic and requires a human to infer the logic bomb.

Four years later, Zheng & al. [8] focused on finding a user interface-based trigger that could be used to hide malicious code to traditional analysis in Android applications. They construct what they call the FCG (Function Call Graph) to retrieve call paths to sensitive Android APIs. The next step is constructing what they call the ACG (Activity Call Graph) to have the relationship between the application activities, i.e., how to go from one activity to another via user interface methods. Having those details, they run the application by triggering user interface elements to go to the sensitive activity, which calls a sensitive API and monitors the behavior to check if anything suspicious happens. That way, they can deduce if the user interface triggering process is used to trigger malicious code. Their approach is not generic and focuses on one single type of logic bomb. Also, we note the use of dynamic analysis of their approach again.

Pan & al. [38] presented a new machine-learning based technique to detect *Hidden Sensitive Operations* in Android apps. They do not specifically focus on malicious behavior contrary to TriggerScope's approach, which targets malicious activities. Their approach is composed of a pre-processing part where lightweight data-flow and control-flow analysis are performed to extract a condition-path graph. This latter is then used to extract features that will feed the SVM classification. Doing so, HSOMiner performs a precision of 98.4% (based on 125 randomly chosen apps from a set of 63372) and a recall of 94%. Though the approach is interesting (SVM is resistant to overfitting), it does not fit the goal of detecting logic bombs hidden in Android apps.

In 2017, Papp & al. tried to work directly at the source code level to detect trigger-based behavior in legitimate applications [9]. Their goal is not to work on a malicious application. They want to emphasize triggered behavior or backdoor behavior in legitimate open-source applications. For this purpose, they use KLEE [39] to perform mixed concrete and symbolic execution (also called concolic execution). However, first, they have to instrument the considered

application to add specific library calls to use these existing tools. Then they execute the concolic execution and based on the result, and they generate different test cases. Out of these test cases, some can be highlighted by their program to be verified by an analyst to check whether it is a trigger-based behavior or not. Though it is promising for a semi-automatic detection tool, their approach can take a large amount of time to generate test cases and lead to a large number of false-negatives.

We now present a more advanced and promising method developed by Bello and Pistoia [40]. Their approach aims at exposing evading techniques that sophisticated malware use. Nevertheless, they do not stop at the detection, as we will see. Their work is divided into three parts, the first one being the detection of *evasion point candidate* by using information flow analysis. They use the notion of source and sink, that is to say, detecting information going from a source and going to a sink. Once detected, they step into stage two, it is simply the instrumentation of the Java bytecode to force the untaken branch during the following analysis. The last stage is precisely the execution of the instrumented application in a controlled environment to monitor the malicious code's behavior. According to the authors, their approach is unsound but a stepping stone toward detecting new malware behavior. The first stage of their work is related to our work, except that they follow the flow of fingerprinting methods to branches.

Logic bombs can be used for venerable purposes, indeed in a recent study, Zeng & al. [41] presented an approach to detect repackaging using triggers. Indeed, their approach consists of instrumenting a legitimate app that could be repackaged by an attacker and adding an instruction to make the app "repackage-proof". They introduce cryptographically obfuscated trigger conditions that, when triggered, can detect if the program has undergone a repackaging process. The term "logic bomb" specifically means triggering malicious code under specific circumstances. However, the code triggered is not malicious but preventive. Therefore, although they use the same mechanism as malware developers, the authors should talk about *Hidden Preventive Code*. Nonetheless, their approach is resilient since we assume the condition has been detected, which is already a challenging problem. One cannot resolve the condition due to its cryptographic properties (using hash functions). Therefore the guarded code cannot be decrypted and executed. Inserting checks in legitimate apps is promising for protecting Android apps from malware developers and has well been studied in the literature [42].

7 Conclusion

In this paper, we have implemented TSOpen, the first open-source version of the state-of-the-art approach for detecting logic bombs in Android applications. We first conducted a large-scale analysis over a set of more than 500000 Android applications and observed that the approach scales.

However, the approach is not appropriate for automatically detecting logic bombs because the false-positives rate of 17% is too high. We conducted multiple experiments on the approach's parameters to understand the impact on the false positive rate and identify that a low false-positive rate could be reached but at the expense, for instance, of missing a large number of sensitive methods. That is to say, the approach with a low false-positive rate misses a large number of logic bombs. We experimentally show that TSOpen's approach might not be usable in a realistic setting to detect logic bombs with the original paper's information.

Moreover, we have seen that TSOpen's approach is not sufficient to detect logic bomb because benign and malicious apps can use the same code for benign and/or malicious behavior. This is a direct consequence of the lack of a formal definition of what a malware is. We empirically show that using TSOpen's approach, *trigger analysis* is not sufficient to detect logic bombs. Indeed, dissociating the trigger condition and the guarded behavior produces false-positives. Thus, an analyst is necessary to verify the behavior. Nevertheless, when manually inspecting malicious applications containing logic bombs, provided by TSOpen, we were quickly able to verify if the triggered code is malicious. We did not have to search through all the code, which saved us a lot of time. Hence, TSOpen's approach seems promising to locate the malicious code using logic bombs in applications being reverse engineered.

We created TrigDB, a database of 68 applications containing localized trigger behavior. We manually analyzed and classified each logic bomb. We make it publicly available for future research as a ground-truth promised to evolve.

In a nutshell, our work shows that, even though TSOpen's approach is interesting, it is not suitable for automatically detecting logic bombs. Indeed, the control dependency step is not sufficiently representative of malicious behavior. Our results contradict state-of-the-art by two orders of magnitude regarding the rate of false-positives. We have identified several parameters, such as the list of sensitive methods, which could have a two-order magnitude impact on the false positive rate. These parameters should be detailed in every paper tackling the challenging task of detecting logic bombs in Android applications. Our hope is that future publications do not omit this information to makes their experiments reproducible.

8 Future Work

We have seen that if an application uses trigger-based behavior combined with malicious code, they are inseparable, sophisticated techniques should be used to qualify the behavior as malicious to flag it as a logic bomb. TriggerScope checks if a sensitive method is called, but an in-depth analysis of the guarded behavior is to be preferred to understand the triggered behavior. Because a sophisticated analysis of the guarded behavior reduces to detecting malicious code, the difficulty remains due to the lack of a formal definition of what malware is. We aim at going further by linking the guarded code with its condition. The trigger would somehow be the entry point for the code. Hence we would be able to locate the malicious code, or at least be able to track it. We also want to verify if the guarded code is enough to execute a behavioral study and qualify the application's maliciousness.

We do not just want to focus on the internal context of the application. Indeed, the external context, i.e., the application description or the application reviews, can be processed by

applying NLP algorithms ([43], [44]). Existing techniques such as [45] and [46] already use the application descriptions to compare it with the permissions or the application's behavior.

Finally, we aim to build a classification of types of triggers to understand their use in malicious applications. This would help build a detector that could take into account those classes and study their use in the wild.

Acknowledgments

This work was supported by the Luxembourg National Research Fund (FNR) under Grants 12696663 and 14596679. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The authors would like to thank Dr. Francine Herrmann (University of Lorraine, Dr. Yann Lanuel (University of Lorraine) and Dr. Imen Sayar (University of Luxembourg) who have provided insightful comments for writing this paper. Also, the authors want to thank Prof. Loïc Colson (University of Lorraine) and Timothé Riom (University of Luxembourg) for their qualitative reflection on technical aspects of this paper.

References

[1] IDC, "Smartphone market share," 2020. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[2] Google, "Google play protect," 2019. [Online]. Available: https://www.android.com/play-protect/

[3] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for java bytecode," *J. Syst. Softw.*, vol. 71, pp. 1–10, 2004.

[4] S. Liang and G. Bracha, "Dynamic class loading in the java [tm] virtual machine," *ACM SIGPLAN Notices*, vol. 33, pp. 36–44, 1998.

[5] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2014, pp. 140–154.

[6] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2015, pp. 293–311.

[7] 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.

[8] 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.

[9] 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, Art. no. 64.

[10] 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.

[11] R. Vall ée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. Armonk, NY, US: IBM, 2010, pp. 214–224.

[12] R. Vallee-Rai and L. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," McGill University, Canada, Tech. Rep. 1998-4, 1998.

[13] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[14] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proc. IEEE/ACM 13th Working Conf. Mining Softw. Repositories*, 2016, pp. 468–471.

[15] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 625–642.

[16] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *Proc. 7th Eur. Workshop Syst. Secur.*, 2014, Art. no. 5.

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

[18] 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.

[19] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong, "Groddroid: A gorilla for triggering malicious behaviors," in *Proc. 10th Int. Conf. Malicious Unwanted Softw.*, 2015, pp. 119–127.

[20] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proc. 23rd SENIX Secur. Symp.*, 2014, pp. 829–844.

[21] P. Chen, L. Desmet, and C. Huygens, "A study on advanced persistent threats," in *Proc. IFIP Int. Conf. Commun. Multimedia Secur.*, 2014, pp. 63–72.

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

[23] D. Octeau *et al.*, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 543–558.

[24] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic HPC cluster: The UL experience," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2014, pp. 959–967.

[25] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: appropriate use and interpretation," *Anesthesia Analgesia*, vol. 126, no. 5, pp. 1763–1768, May 2018, doi: 10.1213/ANE.0000000000002864.

[26] V. Total, "Virustotal-free online virus, malware and URL scanner," 2012. [Online]. Available: https://www.virustotal. com/en

[27] L. Li, T. F. Bissyandé, J. Klein, and Y. Le 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.

[28] 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.

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

[30] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proc. Int. Conf. Compiler Construction*, 2003, pp. 153–169.

[31] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. Eur. Conf. Object-Oriented Program.*, 1995, pp. 77–101.

[32] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proc. 11th ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 1996, pp. 324–341.

[33] V. Sundaresan *et al.*, "Practical virtual method call resolution for java," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 264–280, 2000.

[34] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. Symp. Secur. Privacy*, 2012, pp. 93–104.

[35] T. Seals, "Cerberus enters the android malware rental scene," 2019. [Online]. Available: https://threatpost.com/cerberus-android-malware-rental/147280/

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

[37] K. Sun, "Google play apps drop anubis banking malware, use motion-based evasion tactics," 2019. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/google-play-apps-drop-anubis-banking-malware-use-motion-based-evasion-tactics/

[38] 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.

[39] C. Cadar *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
[40] 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.
[41] Q. Zeng *et al.*, "Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs," *IEEE Trans. Dependable Secure Comput.*, to be published, doi: 10.1109/TDSC.2019.2957787.
[42] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016, pp. 550–561.
[43] A. G. Jivani *et al.*, "A comparative study of stemming algorithms," *Int. J. Comp. Tech. Appl*, vol. 2, no. 6, pp. 1930–1938, 2011.
[44] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.
[45] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1025–1035.
[46] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1354–1365.
[47] G. Phipps, "Comparing observed bug and productivity rates for java and c++," *Softw. Pract. Exper.*, vol. 29, no. 4, pp. 345–358, 1999.
[48] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2012, pp. 1–12.

Jordan Samhi received the master's degree in computer and information systems security from the University of Lorraine, France, in 2019. He is currently a doctoral researcher with the University of Luxembourg, in 2021. His research interests include the security aspects of software engineering, with a particular focus on malware and vulnerability detection.

Alexandre Bartel is currently a full professor with the Computing Science Department, Umeå University. His research interests include and combine software testing, Android security and vulnerability analysis.