2016 IEEE Symposium on Security and Privacy

TriggerScope: Towards Detecting Logic Bombs in Android Applications

Yanick Fratantonio*, Antonio Bianchi*, William Robertson†, Engin Kirda†, Christopher Kruegel*, Giovanni Vigna*
*UC Santa Barbara
{yanick,antoniob,chris,vigna}@cs.ucsb.edu
†Northeastern University
{wkr,ek}@ccs.neu.edu

***Abstract*—Android is the most popular mobile platform today, and it is also the mobile operating system that is most heavily targeted by malware. Existing static analyses are effective in detecting the presence of most malicious code and unwanted information flows. However, certain types of malice are very difficult to capture explicitly by modeling permission sets, suspicious API calls, or unwanted information flows.**

One important type of such malice is *malicious application logic*, where a program (often subtly) modifies its outputs or performs actions that violate the expectations of the user. Malicious application logic is very hard to identify without a specification of the "normal," expected functionality of the application. We refer to malicious application logic that is executed, or triggered, only under certain (often narrow) circumstances as a *logic bomb*. This is a powerful mechanism that is commonly employed by targeted malware, often used as part of APTs and state-sponsored attacks: in fact, in this scenario, the malware is designed to target specific victims and to only activate under certain circumstances.

In this paper, we make a first step towards detecting logic bombs. In particular, we propose *trigger analysis*, a new static analysis technique that seeks to automatically identify triggers in Android applications. Our analysis combines symbolic execution, path predicate reconstruction and minimization, and interprocedural control-dependency analysis to enable the precise detection and characterization of triggers, and it overcomes several limitations of existing approaches.

We implemented a prototype of our analysis, called TRIGGERSCOPE, and we evaluated it over a large corpus of 9,582 benign apps from the Google Play Store and a set of trigger-based malware, including the recently-discovered HackingTeam's RCSAndroid advanced malware. Our system is capable of automatically identify several interesting time-, location-, and SMS-related triggers, is affected by a low false positive rate (0.38%), and it achieves 100% detection rate on the malware set. We also show how existing approaches, specifically when tasked to detect logic bombs, are affected by either a very high false positive rate or false negative rate. Finally, we discuss the logic bombs identified by our analysis, including two previously-unknown backdoors in benign apps.

I. INTRODUCTION

Android is currently the most popular mobile platform. 78% of all smartphones sold in Q1 2015 [39] were shipped with Android installed, and the Google Play Store now hosts more than two million applications [17]. Unfortunately, Android has also become the most widely-attacked mobile platform; according to a recent report, it is the target of 79% of known mobile malware instances [59].

App store providers invest significant resources to protect their users and keep their platforms clean from malicious apps. To prevent malicious apps from entering the market (and to detect malice in already-accepted applications), these providers typically use a combination of automated program analysis (e.g., Google Bouncer [41]) and manual app reviews. These automated approaches leverage static and/or dynamic code analysis techniques, and they aim to detect potentially-malicious behaviors – e.g., exfiltrating personal private information, stealing second-factor authentication codes, sending text messages to premium numbers, or creating mobile botnets. These techniques are similar in nature to the numerous approaches to detecting Android malware proposed in academia [29], [31], [40], [36], [64], [14], [18], [19], [32]. These approaches proved to be effective when detecting *traditional* malware [63], and recent reports show that the official Google Play app store is reasonably free from malicious applications [15].

Nonetheless, there are certain types of malice that are still very difficult to capture explicitly by modeling permission sets, suspicious API calls, or unwanted information flows (i.e., all those features used by existing analysis approaches). One important type of such malice is *malicious application logic*. We consider a program to contain malicious application logic when it (often subtly) modifies its outputs, providing results that violate the expectations that a user can reasonably have when interacting with this app. In particular, we refer to malicious application logic that is executed, or triggered, only under certain (often narrow) circumstances as a *logic bomb*.

As an example of a logic bomb, consider a navigation application (similarly to Google Maps) that is meant to assist a soldier in the battlefield when determining the shortest route to a given location. As a legitimate part of its intended behavior, this application would collect GPS-related information, send the information over the network to the application's back-end for processing, retrieve the results, and display to the user some helpful information (such as the route to follow). Assume further that this app contains a functionality that checks whether the current day is past a specific, hard-coded date: If the current day is indeed past this date, the app subtly queries the network back-end for a *long* route, and not for the *shortest* one as the user would expect. Thus, after the

DOI 10.1109/SP.2016.30

```
1 public void f() {
2     Date now = new Date();
3     Date target = new Date(12,22,2016);
4     // 1) retrieve GPS coordinates;
5     ...
6     if (now.after(target)) {
7         // 2) query network back-end
8         //    for a *long* route
9         g();
10    } else {
11        // 2) query network back-end
12        //    for the *shortest* route
13        //    (as expected)
14        h();
15    }
16    // 3) show computed route to user
17    ...
18 }
```

Figure 1: A possible implementation of a *logic bomb*.

hard-coded date, this application would provide misleading information, and the results could seriously affect the well-being of the user. Figure 1 shows a possible implementation of this behavior.

While traditional malware rarely implements this kind of stealthy behavior, these techniques are often used by targeted malware employed by APT actors when executing targeted, state-sponsored attacks. In fact, in this scenario, malware is designed to target specific victims and to only activate under certain circumstances. Unfortunately, targeted malware are becoming more prevalent. As a clear example, in July 2015 the HackingTeam security company was victim of a sophisticated attack [56], and all its internal resources and personal communications got publicly leaked: This attack led to the identification of RCSAndroid [60], one of the most sophisticated malware sample for Android ever discovered. This malware has the ability to leak the victim's private conversations, GPS location, and device tracking information, but it is also able to capture screenshots, collect information about online accounts, and capture real-time voice calls. However, these malicious behaviors are not manifested when the application starts. Instead, to increase its stealthiness, RCSAndroid waits for incoming SMS messages and checks whether these messages are sent from specific senders and contain specific commands. While this application was officially sold to law enforcements agencies and governments, the HackingTeam company has been accused by anti-surveillance campaigners of collaborating with governments with poor human rights records [57], and also in conducting targeted attacks against activists [58]. In particular, RCSAndroid's usage *in the wild* is documented by several (now public) internal communications [7], [5], [6].

Another scenario where trigger-based malware poses a real threat is related to the Android app store curated by the U.S. Department of Defense [26], which collects applications to assist officers and soldiers in the battlefield. The DoD marketplace features applications that are internally-developed but also many applications developed by government contractors and third parties, such as commercial entities. The current solution to finding malicious application logic is manual audit, often in combination with dynamic analysis. That is, to vet an application, a human analyst executes the program in an instrumented environment and studies its behavior under various inputs. In this model, the analyst's own judgment and the app's description serve as the guidelines to help determine whether the program functions as expected. Unfortunately, even this costly (both in terms of time and labor) manual process does not guarantee the identification of logic bombs, especially for those cases where the source code is not available (e.g., Google Maps).

Logic bombs are particularly insidious, since they can elude static analysis efforts and are hard to detect for human analysts, even when equipped with powerful dynamic analysis tools. In fact, consider the example in Figure 1. When examining this application, a static analysis system will not find any unusual permission or unwanted API calls, or any clearly-malicious action (in fact, invoking network-related APIs is perfectly legitimate for a navigation app), thereby bypassing traditional approaches such as [31], [18], [36], [64]. Also, all information flows (e.g., location-related source flows to a network sink) are expected, as they correspond to the description of the app in the store, rendering ineffective the detection capabilities provided by [19], [35]. Approaches based on dynamic analysis [41], [29], [40], [53], [49], [55] are ineffective as well: since the hardcoded date is set *in the future*, the time-related check will not be satisfied when testing the app, and the malicious functionality will thus not be executed. As another example, an app could run its malicious behavior only when the user is in a particular location. Unfortunately, these techniques are actively being used to bypass automatic and manual vetting systems [10], [1].

The key challenge is related to the fact that automatically detecting malicious application logic is very hard without taking into account the specific purpose and "normal" functionality of an application, and, hence, it is out of reach for most existing analysis tools. In fact, even those dynamic analysis tools designed to increase the code coverage (e.g., approaches based on multipath execution and/or dynamic symbolic execution [43], [22], [34], [8], [42], [61], [48]) would not have access to enough information to discern whether the just-executed functionality was malicious or not. At the very least, these tools would require a very fine-grained specification of the intended app behavior, something that is typically not available.

In this paper, we make a first step towards the automatic detection of logic bombs. Our work is based on the following key observation: an aspect that, at least in principle, is necessary for the implementation of a logic bomb is that the malicious behavior is triggered only under very specific circumstances. Thus, in this work we propose to detect logic bombs by precisely analyzing and characterizing the *checks* that guard a given behavior, and to give less importance to the behavior itself. To this end, we developed a new static analysis

technique, called *trigger analysis*, which combines traditional program analysis techniques with novel elements used for automatically and precisely identifying *triggers*. We (informally) define triggers as *suspicious* predicates (or checks) over program inputs that guard the execution of potentially-sensitive behavior, where a predicate is (intuitively) considered as suspicious if it is satisfied only under very specific conditions. In particular, we use static code analysis and symbolic execution to first identify checks that operate on sensitive input, and to then extract their precise semantics (i.e., which inputs are used, what operations were performed on these inputs, and what values are they compared against). We then use path predicate reconstruction, path predicate minimization, and predicate classification to identify *interesting* checks, and, as a last step, the analysis performs inter-procedural control-dependency analysis to determine whether a specific check guards *sensitive* operations.

We propose to use trigger analysis for the identification of logic bombs. However, of course, not every trigger (according to our definition) is part of a logic bomb. As a result, the fact that an app contains a trigger is typically not enough to outright convict an application as being malicious. However, we show in our experiments that a detection system that flags all applications that contain an interesting trigger as malicious, delivers excellent detection results for targeted malware, while raising a very small number of false positives on benign apps (outperforming existing malware detection systems that focus on opportunistic malware). Our system also returns a series of detailed information for each of the detected trigger, thus going beyond the mere identification of each of them. This greatly simplifies the work of a human analyst who has to make the decision whether a trigger is acceptable or malicious. Moreover, the output of our analysis can be also used as a starting point to craft inputs for a dynamic analysis system to exercise and vet the relevant behavior.

We have implemented our trigger analysis in a system called TRIGGERSCOPE. Our analysis operates directly on Dalvik bytecode, and it does not rely on access to source code. Our current prototype handles a number of different program inputs that have been traditionally used to activate malicious behavior: time, location, and the content (and sender) of text messages (SMS). We have extensively evaluated TRIGGERSCOPE over a large corpus of benign and malicious applications. Our benign dataset is constituted by 9,582 Android applications downloaded from the Google Play Store, while our malicious dataset is constituted by several malicious apps that were either developed by an independent DARPA Red Team organization (developed with the aim of resembling state-sponsored malware) or real-world malware samples containing logic bombs, including the HackingTeam's RCSAndroid application.

Our experiments demonstrate the ability of our system to precisely and efficiently detect triggered behavior in these applications. In particular, TRIGGERSCOPE was able to automatically identify several interesting triggers, including two previously-unknown backdoors in supposedly-benign apps, and a variety of logic bombs in the malicious samples. TRIGGERSCOPE's output also proved to be useful for constructing proof-of-concepts that exercise the relevant behaviors. To assess the precision of our tool, we performed manual analysis (on more than 100 applications) and we compared our results against the ground truth. TRIGGERSCOPE has a low false positive rate of 0.38%, and we did not encounter any false negative. Although we acknowledge that this evaluation does not definitely exclude the possibility of false negatives in the benign apps (see Section VI for a discussion about the limitations of this work), we believe our results are an encouraging step towards the detection of trigger-based behavior in Android applications.

As the second part of our evaluation, we considered several state-of-the-art Android malware detection tools, each of which relies on a different approach. In particular, we considered Kirin [31], which relies on permission-based signatures, DroidAPIMiner [14], which relies on machine learning, and FlowDroid [19], which relies on taint analysis. Our experiments show that all these existing tools are not suitable for the detection of logic bombs, as they either have a very high false negative rate (78.57%) or a very high false positive rate (69.23%). We show that TRIGGERSCOPE significantly outperforms them.

To summarize, this paper makes the following contributions:

- We make a first step towards the automatic detection of logic bombs in Android applications. To this end, we introduce *trigger analysis*, a static program analysis technique that discovers hidden triggers. Our analysis combines both existing and novel analysis techniques: symbolic execution (§III-A), block predicate extraction (§III-B), path predicate reconstruction and minimization (§III-C), predicate classification (§III-D), and inter-procedural control-dependency analysis (§III-E).
- We developed a prototype, called TRIGGERSCOPE, and we evaluated it over a large corpus of benign and malicious Android applications. Our experiments show that TRIGGERSCOPE is able to efficiently and effectively identify previously-unknown, interesting triggers, including two backdoors in benign apps and a variety of logic bombs in the malicious samples. Our evaluation also shows that TRIGGERSCOPE has a very low false positive rate, and it outperforms several other state-of-the-art analysis tools when detecting logic bombs.
- We show how TRIGGERSCOPE can effectively assist a human analyst who aims to identify hidden *logic bombs* in Android apps. In fact, TRIGGERSCOPE's analysis output includes rich details about the detected triggers, and enables the quick verification of its findings through proof-of-concepts that exercise the relevant behaviors. We also empirically found that triggers are relatively rare in benign apps, and their presence can therefore be used a strong signal that motivates further scrutiny.

II. SYSTEM OVERVIEW

In the previous section, we informally introduced the notion of *triggers*. In the following paragraphs, we first sharpen that

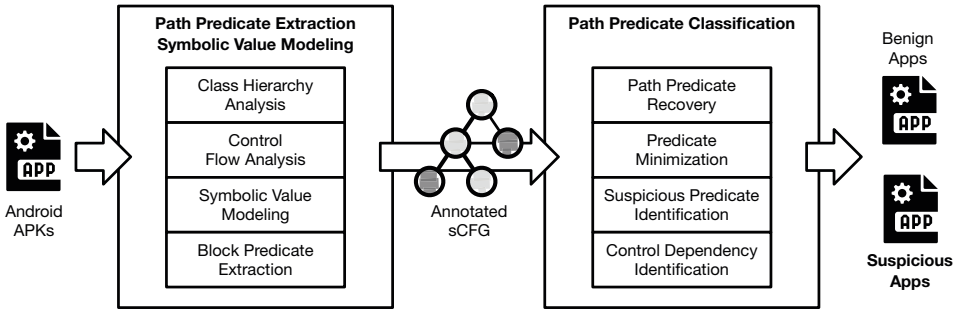

Figure 2: Overview of the components that comprise our trigger analysis. In the first phase, Android APKs are disassembled, and an Android-specific forward symbolic execution is performed on the Dalvik bytecode to recover an sCFG annotated with block predicates and abstract program states at all program points. In the second phase, full path predicates are recovered and checked whether they represent potential triggers for malicious behavior.

definition to provide the reader with a better understanding of our threat model. Then, we describe at a high level how our system can find triggers in Android apps.

A. Trigger – A Definition

Before providing a definition of triggers, we first introduce some relevant terminology. We define a *predicate* as an abstract formula that represents a condition in a program: a condition is introduced by a branch (such as an *if* statement) and ensures that some program code is executed only when the abstract formula (i.e., the predicate) evaluates to true. Moreover, a predicate is said to be *suspicious* if, intuitively, it represents a condition that is satisfied only under very specific, narrow circumstances (Section III-D provides a more concrete definition). We then define a *functionality* as a set of basic blocks in a program. A functionality is said to be *sensitive* if at least one of its basic blocks performs, directly or indirectly (i.e., through a method call), a sensitive operation. The definition of *sensitivity* can be specified through a user-defined policy (Section III-E describes the concrete policy we used for this paper). We now define a *trigger* as a suspicious predicate that (directly or indirectly) controls the execution of a sensitive functionality.

More formally, a *trigger* is a predicate $\mathtt{p}$ such that the following property holds: $\mathtt{isSuspicious}(\mathtt{p}) \wedge \exists\ \mathtt{F} : (\mathtt{isSensitive}(\mathtt{F}) \wedge \mathtt{controlsExec}(\mathtt{p}, \mathtt{F}))$. The $\mathtt{isSuspicious}(\mathtt{p})$ and $\mathtt{isSensitive}(\mathtt{F})$ properties are satisfied if the predicate $\mathtt{p}$ is suspicious and the functionality $\mathtt{F}$ is sensitive, respectively. The $\mathtt{controlsExec}(\mathtt{p}, \mathtt{F})$ property is satisfied if either one of the following two properties hold: 1) $\mathtt{p}$ *directly controls* the execution of $\mathtt{F}$;[1] 2) $\exists\ \mathtt{F}'$ such that $\mathtt{p}$ *directly controls* the execution of $\mathtt{F}' \wedge$ "$\mathtt{F}'$ (intra- or inter-procedurally) alters the value of a field (or object) that is part of a predicate $\mathtt{p}'$" $\wedge\ \mathtt{controlsExec}(\mathtt{p}', \mathtt{F})$. For the interested reader, Figure 7 (in Appendix B) reports the implementation (in pseudocode) of the function that determines whether a given predicate matches our definition of trigger.

Throughout this paper, we will discuss how TRIGGERSCOPE's analysis steps play a key role in effectively detecting triggers: in particular, we will show that `isSuspicious` heavily relies on the information extracted by the symbolic execution and predicate classification, while `controlsExec` relies on the path reconstruction and minimization technique, and on the control-dependency analysis.

B. Analysis Overview

At a high level, our trigger analysis for Android applications proceeds in two phases, an overview of which is depicted in Figure 2. In the first phase, Android APKs are unpacked and subjected to forward static symbolic execution. For this, we leverage a flow-, context-, and path-sensitive analysis that also takes into consideration the Android application lifecycle and interactions between Android application components. This phase produces an annotated super control-flow graph (sCFG), which consists of the inter-procedural CFG superimposed on the intra-procedural CFGs for each method. The annotations store all possible values (upper and lower bounds) for local and field variables in the program, as well as detailed information about how the objects relevant to our analysis are created and modified.

The second phase takes this annotated graph as input, with the goal of identifying all *triggers* contained in the program. The first step of this phase is to recover the intra-procedural *path predicates* associated with each basic block. A path predicate for a basic block $\mathtt{b}$ is a predicate $\mathtt{p}$ such that if 1) the execution reaches the entry block of the method containing $\mathtt{b}$, and 2) $\mathtt{p}$ is satisfied, then $\mathtt{b}$ will be necessarily executed. These path predicates give us information about which conditions in the program control the execution of which blocks. As the next step, the analysis identifies all *suspicious* path predicates in the program (this is possible thanks to the information extracted during the symbolic execution step), and, for each of them, it checks whether the predicate guards the execution of a *sensitive* functionality: these predicates are exactly the ones that match our definition of *trigger*.

[1] In this paper, "*directly controls*" indicates that $\mathtt{p}$ is part of the intra-procedural path predicate that controls the execution of $\mathtt{F}$. All details are explained in Section III-E

```
1 .method public f()V
2   // Date now = new Date();
3   new-instance v0, Ljava/util/Date;
4   invoke-direct {v0}, \
5     Ljava/util/Date;-><init>()V
6
7   // Date target = new Date(12,22,2016);
8   new-instance v1, Ljava/util/Date;
9   const/16 v2, 0xc
10  const/16 v3, 0x16
11  const/16 v4, 0x7e0
12  invoke-direct {v1, v2, v3, v4}, \
13    Ljava/util/Date;-><init>(III)V
14
15  // if (now.after(target)) {...}
16  invoke-virtual {v0, v1}, \
17    Ljava/util/Date;-> \
18      after(Ljava/util/Date;)Z
19  move-result v2
20
21  // suspicious check!
22  if-eqz v2, :cond_0
23
24  // g();
25  invoke-virtual {p0}, LApp;->g()V
26  goto :goto_0
27
28  :cond_0
29  // h();
30  invoke-virtual {p0}, LApp;->h()V
31
32  :goto_0
33  return-void
34
35 .end method
```

Figure 3: This figure shows the Dalvik bytecode representation of the `f` function presented in Figure 1. The Java-equivalent of each set of instructions is reported in the comments. This example clearly shows how the semantics of the suspicious check is lost. In fact, the check is translated into a simple `if-eqz` bytecode instruction (line 22): both the type of operation and its arguments are lost. TRIGGERSCOPE uses symbolic execution to reconstruct the semantics of these checks, to then perform a classification step.

In the remainder of this section, we provide an overview of the main analysis steps, and discuss their role in the entire analysis process.

Symbolic Execution. One of the key aspects of our analysis is the capability to *classify* predicates (or checks). The main challenge in doing so is related to the fact that the *semantics* of each check is lost during the translation of the program from Java source code to Dalvik bytecode. As an explanatory example, consider again the snippet of code in Figure 1. As we already discussed, for a human analyst, it is straightforward to recognize that the check contained in function `f` is *suspicious*. However, at the bytecode level, the clearly-suspicious check in the Java snippet (line 6) is translated into a `if-eqz` bytecode instruction, which simply checks that the content of a register is different from zero. Thus, the semantics of the check is not easily-accessible anymore and must be reconstructed. Figure 3 shows the Dalvik bytecode corresponding to the example in Figure 1.

To overcome this limitation, our approach relies on symbolic execution and it precisely models several Java and Android APIs. This allows our approach to annotate each object referenced in a check, with precise information about its type, (symbolic) value, and the operations that influence it. As we discuss in details in Section III-A, these annotations allow the analysis to generate *expression trees* that contain all the necessary information to reconstruct the semantics of the check and to consequently classify it.

Block Predicate Extraction. As we mentioned, one of the main steps of our analysis is to reconstruct the path predicates associated with each basic block of the program. To do so, the analysis first extracts *simple* block predicates – i.e., symbolic formulas over the abstract program state that must be satisfied in order for a basic block to be executed. In particular, during the symbolic execution step, the system annotates the CFG with information about the low-level conditions that need to be satisfied in order to reach each block, assuming that the execution already reached one of its predecessors. Moreover, these conditions are also annotated with information about the semantics of the objects involved in the check. This step is discussed in detail in Section III-B.

Path Predicate Recovery and Minimization. In the next phase, the analysis combines together the simple block predicates, to then recover the path predicates for each basic block in the program. To this end, the analysis performs a backwards traversal of the CFG, it recovers the full path predicates, and it then minimizes them to remove redundant terms, which would otherwise introduce false dependencies. The details of this step are discussed in Section III-C.

Predicate Classification. While the aforementioned techniques greatly reduce the candidate set of path predicates that must be considered, this alone is not enough to precisely identify suspicious predicates. As an example, consider a game that implements a recurring check that triggers an action every few seconds. Although it depends on time, this behavior is perfectly legitimate. For this reason, our analysis considers multiple characteristics of a predicate in order to classify it. This not only includes whether the predicate involves values labeled as originating from a potential trigger input, but also the type of the comparison performed. Note that this is technically possible only because the system has access to the information extracted during the symbolic execution step. The full details of this step are presented in Section III-D.

Control-Dependency Analysis. As a final step, our system checks whether a suspicious predicate *guards* any sensitive operations. In particular, the system recursively checks, for each block guarded by a suspicious predicate, whether this block (intra- or inter-procedurally) invokes a sensitive method, or whether it modifies a field or an object that are later involved in a predicate that, in turn, *guards* the execution of a sensitive operation. This step allows us to detect explicit as well as *implicit* control dependencies, and it significantly improves the precision over systems that simply look for any kind of checks

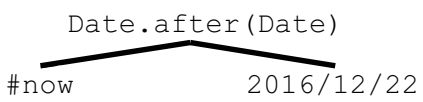

Figure 4: Example of an expression tree.

against sensitive values, in terms of both false positives and false negatives. The details about this step are provided in Section III-E.

III. Analysis Steps

While the previous section provided a high-level overview of the analysis steps, in this section we elaborate upon the details.

A. Symbolic Execution

The analysis begins by first unpacking the Android APK and extracting the DEX file that contains the application's Dalvik bytecode, the encoded application manifest, and encoded resources such as string values and GUI layouts. The bytecode is then lifted into a custom intermediate representation (IR) that all our analysis passes operate on. The analysis then performs a class hierarchy analysis and control flow analysis over the IR to construct the intra- and inter-procedural control-flow graphs. After these preliminary steps, the application's bytecode is subjected to forward static symbolic execution, applying a flow-, path-, and context-sensitive analysis, where the particular context used ranges from full insensitivity to 2Type1Heap object sensitivity [51], which is known to provide a good trade-off between precision and performance when performing symbolic execution on object-based programs.

Android Framework Modeling. A notable feature of our analysis that bears mention is its awareness of *i*) the Android application lifecycle, *ii*) control flows that traverse the Android application framework due to the pervasive use of asynchronous callbacks used in Android applications, and *iii*) inter-component communication using the Android intent framework. The precise modeling of these aspects has been widely studied in the literature and, for our design and implementation, we mainly reused ideas from previous works. In particular, we follow the approach described in FlowDroid to model Android application components' lifecycle [19]; we integrate EdgeMiner's results to model the control flow transfers through the Android framework [24]; and we reused ideas presented in Epicc [45] to precisely model inter-component communications among the applications components.

Symbolic Values Modeling. Our symbolic execution engine models the sets of possible values that local and field locations can contain. In particular, the analysis focuses on integer, string, time, location, and SMS-related objects, and it records operations performed over concrete and symbolic values in these classes. For example, the analysis faithfully models symbolic string values produced and manipulated by many important classes (e.g., `String`, `StringBuffer`, `StringBuilder`, and related classes) and their respective methods (e.g., `append`, `substring`, and similar APIs). Moreover, our system models symbolic integer values resulting from APIs such as `equals`, `startswith`, and `contains`, which are particularly important when detecting suspicious checks on String-based objects (such as the body and the sender retrieved from the `SmsMessage` object). The accurate modeling of `String` objects is also useful to precisely model `Intent`s and `Bundle`s objects (in essence, a key-value store).

Similarly, our prototype also models time-related objects. For example, time values can be introduced by construction from a constant, in which case the analysis computes the corresponding concrete value. Values representing the *current time* are also specially handled by modeling the Android APIs known to return such values (which are represented by the `#now` tag in the examples mentioned in this paper). Our system also annotates time-related objects with special tags to encode which *time component* is stored in a given object. For example, an application can invoke the `Date.getMonth()` method to access the *month* component of a given time object, or the `Date.getSeconds()` method to access the *seconds*. In this case, our analysis annotates the return value of these methods invocations with the special tags `#now/#month` and `#now/#seconds`. All other components are similarly modeled. This information can be useful, for example, to determine how *narrow* is the condition represented by a given time-related check.

Note also that it is not sufficient to simply record time values representing the current time as a singular "now" value, as the notion of "current time" monotonically increases during program execution. Therefore, in addition to recording that a time object corresponds to the current time at the point where it is returned from the Android framework, the analysis additionally annotates such value with an integer identifier that is incremented for each new current time value observed during symbolic execution. This additional information is used to reconstruct the semantics of a given check (e.g., hard-coded vs. recurring check) more precisely.

In addition, our analysis also models symbolic location- and SMS-related values and operations. Sources of location-related values include invocation of Android APIs related to the GPS and cellular radio devices, operations on `Location` objects, and transformations on raw integer and double values extracted from `Location` objects. Our analysis also keeps track whether a given `Location` object represents the *current* location (in which case the value is annotated with the special `#here` tag), and, when an application accesses a specific location component, such as longitude and latitude, our analysis encodes this information with the special tags `#longitude` and `#latitude`.

Finally, sources of SMS-related values are modeled similarly, and include operations performed on `SmsMessage` objects, such as `createFromPdu`, `getMessageBody`, and `getOriginatingAddress`, to which our analysis associates, respectively, the tags `#sms`, `#sms/#body`, and `#sms/#sender`.

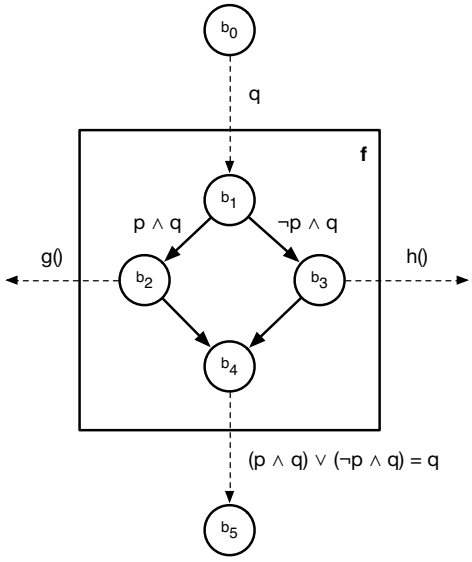

Figure 5: Path predicate reconstruction from block predicates.

Expression Trees. During the symbolic execution step, each symbolic object is also annotated with an *expression tree* that encodes the operations that influence its value. As an example, for the code snippet in Figure 1, our system would annotate the object involved in the check with the expression tree depicted in Figure 4, whose text representation is "`(#now after Date(2016/12/22))`." As we will show, these annotations contain all the necessary information to reconstruct the semantics of checks that operate on these inputs (and later classify them). Of course, to avoid performance issues, we keep track of precise information only for objects that are *relevant* to our analysis.

B. Block Predicate Extraction

We already mentioned how, during the symbolic execution step, the sCFG is also annotated with *simple* block predicates. To better explain this step, consider, once again, the program snippet in Figure 1 and its schematic representation in Figure 5. The analyzer would first recover the block predicate p as a path condition introduced by b_1, and it would annotate the $b_1 \rightarrow b_2$ edge (where b_2 is the block that contains the invocation of method `g`) with predicate p, representing that, once the execution reaches b_1, b_2 is executed if and only if p holds. For this example, our system would determine that p is a comparison of the value of an object x against the constant zero. The expression tree of x (i.e., Figure 4) is then retrieved, and the predicate p is determined to be "`(!= (#now after Date(2016/12/22)) 0)`." Similarly, the $b_1 \rightarrow b_3$ edge (where b_3 is the block that contains the invocation of method `h`) would be annotated with the predicate $\neg p$.

C. Path Predicate Recovery and Minimization

In the following step, *simple* block predicates are combined together to recover, for each basic block, the full intra-procedural path predicate. Given the block predicates for each basic block, path predicate recovery is conceptually straightforward. For each block, the analysis performs a backwards traversal of the enclosing method's CFG and builds a complex Boolean formula that represents all paths from the current block to the entry point of the method. Predicates for sequences of basic blocks are combined using conjunction (i.e., logical AND), while blocks with multiple predecessors (due to branch joins) take the disjunction (i.e., logical OR) of the corresponding incoming path predicates. These predicates not only capture the control-dependency relation between blocks, but also precisely encode the condition that must be satisfied for a block to be executed.

Note that, without any further steps, this simple path predicate reconstruction algorithm might produce path predicates that contain redundant terms. These terms could, if not removed, result in a large number of blocks with false dependencies on values derived from trigger inputs. For instance, if a basic block is only executed when a certain condition evaluates to true over a value derived from a trigger input, then the path predicate recovery algorithm will correctly identify that the basic block depends on that trigger input. But, it will also erroneously state that successor blocks – including blocks executed after a join with the *else* path – depend on the trigger input.

To illustrate, consider once again the graph shown in Figure 5, extracted from the trigger example introduced in Figure 1. After the analysis has computed block predicates over the entire sCFG, the path predicate from block b_0 to b_2 – where the incoming path predicate to `f` is q – is represented by the conjunction $p \wedge q$. Since p involves symbolic time values, blocks b_2 and b_3 have a control dependency on a time-based input. However, the path predicate associated with the join of the two branches, represented by b_4, would result in the following expression: $(p \wedge q) \vee (\neg p \wedge q)$. This will also be the path predicate associated with the exit from `f`, represented by b_5. Note how this path predicate has a dependency on a time-based input (through the predicate p), even if the execution of the basic block b_5 is clearly not guarded by any time-related check.

For this reason, after path predicates are recovered for each basic block, the analysis *minimizes* each path predicate. This is accomplished by recursively simplifying the full formulas for each basic block (using standard Boolean laws such as the distributive law) until no further simplifications can be performed. To conclude our example, the minimization of the path predicate associated with b_5 would result in the expression $(p \wedge q) \vee (\neg p \wedge q) = (p \vee \neg p) \wedge q = q$, which, as expected, does not contain a spurious dependency on time input. We note that minimization of Boolean formulas is NP-hard in the general case. However, we found this technique to be fast in practice in our system. Finally, to compute path predicates for those cases that include loops, we make use of the techniques described in [33].

D. Predicate Classification

Once the minimized path predicates have been recovered, the analysis proceeds to classify these path predicates according to the semantics of the comparison performed on potential trigger inputs. To this end, the analysis checks the type of comparison performed (e.g., $=, <, \leqslant, >, \geqslant$) as well as the types of the operands used in the comparison (e.g., whether a value is a constant, purely symbolic, or a special symbolic value such as the current timestamp, the current location, or an incoming SMS).

Intuitively, a check (predicate) is considered as suspicious if it encodes a *narrow* condition. In fact, these are the types of checks used by malicious applications that, for example, are programmed to mount an attack only when the user is at a very specific location (e.g., a soldier in a war zone).

In our current implementation, a check is flagged as *suspicious* according to the following policy: If a predicate is semantically equivalent to one (or more) ordering comparison(s) between a current time value and a constant, then the predicate is labeled as suspicious. A similar criterion is used for location values. That is, if a predicate is equivalent to a bounds check on symbolic values derived from an Android location object, then the predicate is labeled as suspicious. We apply a similar policy for SMS objects as well. Our current implementation flags as suspicious any functionality that attempts to match hard-coded patterns against the body (or sender) of an incoming SMS.

Clearly, this is not the only possible valid definition of *suspiciousness*. In fact, in certain scenarios, recurring checks that, for example, trigger the execution of a given functionality only once every month could be considered suspicious as well. We believe that our observation that predicates encoding *narrow* conditions deserve special attention is generic, and it is independent from how such narrow checks are implemented. Moreover, our analysis framework extracts all the relevant information about path predicates that makes the detection of these different kinds of checks possible.

Additionally, we implemented a few post-filter steps. These are useful to filter out checks that match our definition of *suspiciousness* but that are clearly benign. For example, we empirically found that several applications check the value of a field (that might contain a timestamp) against the constant 0 or -1: in effect, the application is just performing a "Is this field already set with a valid timestamp?" type of check. Similarly, for the location domain, we found cases where location objects were compared against the constant value 0 – in effect, the application performs a null reference check. We also found that a similar kind of *sanity check* is performed on strings representing the body of an SMS. In particular, some applications check whether the length of the SMS's body and sender is greater than 0. As the 0 and -1 values do not represent a valid (or interesting) absolute time or location, and as all valid SMS have a non-empty body, we consider these checks to be benign, and, therefore, our analysis does not flag them as suspicious. We note that these post-filter steps are *safe* (i.e., they do not filter potentially-suspicious checks out) and that, as we will discuss in Section V, they only affect a very limited number of applications.

E. Control-Dependency Analysis

The final step of our analysis consists in determining whether a control dependency exists between predicates identified as suspicious in the prior step and basic blocks that contain invocations of sensitive Android framework methods. This is accomplished using a forward traversal of the sCFG, starting from each basic block that is guarded by a suspicious path predicate. For each of these blocks, if that block invokes a sensitive API method, then the path leading to it is labeled as a *suspicious triggered behavior*.

Furthermore, our control-dependency analysis is interprocedural, and it also propagates suspicious predicates through field locations or object definitions. That is, if a block guarded by a suspicious predicate updates a field or sets an object, subsequent blocks (potentially contained in different methods) that are guarded by a check on that field (or object) are added to the set of suspicious blocks to check. A similar propagation is performed over flows through the Android framework due to callback registrations and invocations, as well as over intent-based inter-component flows. This allows us to detect (implicit) control dependencies that would *evade* simpler static analysis approaches.

Returning to the time-based trigger example in Figure 1 and its schematic representation in Figure 5, our system would consider b_2 and b_3 as blocks to be checked for sensitive operations. If the analysis finds that method `g` (or `h`) can directly or indirectly invoke a sensitive operation, then that behavior would be classified by our analysis as a suspicious triggered behavior.

For this work, we compiled a list of potentially-sensitive operations by considering all Android APIs protected by a permission (we used, as a starting point, the results of PScout [20] and SuSi [47]), and by augmenting it with operations that involve the filesystem (this list can be simply modified through a configuration file). As the reader can notice, our definition of *sensitive* is very conservative: in fact, the vast majority of the operations considered as such are not even security-relevant. As shown in the evaluation section, even with this conservative definition of *sensitiveness*, our approach is affected by a very low false positives rate, the reason being that we focus on *characterizing the check that guards a given behavior, and not on the guarded behavior per se*. It is also worth noting that if existing approaches would adopt a similar definition of *sensitiveness*, they would all be affected by an unacceptable rate of false positives.

IV. Implementation Details

We implemented our approach in a tool called TriggerScope. The initial parsing of the bytecode uses the `libdex` library from the AOSP project [3]. Then, the bytecode is lifted into an intermediate representation (IR) suitable for performing symbolic execution. Our symbolic execution engine models the semantics of all individual Dalvik virtual machine

instructions over an abstract representation of program states, including the current program counter, abstract store, and local environment. Thus, TRIGGERSCOPE is a pure static symbolic execution engine for Dalvik bytecode, and it is completely independent from the Android framework itself. TRIGGERSCOPE is mostly written in C++, and consists of 18.6K SLOC.

Our prototype faithfully models all Dalvik bytecode instructions and propagates the taints (and the information about the objects) accordingly. Moreover, as we described, our system precisely models a number of relevant APIs. For these APIs, the taints are propagated according to their model. For all other APIs for which a model is not available, our system applies a default tainting policy, which propagates all taints on method arguments to the receiver object and to the returned value. This conservative approach helps in case of a lookup in a data structure like a map. Another problematic case is posed when a tainted object is written to a file. To conservatively handle this and similar cases, one would need to keep track of, for example, file paths. When it is not possible to do so, our current prototype errs on the side of false positives: if a tainted value is written to the filesystem and no information on the file path is available, the analysis will propagate such taint upon subsequent file reads.

From a conceptual point of view, these design choices would lead to more imprecision, which, as everything is handled conservatively, would directly translate to over-tainting and a high number of false positives. However, we empirically found that this is not the case in practice. In fact, as we show in the evaluation section, our analysis is quite precise in terms of detecting suspicious triggered behaviors. Intuitively, this is possible as time-, location-, and SMS-related functionality is often well-isolated and self-contained, and even such aggressive policies do not affect the overall precision of our analysis.

Another important aspect is related to keeping track of constant values. In fact, determining whether an object (independently from its representation) encodes a constant value is critical for our classification steps. Unfortunately, in the general case it is not possible to record the exact value for each constant. For this reason, when such information is not available, our implementation falls back to a basic taint propagation system, which can label a symbolic object as constant.

Developing a symbolic execution engine for Dalvik bytecode is not novel, and for the design and implementation of our prototype, we reused many of the ideas from existing static analysis tools [13], [12], [19], [23], [36]. However, none of these frameworks has been designed to perform symbolic execution directly on Dalvik bytecode and, therefore, we were not able to reuse their implementation. Our approach also relies on detailed CFG annotations (described throughout the paper) as well as on precise models of symbolic values, which other approaches do not provide.

V. EVALUATION

In this section, we evaluate TRIGGERSCOPE's ability to effectively and efficiently identify suspicious time-, location-, and SMS-based logic bombs. We first describe the data sets we used for the experiments. Next, we evaluate the performance aspect, we discuss the results of our trigger analysis, and we perform an evaluation of the accuracy of our analysis. We then compare our system's accuracy against the accuracy achieved by three existing state-of-the-art tools, namely Kirin [31], DroidAPIMiner [14], and FlowDroid [19], and we discuss why they are not adequate for the automatic detection of logic bombs. We conclude by discussing case studies of a number of interesting triggers found by TRIGGERSCOPE, including two previously-unknown backdoors in benign apps, and a variety of logic bombs in the malicious samples.

A. Data Sets

For this work, we built a dataset of applications that include both benign and malware samples relevant to our analysis. This section describes how we obtained these two datasets.

Benign Applications. Since the current prototype focuses on the detection of suspicious time-, location-, and SMS-based triggers, we selected applications from the Google Play Store that were known to use time-, location-, and SMS-related APIs. To this end, we built three different sets: we selected 5,803 applications that are known to make use of time-related APIs, 4,135 applications that invoke location-related APIs, and 1,400 applications that have the capability to receive SMS. In total, these three sets contain 9,582 unique applications (some make use of a combination of time-, location-, and SMS-related functionality). These applications were selected among a total of 21,747 (free) applications obtained from a previous crawl of the market. These applications were crawled without focusing on any specific selection criteria, and they span various *app categories*, include well-known frameworks, and contain, on average, hundreds of methods.

We built the first two data sets by statically checking all apps from the crawl for the use of a predefined set of Android time- and location-related APIs. The third data set included all apps that require the `android.permission.RECEIVE_SMS` Android permission, which is necessary for an app to *receive* and process incoming SMS messages.

Malicious Applications. Our dataset of malicious applications is constituted by 14 applications that are relevant to our analysis. These applications have been taken from several sources. First, we considered 11 applications that were prepared by a Red Team organization (an external, independent government contractor) as part of a DARPA engagement related to the analysis and identification of malicious Android applications. These applications have been developed with the idea of resembling state-sponsored malware and they are intentionally designed to be as stealthy as possible, with the aim of circumventing all existing automated malware analysis tools. Additionally, we considered real-world malware samples that contained a time-based logic bomb [54], SMS-based C&C server [27], and the RCSAndroid malware sample written by the HackingTeam security company [60].

Domain	# Apps	# Apps With Checks	# Apps With Suspicious Checks	# Apps With Suspicious Triggered Behavior	# Apps After Post-Filter Steps
Time	4,950	1,026	302	30	10
Location	3,430	137	71	23	8
SMS	1,138	223	89	64	17

Table I: This table summarizes how the different steps of our analysis are able to drastically reduce the number of false positives when detecting triggered malware in a large set of benign applications obtained from the Google Play store.

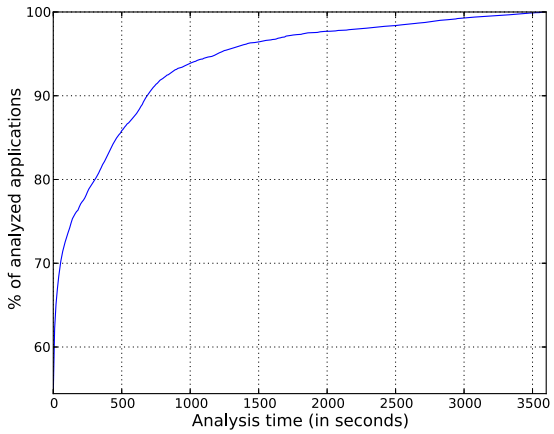

Figure 6: CDF of the elapsed analysis time over the three test sets of applications that use Android time, location, and SMS APIs. In this experiment, 90% of the applications tested were successfully analyzed for suspicious triggers in under 750 seconds.

B. Performance

Our experiments suggest that the performance of the TRIGGERSCOPE prototype is good enough to be able to scale its analysis to thousands of real-world Android applications. In particular, we analyzed the three data sets of applications from the Google Play Store and set a timeout of one hour for each instance. The tool was able to successfully analyze 4,950 out of a total of 5,803 time-related applications, 3,430 out of a total of 4,135 location-related applications, and 1,138 out of a total of 1,400 SMS-related applications (9,313 unique samples in total). The analysis of the remaining applications did not complete before the timeout was reached. Figure 6 shows a cumulative distribution of elapsed analysis times (for those applications that were successfully analyzed), indicating how many applications were analyzed within a given number of seconds. In particular, we observe that 90% of the applications we tested were completely analyzed for triggers in under 750 seconds. Moreover, on average, the analysis of each of these applications required 219.21 seconds. This suggests that performing trigger analysis over large sets of Android applications, perhaps centrally at an app store, is feasible, especially since the analysis can easily be horizontally scaled.

C. Trigger Analysis Results

In the 4,950 time-related applications, our tool identified a total of 12,465 basic blocks whose execution is guarded by a time-related constraint, contained in 1,026 different applications (§III-B). After reconstructing the minimized path predicates that guard each block (§III-C), TRIGGERSCOPE performs a classification step for each of them (§III-D). In this experiment, TRIGGERSCOPE detected 302 applications containing at least one *suspicious* time-related predicate. Then, TRIGGERSCOPE performs control-dependency analysis (§III-E) to determine whether these predicates guard the execution of any *sensitive* operations. This analysis step reduced the number of applications to be manually inspected to 30 samples, a number that was further lowered to 10 by using the post-filter steps (§III-D).

For what concerns the 3,430 applications containing location-related APIs, TRIGGERSCOPE identified a total of 137 applications that cumulatively contain 869 location-related predicates. TRIGGERSCOPE's analysis steps were then able to progressively reduce this set of applications to 71 (identification of suspicious checks), 23 (identification of sensitive operations), and 8 (post-filter steps). Similarly, for the SMS domain, the tool identified a cumulative total of 1,087 SMS-related predicates in 223 applications (out of 1,138). Of these, the analysis steps reduced this set to, respectively, 89 (identification of suspicious checks), 64 (identification of sensitive operations), and 17 (post-filter steps) applications.

In total, TRIGGERSCOPE flagged 35 applications as *suspicious* among the apps obtained from the Google Play Store. Table I provides a summary of the results and underlines how all the different analysis steps that constitute our *trigger analysis* technique are relevant to reduce the number of *flagged* applications.

For what concern the malicious applications, TRIGGERSCOPE was able to analyze and detect a trigger in all of them. We discuss several insights in Sections V-F and V-G.

D. Accuracy Evaluation

We now discuss the precision of our analysis and we study how the different analysis steps contribute to the end result. We first computed the false positive ratio (FPR), which is computed as the number of *false alarms* over the number of the considered benign samples. We opted to evaluate our system by using this metric (instead of others) since we believe it answers the most relevant question when such systems are deployed in real-world scenarios (where the vast majority of the samples are benign): "Given a dataset of benign

applications, how many false alerts does a system raise?" As discussed in the previous section, TRIGGERSCOPE detected 35 benign applications (among the 9,313 samples successfully analyzed) that contained at least one time-, location-, or SMS-based trigger matching our definition of suspiciousness.

To evaluate the accuracy of our system, we manually inspected all these applications, by using IDA Pro [37] to disassemble the apps and, when possible, by using `dex2jar` [4], JD-Gui [28], and JEB [52] to decompile them. We were able to manually verify that TRIGGERSCOPE correctly and precisely identified at least one interesting/suspicious trigger in each of these applications. While most of these behaviors appear to be legitimate, we identified two applications that contain a backdoor-like functionality (these and other cases are discussed in-depth in Section V-F). That being said, for the sake of this evaluation, we consider all these 35 applications as false positives (even if, depending from the context, the two backdoors might be considered as true positive). Thus, TRIGGERSCOPE has a false positive rate of 0.38%. Note also that the false positive rate is even lower (0.16%) when considering all the applications in our initial dataset (and not only the ones that have access to input we check for triggers).

To assess whether TRIGGERSCOPE is affected by false negatives, we manually inspected two sets of applications. First, we inspected all 82 applications that our post-filter steps discarded. In all cases, we were able to establish that the discarded instances were not interesting. Second, we manually inspected a random subset of 20 applications for which our analysis did not identify any suspicious check. We spent about 10 minutes per application, and, once again, we did not find any false negatives. We acknowledge that this evaluation does not definitely exclude the possibility of false negatives (see Section VI for a discussion about the limitations of this work), but we believe our results are an encouraging step towards the detection of trigger-based behavior in Android applications.

Table II provides a summary of the accuracy evaluation of our analysis. The table also shows how the accuracy changes when only a subset of TRIGGERSCOPE's analysis steps are used, clearly showing how all these steps actively contribute to improve the overall accuracy of our approach.

E. Comparison with Existing Approaches

As part of our evaluation, we studied whether existing malware analysis tools (which proved to be very effective when detecting *traditional* malware) are suitable to detect targeted malware (that leverages logic bombs) in Android applications. We selected the most representative works in the area of Android malware analysis: Kirin [31], DroidAPIMiner [14], FlowDroid [19], AppContext [62], DroidRanger [64], Drebin [18], and Apposcopy [32]. We attempted to obtain or reproduce all of them, but we were able to do so only for the first three tools: Kirin, DroidAPIMiner, and FlowDroid. DroidRanger, Drebin, and Apposcopy are currently not open source. AppContext, instead, has recently been released as open source. However, we encountered several difficulties when attempting to use it, and we are currently seeking help from the authors.

We believe these tools to be representative of very different approaches to detect malicious and unwanted behavior in Android apps. In fact, Kirin relies on permission analysis; DroidAPIMiner applies machine learning based on the Android APIs invoked by a given application; and FlowDroid applies taint analysis to identify sensitive data flows (such as privacy leaks). We acknowledge that FlowDroid is not meant to be used for malware analysis, and this obviously affects its performance. However, we included it in our evaluation because we believe it is interesting to also consider an approach based on taint analysis, since it constitutes one of the possibilities for detecting malicious/unwanted functionality.

The remainder of this section describes the details and the results of our experiment.

Kirin. Kirin is an analysis tool that performs lightweight malware detection by flagging an application as suspicious according to a set of rules based on the requested permissions. Kirin is open source [30], and we were easily able to reproduce the analysis. Kirin relies on the specification of a rule set: for this evaluation we considered the rules described in the paper [31] and included in the source code. We used this tool to analyze the applications in our dataset. Table III shows a summary of the results. Kirin has a relatively low false positive rate (6.38%), while it is affected by a very high false negative rate (57.14%). These results are not surprising, since Kirin relies on a very conservative set of rules based on permissions. Thus, it is affected by a reasonable number of false positives, at the price of missing many malicious behaviors, the underlying issue being that a logic bomb can be implemented without requesting highly-privileged permissions.

DroidAPIMiner. DroidAPIMiner is a malware detection tool based on machine learning. In particular, this tool uses as feature vector the set of Android APIs used by a given application. DroidAPIMiner is not open source: nonetheless, we were able to re-implement it based on the details provided in the research paper and the help of the authors. For our experiment, we used the k-nearest neighbors algorithm, since the authors reported it to be the most effective algorithm when detecting malware. We performed this experiment with multiple values of `k` (1, 3, and 5), we trained the classifier with our entire dataset, and we evaluated it by using leave-one-out cross-validation. On the one hand, the tool does not raise any false positive; on the other hand, it is affected by a very high false negative rate (78.57%), making this approach not reliable. These results are due to the fact that this approach would flag an application as malicious only if the dataset contains a malicious application that invokes very similar APIs. To make things even more difficult, it is very challenging (if not impossible) to obtain a comprehensive dataset of applications containing logic bombs, which makes any approach based on machine learning even less applicable when used to detect this category of malicious applications.

FlowDroid. FlowDroid is a state-of-the-art static analysis tool that aims at detecting sensitive privacy leaks in Android applications. Clearly, this tool has not been designed

Enabled Analysis Steps	True Positives	False Positives	True Negatives	False Negatives	False Positive Rate	False Negative Rate
Predicate Detection	14	1,386	7,927	0	14.88%	0%
Suspicious Predicate Analysis (in addition to the previous step)	14	462	8,851	0	4.96%	0%
Control-Dependency Analysis (in addition to the previous steps)	14	117	9,196	0	1.26%	0%
TRIGGERSCOPE (all analysis steps)	14	35	9,278	0	0.38%	0%

Table II: The table summarises the accuracy results of TRIGGERSCOPE. The total number of applications considered is 9,327, of which 9,313 benign and 14 malicious. This table also shows the results that would be obtained when only a subset of TRIGGERSCOPE's analysis steps is enabled.

Existing Analysis Tool	# Benign Apps	# Malware Apps	True Positives	False Positives	True Negatives	False Negatives	False Positive Rate	False Negative Rate
Kirin	9,309	14	6	594	8,715	8	6.38%	57.14%
DroidAPIMiner	9,313	14	3	0	9,313	11	0%	78.57%
FlowDroid	9,084	9	7	6,289	2,795	2	69.23%	22.22%
TRIGGERSCOPE	9,313	14	14	35	9,278	0	0.38%	0%

Table III: This table compares TRIGGERSCOPE's accuracy against the accuracy achieved by three existing, state-of-the-art analysis tools when tasked to detect logic bombs. The column "Benign Apps" and "Malware Apps" indicate the number of benign and malicious applications for which the given tool was able to successfully complete the analysis. For fairness, the false positive and false negative rates only consider those applications that were successfully analyzed.

to detect logic bombs. Nonetheless, we wanted to measure the accuracy of these systems when used in our context. For our experiment, we analyzed all the applications in our dataset by setting a timeout of one hour, and we considered a given application as malicious (or, more precisely, potentially-suspicious) if FlowDroid identified at least one suspicious data flow. Unsurprisingly, our results indicate that FlowDroid has a relatively low false negative rate (22.22%), while it is affected by a very high false positive rate (69.23%). Once again, these results are not surprising: benign applications often contain sensitive data flows, which directly lead to a high false positive rate. On the other hand, the presence of a sensitive data flow is not a necessary condition for the implementation of a logic bomb: this observation is the root cause for false negatives.

Discussion. Table III summarizes the results of this experiment. On the one hand, our results clearly show that existing analysis approaches are not suitable for the detection of logic bombs. In fact, analysis tools are either affected by a very high false negative rate (Kirin and DroidAPIMiner) or by a very high false positive rate (FlowDroid). On the other hand, TRIGGERSCOPE provides an excellent trade-off between false positives and false-negatives. Our analysis achieves better results due to the key observation upon which our approach is built: logic bombs are characterized by operations that are executed only under very narrow circumstances, while the actual, triggered behavior (encoded by the requested permissions, APIs that are invoked, and data flows throughout the application) plays a minor role.

F. Triggers in Benign Applications

This section provides insights related to the triggers that TRIGGERSCOPE detected in the three sets of benign applications crawled from the Play Store. Since these applications are benign, it is expected that the detected triggers are likely to be legitimate. However, all the detected triggers proved to be *interesting*, and worth of manual inspection. In particular, as reported later in this section, we identified two applications containing a SMS-based backdoor-like functionality. We also note that the manual analysis was quite effective because TRIGGERSCOPE returned precise information about the location and type of the trigger, hence making the task of manual vetting much simpler. The remainder of this section provides an overview of our findings. The detailed analysis of each of the detected triggers is reported in Appendix A, which we invite the interested reader to consult for more information.

Time-related checks. One common case is that the application contains a predicate that checks whether the current date is greater than a specific constant. Such a check is usually followed by a sensitive operation, such as a connection to the Internet, or setting an "expired" flag in a file. Another benign case of hard-coded checks is represented by those applications created for specific events that offer a countdown and alert the user whenever a specific date is reached. Another case is that of applications that allow users to schedule the sending of text messages at a future time. These classes of features are usually implemented by means of several checks on date-related objects – some of which, such as the minute value, are against hard-coded values.

Location-related checks. The number of applications that are characterized by suspicious location-related checks is lower than in the case of time-related checks. Intuitively, this makes sense. In fact, it is easier to envision reasons why benign applications might implement hard-coded time-related checks (for example, to implement expiration functionality or checks for updates) than to imagine cases where a benign application contains operations that are triggered only when the current location is within a specific area.

However, we did find a few applications that execute specific behaviors only if the current location is within a specific range. The prototype correctly identifies the suspicious location predicate as the conjunction of path constraints on the current latitude and longitude. In one example, when the user's location satisfies a series of constraints, the application displays "Welcome to Yamagata Station!" to the user. We manually verified that the range of *valid* latitudes and longitudes indeed identifies the actual location of the Yamagata train station in Japan. Note that while this specific trigger is clearly benign, this kind of check is exactly what could be used to implement a logic bomb to trigger a malicious behavior only when a soldier is located in a given war zone.

SMS-related checks. TRIGGERSCOPE identified a number of suspicious/interesting SMS-related checks in 17 apps. In particular, TRIGGERSCOPE identified an application, called MyRemotePhone [9], that allows users to remotely locate the device running the application itself. For this app, our tool automatically identified a suspicious SMS-related predicate that guards the execution of location-related APIs. In particular, the tool reported the following predicate: `(&& (!= (#sms/#body contains "MPS:") 0) (!= (#sms/#body contains "gps") 0))`. In natural language, the predicate is satisfied when the body of an SMS contains both the strings `"MPS:"` and `"gps"`. We then manually analyzed the application, and what we found was surprising: upon reception of an SMS satisfying the reported constraint (e.g., the string `"MPS: gps"`), the application would automatically reply back with an SMS containing the GPS coordinates of the device's position, thus leaking this sensitive information. To confirm this finding, we installed and executed the application on a real device, we sent an SMS containing the string `"MPS: gps"`, and, after a few seconds, we received an SMS containing the message `"Found at <latitude>, <longitude>!"`, where `<latitude>` and `<longitude>` identified the exact location of the device.

Another interesting application identified by our system is called RemoteLock [11]. A user of this application has the ability to remotely lock and unlock her device by sending an SMS containing a keyword. This keyword is user-defined, so it does not represent anything suspicious. However, TRIGGERSCOPE identified the following predicate: `"(!= (#sms/#body equals "adfbdfgertefvgdfgrehgj uiokhjgvbewruitkmbcvdfsgyhytdfsw")) 0)"`. In natural language, the predicate is triggered when the body of an incoming SMS contains a long, hard-coded string. Through manual analysis, we quickly discovered that this suspicious check is, in fact, used to implement a backdoor. To confirm our finding, we installed and executed the application on a real device, and we were able to unlock a *locked* phone just by sending an SMS with the hard-coded string identified by our system.

We found that the remaining apps make use of interesting checks to implement a variety of functionality. For example, we found a bank application that performs several checks on all incoming SMS as part of its implementation of a two-factor authentication scheme. Other applications implemented a mechanism similar to the one implemented by the MyRemotePhone app, but more securely, for example by authenticating the sender. Finally, other applications perform several checks on the body of all incoming SMS to implement some simple parsing routines, useful as a first step to implement custom communication protocols between compatible apps.

G. Logic Bombs in Malicious Applications

In our final experiments, we tested TRIGGERSCOPE on malicious samples. In a first experiment, we used our system to analyze potentially-malicious applications developed by a hostile DARPA Red Team organization. This data set is constituted by 11 applications. TRIGGERSCOPE identified a suspicious trigger in all of them. Five of them contained time-related triggers, implemented by comparing the current day, month, or year to hard-coded values; one application contained a location-related trigger, where it first performs several mathematical operations and conversions and then compares the current location against a hard-coded position using the `Location.distanceBetween()` method; the remaining five application contained triggers based on the content of SMS messages. When the trigger conditions are satisfied, these applications would execute a variety of unwanted behaviors, ranging from leaking sensitive information (e.g., user's location) to changing a security-sensitive password to a default, hard-coded one. For this data set, after the experiment the DARPA Red Team provided us the ground truth for each application (including details about all the triggers they contained), and we were able to verify the absence of both false positives and false negatives.

As a second experiment, we demonstrate how TRIGGERSCOPE is able to effectively identify trigger-based malicious behavior in real-world malware. The first example we considered is (informally) called "Holy Colbert" [54], collected by the Android Malware Genome Project [63]. For this sample, TRIGGERSCOPE was able to automatically discover a time-bomb: the app first retrieves the current date, converts it to a string by means of the `SimpleDateFormat` Android API, and it then compares the resulting string with the hard-coded value `"05212011"`. When this condition is satisfied, the application starts to send spam text messages to the entire contact list.

The second example we considered is a sample belonging to the infamous Zitmo malware family [27]. Zitmo is well-known for stealing mobile transaction numbers (mTANs) used to

implement two-factor authentication in banking applications. However, TRIGGERSCOPE was able to detect a different malicious functionality: our system detected several suspicious SMS-related checks. Upon manual inspection, we determine that the detected checks are used to implement a SMS-based bot-like C&C behavior. For the interested reader, we report the decompiled version of the function implementing this malicious functionality in Figure 8 in Appendix B.

The last real-world case-study we discuss is related to HackingTeam, a security company known to write surveillance software to assist law enforcement agencies and governments around the world. As mentioned in the introduction, in July 2015 this company was subject of a sophisticated attack, and, as a consequence, all its internal resources and personal communications were publicly leaked. Among these resources, researchers discovered the RCSAndroid [60] Android malware, a very powerful malicious Android application that offers remote control and spying capabilities. There are indications that this malicious application has been used as part of targeted attacks [57], [58], and it is thus relevant to our work.

We obtained a sample of this application, and we analyzed it using TRIGGERSCOPE. Our tool was able to identify a suspicious SMS-based trigger. In particular, TRIGGERSCOPE returned the following low-level constraint: `(&& (!= (#sms/#sender endswith v(#storage)) 0) (!= (#sms/#body startswith v(#storage))) 0)`. The constraint indicates that the sender of the SMS and the message body are checked against hard-coded values read from the file-system (indicated by the `#storage` tag). The analysis also determined that, only if the checks are satisfied, the `abortBroadcast()` method is invoked[2], which is suspicious. Upon manual investigation, we determined that these checks are implemented as part of a SMS-based backdoor-like functionality that gives the possibility to the *owner* of the application to trigger, at will, a variety of malicious actions. For example, it is possible to leak the victim's private conversations, GPS location, and device tracking information, but it is also able to capture screenshots, collect information about online accounts, and capture real-time voice calls.

VI. LIMITATIONS AND COUNTERMEASURES

Our analysis system has a number of limitations, which we discuss in this section.

First, our analysis system shares the same limitations of many other static analysis approaches. For example, it is possible that our implementation does not model precisely-enough the many Android-specific components (e.g., Binder RPC). We consider the complete and precise modeling of these aspects as out of scope and subject for future research.

Our prototype cannot currently fully analyze functionality that is implemented through the usage of reflection, dynamic code loading [46], native code [16], or invocations to the `Runtime.exec()` API: All these techniques could be used to implement generic forms of obfuscation, or to specifically hide the trigger-based nature of the code [50]. However, note that in a context like the U.S. DoD marketplace, the mere usage of these techniques (which can be reliably detected by any static analyzer, including TRIGGERSCOPE) would certainly raise suspicion, it would affect the stealthiness of the malware, and it would make existing malware detection techniques [18], [36], [64] (whose feature sets cover the previously-listed forms of obfuscation) more effective.

A second limitation is that our current prototype handles a limited number of trigger inputs (i.e., time, location, and text messages). However, we note that, although it would require a substantial engineering effort, it is conceptually easy to extend our prototype to handle additional sources. In fact, the current prototype already shows that it is possible to model complex Android objects (like `Date` and `Location`) as well as `String` objects (like a text message's body and sender). We also note that the key contribution of our work is not the development of a complete product, but it is to show that the idea of detecting malicious behavior by focusing on triggers – and not on the triggered behavior *per se* – is effective in practice. However, we acknowledge that, while the extension of the prototype is conceptually trivial, one would also need to perform additional experiments and analysis to study and characterize the kind of checks *normally* employed by benign apps, and to tune the classification routines accordingly.

The third limitation relates to the possibility for a malicious application to *move* the suspicious trigger outside the application itself, for example, to a web server. As our static analysis tool has no access to any remote code, the current prototype would not detect this suspicious behavior. However, a malicious application now relies on an external component, and this would affect both its reliability and stealthiness. Moreover, in order for an application to execute a sensitive operation only after a "signal" from an external component, the application would still need to include a check in its code, which could be detected by extending our prototype to detect triggers based on network inputs.

A final limitation is that a malicious application could attempt to obfuscate the implementation of a check so that it would resemble an innocuous recurring check. For example, the application could perform a series of mathematical operations that consists in adding and subtracting the same quantities so that, as a net result, the check is equivalent to a hard-coded check. However, while an application could attempt to obfuscate the semantics of a check, our system would still accurately record (in an expression tree) all the operations performed on the relevant objects. Even if, in the general case, it can be difficult to reconstruct the *un-obfuscated* semantics of a check, approaches based on anomaly detection (on the number and complexity of the operations involved) could be able to at least detect the mere obfuscation attempt, which, in certain scenario, might already be considered as ground for rejection.

[2]In previous versions of Android, the invocation of this method would have prevented the suspicious SMS to reach the default messaging application, hiding the reception of the SMS from the device's user.

VII. Related Work

In this section, we place our work in the context of other approaches to improving the security of the Android platform. Most relevant are static and dynamic analyses for either detecting or preventing attacks, which we describe in this section.

Static Analyses. Several static analysis approaches have been proposed to detect malicious Android applications. One of the first was Kirin [31], which recovers the set of permissions requested by applications with the goal of identifying potentially-malicious behavior. Other works include RiskRanker [36] and DroidRanger [64], which rely on symbolic execution and a set of heuristics to detect unknown malicious applications. FlowDroid [19] and DroidSafe [35] propose precise static taint analyses to detect potentially malicious data flows. Differently, Drebin [18] and DroidAPIMiner [14] are two approaches that extract several features from Android applications (e.g., requested permissions, invoked framework APIs) and then apply machine learning techniques to perform classification.

Similarly to these projects, TriggerScope aims to identify *suspicious* behavior in Android applications. However, while the goal of existing systems is to perform malware detection in the general sense, the main goal of our work is the identification of triggered malware (through the identification of logic bombs). In other words, we focus on the detection of functionality that are not malicious *per se*, but that could be considered as such because they are executed only under very specific circumstances. As discussed in the evaluation section, existing approaches are clearly outperformed by TriggerScope when tasked to detect triggered malware (like the one shown in Figure 1).

AppContext [62] is a system proposed in a software engineering venue. It leverages supervised machine learning to classify potentially malicious behaviors by taking into account the context in which such behaviors are executed. AppContext works in two steps: First, it starts from a set of actions that are known to be suspicious (i.e., methods that match known malware signatures [32], [2]); Then, the analysis adds context by considering which *category* of input controls the execution of such suspicious actions. While this approach shares the same basic observation as our work (i.e., just looking at behaviors alone is not enough to perform precise classification), it also significantly differs.

One of the main differences is that the set of behaviors that AppContext considers as suspicious (or, according to the terminology used in this paper, sensitive) is much narrower than ours. This aspect prevents this approach to detect logic bombs where the triggered behavior is not suspicious *per se* (as in our opening example in Figure 1). Note that AppContext's choice to select a narrow set of potentially-suspicious behaviors is necessary by design. In fact, since AppContext considers *any* check that involves certain inputs as a trigger (independently from the typology of the check itself), flagging a much wider set of actions as suspicious (as we do) would cause a very high false positive rate.

As we explained in the paper, in our work we take the opposite view: we first identify suspicious triggers (based on the checks that the code performs on inputs) and, only as a subsequent step, we consider which behavior these triggers can control. Thus, the triggered behavior *per se* has a much less important role in our analysis, and our approach can hence be much more lenient with the definition of sensitive operations. This different design choice allows us to detect triggers like the one shown in Figure 1, where the action (sending information to the network) is potentially sensitive but **not** suspicious *per se*, while having a very low false positive rate (0.38%).

Another important difference with respect to AppContext is that our static analysis provides details about each suspicious check, going well-beyond the mere detection (see our discussion in Sections V-F and V-G). Such checks reveal the actual trigger condition, such as the inputs needed to reach certain behaviors. This information is invaluable when automatically identifying logic bombs, but it is also very useful for efficient manual analysis.

Dynamic Analyses. As in the case of static analyzers, many systems have been proposed that apply dynamic analyses to Android applications in a security context. Hornyack et al. present AppFence [38], a dynamic system implemented as modifications to the Android framework that prevents attacks against user privacy via data shadowing. Along similar lines, Enck et al. [29] present TaintDroid, a dynamic taint analysis that performs whole-system data flow tracking through modifications to the underlying Android framework and native libraries. Other efforts, such as Mobile Sandbox [53], CopperDroid [49], [55], and Andrubis [40] developed tools and techniques to dynamically analyze unknown Android applications. Google also makes use of dynamic analysis in Bouncer [41], an automated system that screens submissions to the Google Play Store.

Finally, other research has proposed approaches based on dynamic analysis to perform multipath execution and dynamic symbolic execution of unknown Windows binaries [43], and on Java and Android applications [8], [34], [42], [61], [48]. These achieve higher code coverage than simpler dynamic analysis tools.

However, all of these systems share several fundamental limitations. Due to their nature, their analysis can be detected and evaded [21], [44] and they cannot guarantee complete coverage of the applications under test. Even more important, even in those cases where the functionality implemented in a logic bomb is *reached*, these systems are not capable of determining whether the just-executed *check* or functionality was malicious or not. In fact, at the very least, these tools would require a very fine-grained specification of the intended app behavior, something that is typically not available.

To solve this limitation, one approach would be to extend such systems to keep track of detailed information related to these checks, similarly to what TriggerScope does, to then reconstruct their semantics. Whether this is possible is a very interesting direction for future work. Nonetheless, static analysis systems are preferable, as they are not affected

by coverage-related issues, and, therefore, they do not risk missing relevant behavior due to the malicious functionality being executed only after, for example, a user successfully logs in.

One last project that relates to ours was proposed by Crandall et al. [25]. In that work, the authors aim to detect hidden time bombs in Windows binaries by running a virtual machine at different rates of *perceived time* and correlating memory write frequency to timer interrupt frequency. The authors show that their approach was able to detect time-related behaviors in four Windows worms. However, this work has also several limitations. First, it is based on dynamic analysis, and, therefore, it shares the limitations of the previous approaches. Second, this approach is intrinsically related to time-related behaviors and it would be extremely difficult, if not impossible, to adapt it to other trigger inputs. We note that TRIGGERSCOPE could be extended to other trigger inputs, the only challenge being engineering effort.

VIII. CONCLUSIONS

In this paper, we tackle the challenge of precisely identifying logic bombs in Android applications. To this end, we propose the idea of analyzing path predicates (checks) to determine whether they encode a *narrow* condition, and we introduced trigger analysis as a static program analysis for identifying suspicious trigger conditions that guard potentially-sensitive functionality in Android applications.

To evaluate the practicality of our idea, we implemented a prototype called TRIGGERSCOPE to detect time-, location-, and SMS-based triggers, and evaluated it over a large corpus of benign and malicious applications. Our evaluation demonstrates that trigger analysis is capable of automatically and precisely discovering both interesting and malicious path predicates on sensitive operations in these applications, including previously-unknown backdoors in benign apps from the official market, and a variety of logic bombs in real-world malicious samples. Finally, our experiments show that existing approaches are not suitable for detecting logic bombs.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank Sebastian Pöplau and Ryan Magennis for their help with the evaluation of our work, and Yousra Aafer for her help in reproducing DroidAPIMiner. Finally, we would like to thank Betty Sebright and her team for their significant help in motivating the development of this work.

This material is based on research sponsored by DARPA under agreements number FA8750-12-2-0101 and FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] AceDeceiver: First iOS Trojan Exploiting Apple DRM Design Flaws to Infect Any iOS Device. http://researchcenter.paloaltonetworks.com/2016/03/acedeceiver-first-ios-trojan-exploiting-apple-drm-design-flaws-to-infect-any-ios-device/.

[2] Androguard Signatures. https://code.google.com/p/androguard/wiki/DatabaseAndroidMalwares.

[3] Android Open Source Project (AOSP). https://source.android.com/.

[4] dex2jar - Tools to work with android .dex and java .class files. http://code.google.com/p/dex2jar/.

[5] HackingTeam's private conversation with Ecuador's representative – WikiLeaks. https://wikileaks.org/hackingteam/emails/emailid/630533.

[6] HackingTeam's private conversation with Egypt's representative – WikiLeaks. https://wikileaks.org/hackingteam/emails/emailid/530895.

[7] HackingTeam's private conversation with Saudi Arabia's representative – WikiLeaks. https://wikileaks.org/hackingteam/emails/emailid/74975.

[8] JPF-symbc: Symbolic PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

[9] My Remote Phone application. https://play.google.com/store/apps/details?id=com.innovationdroid.myremotephone.

[10] Pirated iOS App Stores Client Successfully Evaded Apple iOS Code Review. http://researchcenter.paloaltonetworks.com/2016/02/pirated-ios-app-stores-client-successfully-evaded-apple-ios-code-review/.

[11] RemoteLock application. http://www.androlib.com/android.application.tw-nicky-lockmyphonetrial-pqwBC.aspx.

[12] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[13] WALA: T.J. Watson Library for Analysis. http://wala.sourceforge.net/.

[14] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2013.

[15] Adrian Ludwig. Android Security State of the Union. Black Hat USA Briefings, 2015.

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

[17] AppBrain. Number of Available Android Applications. http://www.appbrain.com/stats/number-of-android-apps, March 2016.

[18] D. Arp, M. Spreitzenbarth, H. Malte, H. Gascon, and K. Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

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

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

[21] G. S. Babil, O. Mehani, R. Boreli, and M.-a. Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *Proceedings of the International Conference on Security and Cryptography*, 2013.

[22] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection*, 2007.

[23] C. Cadar, D. Dunbar, and D. Engler. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

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

[25] J. R. Crandall, G. Wassermann, D. A. S. D. Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[26] Defense Information Systems Agency, Department of Defense. DoD Mobility Program. http://www.disa.mil/Services/Enterprise-Services/Mobility.

[27] Denis Maslennikov. ZeuS-in-the-Mobile Facts and Theories. https://securelist.com/analysis/36424/zeus-in-the-mobile-facts-and-theories/.

[28] E. Dupuy. JD-Gui: Yet another fast java decompiler. http://jd.benow.ca/.

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

[30] W. Enck, M. Ongtang, and P. McDaniel. Kirin Analysis Tool. http://siis.cse.psu.edu/tools/kirin-0.1.tar.gz, 2009.

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

[32] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-Based Detection of Android Malware Through Static Analysis. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[33] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. CLAPP: Characterizing Loops in Android Applications. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2015.

[34] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM Sigplan Notices*, 2005.

[35] M. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[36] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[37] Hex-Rays. IDA Pro: a cross-platform multi-processor disassembler and debugger. http://www.hex-rays.com/products/ida/index.shtml.

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

[39] IDC. Smartphone OS Market Share, Q1 2015. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, May 2015.

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

[41] H. Lockheimer. Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, February 2012.

[42] N. Mirzaei, S. Malek, C. S. Pasreanu, N. Esfahani, and R. Mahmood. Testing Android Apps Through Symbolic Execution. In *ACM SIGSOFT Software Engineering Notes*, 2012.

[43] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[44] J. Oberheide. Dissecting Android's Bouncer. https://www.duosecurity.com/blog/dissecting-androids-bouncer, June 2012.

[45] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the USENIX Security Symposium*, 2013.

[46] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[47] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[48] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting Runtime Values in Android Applications that Feature Anti-Analysis Techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[49] A. Reina, A. Fattori, and L. Cavallaro. A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. *EuroSec*, 2013.

[50] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2008.

[51] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[52] P. Software. JEB: a Dalvik Bytecode Decompiler. https://www.pnfsoftware.com/.

[53] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, 2013.

[54] Symantec. Android Threat Set to Trigger On the End of Days, or the Day's End. http://www.symantec.com/connect/blogs/android-threat-set-trigger-end-days-or-day-s-end, May 2011.

[55] K. Tam, S. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[56] The Guardian. Hacking Team hacked: firm sold spying tools to repressive regimes, documents claim. http://www.theguardian.com/technology/2015/jul/06/hacking-team-hacked-firm-sold-spying-tools-to-repressive-regimes-documents-claim.

[57] The Huffington Post. Hacking Team, Maker Of Government Surveillance Software, Targeted In Attack. http://www.huffingtonpost.com/2015/07/06/hacking-team_n_7734926.html.

[58] The Register. Hacking Team's snoopware 'spied on anti-communist activists in Vietnam'. http://www.theregister.co.uk/2015/07/13/hacking_team_vietnam_apt.

[59] U.S. Department of Homeland Security, U.S. Federal Bureau of Investigation. Threats to Mobile Devices Using the Android Platform. http://publicintelligence.net/dhs-fbi-android-threats/, August 2013.

[60] Veo Zhang, Trend Micro. Hacking Team RCSAndroid Spying Tool Listens to Calls; Roots Devices to Get In. http://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-rcsandroid-spying-tool-listens-to-calls-roots-devices-to-get-in/.

[61] M. Y. Wong and D. Lie. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[62] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the International International Conference on Software Engineering (ICSE)*, 2015.

[63] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[64] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2012.

APPENDIX

A. Detailed Insights In Detected Suspicious Triggers

Application Package Name	Time-Triggered Behavior
bvz.commutesms	Automatically sends text messages and customizes their content given the hour of the day.
com.BjrM	Writes different files to disk depending on the day of the week.
com.bigtincan.android.adfree	Checks for expiration date.
com.blogspot.markalcalaramos.android	Automatically sends a text message 30 seconds after a missed call.
com.ghostleopard.weathermaxlite	Customizes its GUI by selecting different icons depending on the hour of the day.
com.px3j.lso	Checks for expiration date.
com.sivartech.GoogleIO	Notifications for Google I/O events.
com.vesperaNovus.app.StrayPhoneFinderFree	Uses time as a source of randomness, by checking `(1L & System.currentTimeMillis() / 1000L) == 0L`.
com.zyxsolutions.schedulersms	Performs different SMS-related operations depending on the day of the week.
nz.co.mobiledevelopment.ProfileController	Checks for expiration date.

Application Package Name	Location-Triggered Behavior
com.mv.tdt	Checks the current location against a set of predefined locations.
com.harmanbecker.csi.client	Checks whether the user's location is within a specified area.
net.dotquasar.android.lmakokoroid	Compares the current location with a previously stored location.
com.mv.mobie	Checks the current location against a set of predefined locations.
jp.nekorl.rainnetwork	Checks whether the current latitude is between -90 and 90, and whether the current longitude is between -180 and 180. This is done as a sanity check, and is easy to filter similar cases out if deemed uninteresting.
com.googlecode.androidcells	Compares the current location with a previously stored location.
com.mv.tdtespana	Checks the current location against a set of predefined locations.
jp.co.sha.YamagataMap	Checks whether the device is in the vicinity of Yamagata Station, Japan. If that is the case, the application displays "Welcome to Yamagata Station." The check is implemented by comparing the latitude and longitude against hardcoded values.

Table IV: This table provides an overview of the samples our system flagged as interesting. In particular, this table focuses on the time- and location-related interesting triggers. For each of these apps, TRIGGERSCOPE returned precise and useful information about the suspicious constraints. As an example, we consider the `com.px3j.lso` application. This application first retrieves the current date, and then compares it to a `Date` object that encodes the date May 10th, 2010. The comparison is performed by using the `Date.after()` method. The tool annotates the guarded basic block with the following raw constraint: `(!= (#now after Date(2010/5/10 0:0:0)) 0)`. Note how the current date is symbolically represented by `#now`, how the tool precisely models the involved APIs, and how it correctly identifies the suspicious comparison.

Application Package Name	SMS-Triggered Behavior
se.oscar.skandiabankensms	Online banking application that checks the *sender* number against the hardcoded value `+4781001001`. This is part of the implementation of a two-factor authentication scheme.
mk.bisera.smslocator	It checks if the incoming SMS contains the string `SMSLocator:`. This is done to determine whether the app should *handle* the received message. In such cases, the message was probably sent by the same application (or a *compatible* one) running on a different device.
com.messySMS.android_free	It checks if the incoming SMS starts with the `s://` string to determine whether the app should *handle* it.
tw.nicky.LockMyPhoneTrial (RemoteLock)	It checks whether the incoming SMS matches with the `adfbdfgertefvgdfgrehgjuiokhjgvbewruitkmbcvdfsgyhytdfsw` string. If that is the case, the application *unlocks* the phone. This is the implementation of the backdoor discussed in Section V.F.
com.app.publish	It checks if the *sender* field is set to the `nothing passed in` string. In this case, the application checks the value of the field against the *default* value, set to that hardcoded string.
no.knowit.widgets.beta	It checks if the incoming SMS starts with the `Disponibelt:` string to determine whether the app should *handle* it.
com.innovationdroid.myremotephone (MyRemotePhone)	It checks whether the incoming SMS contains the following two strings: `MPS:` and `gps`. If that is the case, the application automatically sends an SMS to the original sender containing the current GPS coordinates! We discussed this application in detail in Section V.F.
sms.encryptor.v2.free	It checks if the incoming SMS starts with `!@!` or `!$!`, to determine whether the app should *handle* it.
com.gallicsoft.shoppinglist	It checks if the incoming SMS starts with the `#SL#` string to determine whether the app should *handle* it.
com.mobileglobe.android.MobileGlobe	It checks if the incoming SMS starts with `(MobileGlobe)` or `(DirectGlobe)`, to determine whether the app should *handle* it.
com.counterpoint.dondeesta	It checks if the incoming SMS is equal to `"???"`. If that is the case, the application automatically sends an SMS to the original sender containing the current GPS coordinates. However, differently than MyRemotePhone, this application checks that the sender of the message belongs to a set of *trusted* numbers.
com.care2wear.imhere	This application implements the same functionality as the previous application. However, it checks for the `"?pos?"` string (instead of `"???"`), and directly notifies the user about the request (instead of relying on a whitelist).
com.sophos.mobilecontrol.client.android	It checks if the incoming SMS starts with `//sM/`, to determine whether the app should *handle* it.
com.opticaller.opticaller	It checks if the incoming SMS starts with `*OCProvision*` or `*OCLic*`, to determine whether the app should *handle* it.
com.mobileiron.vodafone.MIClient	It checks if the incoming SMS starts with `MICtrlCmd:`, to determine whether the app should *handle* it. If that is the case, the application would parse the SMS to determine which *command* must be executed.
com.telenor.hu.ematrica	It checks whether the *sender* is `+36208100000` or `+36208100100`, to determine whether the app should *handle* it.
com.amine.aloto	It checks whether the *sender* is `LOTO`, to determine whether the app should *handle* it.

Table V: This table provides an overview of the samples flagged as *interesting* by TRIGGERSCOPE. In particular, this table focuses on the triggers based on the content (or the sender) of incoming SMS messages.

B. Relevant Code Snippets

```
def isTrigger(p):
  if not isSuspicious(p):
    return False
  if controlsSensitiveAction(p):
    return True
  return False

def controlsSensitiveAction(p):
  # Get all blocks guarded by a given
  # predicate.
  blks = p.getGuardedBlocks()
  if isSensitive(blks):
    return True
  for b in blks:
    # Get all the objects that are
    # (directly or indirectly)
    # modified within this block.
    for obj in b.getModifiedObjects():
      # Get all the predicates that are
      # related to an object.
      preds = obj.getRelatedPredicates()
      for p2 in preds:
        if controlsSensitiveAction(p2):
          return True
  return False

def isSensitive(blocks):
  for b in blocks:
    # Get all invoked methods within
    # this block.
    for m in b.getInvokedMethods():
      if isSensitiveMethod(m):
        return True
      # Get all blocks of this method.
      mblks = m.getBlocks()
      if isSensitive(mblks):
        return True
  return False

def isSuspicious(predicate):
  # It returns True iff the predicate's
  # structure and semantic match the
  # definition of "suspicious" provided
  # by the user.
  ...

def isSensitiveMethod(method):
  # It returns True iff the target
  # (framework) method matches the
  # definition of "sensitive" provided
  # by the user.
  ...
```

Figure 7: Pseudocode of the `isTrigger` function (and related ones). This function returns True if and only if the given predicate matches our definition of trigger. The pseudocode is simplified for clarity.

```
1  public boolean
2    AlternativeControl(String param) {
3    if (param.startsWith("#")) {
4      SendControlInformation(
5        ExtractNumberFromMessage(param));
6      return true;
7    }
8
9    if (param.startsWith("/")) {
10     String str =
11       ExtractNumberFromMessage(param);
12     if (str.length() > 7) {
13       ValueProvider.SaveBoolValue(
14         "AlternativeControl", true);
15       ValueProvider.SaveStringValue(
16         "AlternativeNumber", str);
17       SendControlInformation(str);
18       return true;
19     }
20   }
21
22   // ...
23 }
```

Figure 8: The `AlternativeControl` method from the Zitmo malware which implements SMS-based command-and-control behavior. TRIGGERSCOPE automatically flagged the predicates involving `startsWith` and `length` method invocations on SMS data as suspicious (lines 3, 9, and 12).