Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models

Guillermo Suarez-Tangil[1], Mauro Conti[2],
Juan E. Tapiador[1], Pedro Peris-Lopez[1]

[1]Department of Computer Science, Universidad Carlos III de Madrid, Spain
guillermo.suarez.tangil@uc3m.es, {jestevez, pperis}@inf.uc3m.es
[2]Department of Mathematics, University of Padova, Italy
conti@math.unipd.it

Abstract. Malware for current smartphone platforms is becoming increasingly sophisticated. The presence of advanced networking and sensing functions in the device is giving rise to a new generation of targeted malware characterized by a more situational awareness, in which decisions are made on the basis of factors such as the device location, the user profile, or the presence of other apps. This complicates behavioral detection, as the analyst must reproduce very specific activation conditions in order to trigger malicious payloads. In this paper, we propose a system that addresses this problem by relying on stochastic models of usage and context events derived from real user traces. By incorporating the behavioral particularities of a given user, our scheme provides a solution for detecting malware targeting such a specific user. Our results show that the properties of these models follow a power-law distribution: a fact that facilitates an efficient generation of automatic testing patterns tailored for individual users, when done in conjunction with a cloud infrastructure supporting device cloning and parallel testing. We report empirical results with various representative case studies, demonstrating the effectiveness of this approach to detect complex activation patterns.

Key words: Smartphone security, targeted malware, cloud analysis.

1 Introduction

Malware for smartphones is a problem that has rocketed in the last few years [1]. The presence of increasingly powerful computing, networking, and sensing functions in smartphones has empowered malicious apps with a variety of advanced capabilities [2], including the possibility to determine the physical location of the smartphone, spy on the user's behavioral patterns, or compromise the data and services accessed through the device. These capabilities are rapidly giving rise to a new generation of *targeted* malware that makes decisions on the basis of factors such as the device location, the user's profile, or the presence of other apps (e.g., see [3–6]).

The idea of behaving differently under certain circumstances was also successfully applied in the past. For instance, Stuxnet [7] remained dormant until a particular app was installed and used at certain location, having as a target Iranian nuclear plants. Other malware has targeted governments and private corporations—mostly in the financial and pharmaceutical sectors [8]. Another representative example of targeted malware is Eurograbber [9], a "smart" Trojan targeting online banking users. The situational awareness provided by smartphone platforms makes this type of attacks substantially easier and potentially more dangerous. More recently, other examples of targeted malware include FinSpy Mobile [10], a general surveillance software for mobile devices, and Dendroid Remote Access Toolkit (RAT) [11], which offers capabilities to target specific users.

A similar problem is the emergence of the so-called *grayware* [3], i.e., apps that cannot be completely considered malicious but whose behavior may entail security and/or privacy risks of which the user is not fully aware. For example, many apps using targeted advertisements are particularly aggressive in the amount of personal data they gather, including sensitive contextual information acquired through the device sensors. The purpose of such data gathering activities is in many cases questionable, and many users might well disapprove of it, either entirely or in certain contexts[1].

Both targeted malware and grayware share a common feature that complicates their identification: the behavior and the potential repercussions of executing an app might depend quite strongly on the context where it takes place [12] and the way the user interacts with the app and the device [13]. We stress that this problem is not addressed by current detection mechanisms implemented in app markets, as operators are overwhelmed by the number of apps submitted for revision every day and cannot afford an exhaustive analysis over each one of them [14]. A possible solution to tackle this problem could be to implement detection techniques based on dynamic analysis (e.g., Taintdroid [15]) directly in the device. However, this is simply too demanding for battery-powered platforms. Several recent works [16–19] have proposed to keep a synchronized replica (clone) of the device virtualized in the cloud. This would facilitate offloading resource-intensive security analysis to the cloud, but still does not solve one fundamental problem: grayware and targeted malware instances must be provided with the user's particular context and behavior, so the only option left would be to install the app, use it, and expect that the analysis conducted over the clone—hopefully in real time—detects undesirable behaviors. This is a serious limitation that prevents users from learning in advance what an app would do in certain situations, without the need of actually reproducing such a situation.

Related Work. Recent works such as PyTrigger [20] have approached the problem of detecting targeted malware in Personal Computers (PC). To do so, it

[1] Classical examples include two popular games, *Aurora Feint* and *Storm8*, which were removed from the Apple Store for harvesting data and phone numbers from the user's contact list and sending them to unknown destinations [2].

is sought to trigger specific malware behaviors by injecting activities collected from users (e.g., mouse clicks and keyword inputs) and their context. This approach cannot be adopted to platforms such as smartphones because the notion of *sensed* context is radically different here. Other schemes, including the work presented in [21–23, 13], do focus on smartphones but concentrate exclusively on interactions with the Graphical User Interface (GUI) and are vulnerable to context-based targeted attacks. Two works closer to our proposal are Context Virtualizer [24] and Dynodroid [25], where a technique called context fuzzing is introduced in the former and used in the latter. The main aim in [24, 25] is to automatically test apps with real-world conditions, including user-based contexts. These tools, however, are intended for developers who want to learn how their apps will behave when used in a real setting. Contrarily, our focus is on final users who want to find out if they will be targeted by malicious or privacy-compromising behaviors. Finally, other works such as CopperDroid [26] focus on malware detection as we do, but with a static approach based on information extracted from the manifest that, besides, does not consider the user context.

Contribution. In this paper, we address the problem of identifying targeted grayware and malware and propose a more flexible approach compared to other proposals to determining whether the behavior of an app is compliant with a particular set of security and privacy preferences associated with a user. Our solution is based on the idea of obtaining an *actionable* model of user behavior that can be leveraged to test how an app would behave should the user execute it in some context. Such a testing takes place over a clone of the device kept in the cloud. This approach removes the need of actually exposing the device (e.g., we let the device access only to fake data and not real one). More importantly, the analysis is tailored to a given user, either generally or for a particular situation. For example, a user might want to explore the consequences of using an app in the locations visited during working days from 9 to 5, or during a planned trip.

Organization. Section 2 introduces the theoretical framework used to model triggering patterns and app behavior. In Section 3, we describe the architecture of our proposal and a proof-of-concept prototype, and discuss the experimental results obtained in terms of testing coverage and efficiency. In Section 4, we discuss the detection performance with two representative case studies of grayware and targeted malware instances. Finally, Section 5 concludes the paper by summarizing our main contributions and describing future research directions.

2 Behavioral Models

This section introduces the theoretical framework used in our proposal (presented later in Section 4) to trigger particular app behaviors and determining whether they entail security risks to the user. More precisely, we next present models for the user-provided inputs, the resulting app behavior, and the mechanism used to assess potential risks.

2.1 Triggering Patterns

Inputs provided by the user to his device constitute a major source of stimuli for triggering certain app behaviors. We group such inputs into two broad classes of patterns, depending on whether they refer to inputs resulting from the user directly interacting with the app and/or the device (e.g., through the touchscreen), or else indirectly by the context (e.g., location, time, presence of other devices in the surroundings, etc.).

Usage Patterns Usage patterns model sequences of events resulting from the actions of the user during his interaction with an app. Such events are internal messages passed on to the app by the device, such as starting an activity or clicking a button. We stress that our focus is on the events and not on the actions that generate them, as the same event can be triggered through different input interfaces (e.g., touchscreen and voice).

Let the following be a set of all possible events for all apps:

$$\mathcal{E} = \{e_1, e_2, \ldots, e_n\}. \tag{1}$$

Thus, the interaction of a user with an app can be represented as an ordered sequence:

$$\mathbf{u} = \langle \epsilon_1, \epsilon_2, \ldots, \epsilon_k \rangle, \quad \epsilon_i \in \mathcal{E}. \tag{2}$$

We will refer to such sequences as *usage traces*. Interactions with an app at different times and/or with different apps will result in different usage traces.

Context Patterns Apps may behave differently depending on conditions not directly provided by the user, such as the device location, the time and date, the presence of other apps or devices, etc. We model this using the widely accepted notion of *context* [27]. Assume that $v_1, \ldots, v_m$ are variables representing contextual elements of interest, with $v_i \in \mathcal{V}_i$. Let the following be the set of all possible contexts:

$$\mathcal{C} = \mathcal{V}_1 \times \cdots \times \mathcal{V}_m. \tag{3}$$

As above, monitoring a user during some time interval will result in a sequence:

$$\mathbf{t} = \langle c_1, c_2, \ldots, c_l \rangle, \quad c_i \in \mathcal{C}. \tag{4}$$

We will refer to such sequences as *context traces*.

2.2 Stochastic Triggering Model

Usage and context traces are used to derive a model that captures how the user interacts with an app or a set of apps. For this purpose, we rely on a discrete-time first-order Markov process (i.e., a Markov chain [28]) $\mathbf{M} = (S, A, \Pi)$ where:

– The set of states S is given by:

$$S = \mathcal{E} \times \mathcal{C} = \{s_1, \ldots, s_N\}. \tag{5}$$

We will denote by $q(t) \in S$ the state of the model at time $t = 1, 2, \ldots$, representing one particular input event executed in a given context.

– The transition matrix is given by:

$$A = [a_{ij}] = P[q(t+1) = s_j | q(t) = s_i], \tag{6}$$

where $a_{ij} \in [0, 1]$ and $\sum_{j=1}^{N} a_{ij} = 1$.

– The vector of initial probabilities is given by:

$$\mathit{\Pi} = (\pi_i) = P[q(1) = s_i], \tag{7}$$

with $\pi_i \in [0, 1]$ and $\sum_{i=1}^{N} \pi_i = 1$.

The model above is simple yet powerful enough to model user-dependant behavioral patterns when interacting with an app. The model parameters can be easily estimated from a number of usage and context traces. Assume that $\mathbf{O} = \{o_1, o_2, \ldots, o_T\}$ is a sequence of observed states (i.e., event-context pairs) obtained by monitoring the user during a representative amount of time. The transition matrix can be estimated as:

$$a_{ij} = \frac{\sum_{t=2}^{T} P[q(t) = s_j | q(t-1) = s_i]}{\sum_{t=2}^{T} P[q(t-1) = s_j]} = \frac{\sum_{t=2}^{T} P[o_t = s_j | o_{t-1} = s_i]}{\sum_{t=2}^{T} P[o_{t-1} = s_j]}, \tag{8}$$

where both probability terms are obtained by simply counting occurrences from $\mathbf{O}$. The process can be trivially extended when several traces are available.

The model above should be viewed as a general modeling technique that can be applied at different levels. Therefore, if one is interested in modeling input events irrespective of their context, the set of states—and, therefore, the chain—can be reduced to $\mathcal{E}$. The same applies to the context, i.e., states could be composed exclusively of time-location pairs.

Markov chains are often represented as a directed graph where vertices represent states and edges between them are labeled with the associated transition probability. We will call the *degree* of a state, denoted by $\mathbf{deg}(s_i)$, to the number of states reachable from s in just one transition with non-null probability:

$$\mathbf{deg}(s_i) = \#\{p_{ij} | p_{ij} > 0\}. \tag{9}$$

The degree distribution of a chain is given by

$$\mathbf{P}(k) = P[\mathbf{deg}(s) = k]. \tag{10}$$

2.3 App Behavior and Risk Assessment

An app interacts with the device by requesting services through a number of available system calls. These define an interface for apps that need to read/write files, send/receive data through the network, make a phone call, etc. Rather than focusing on low-level system calls, in this paper we will describe an app behavior through the sequence of *activities* it executes (see also [29]). Activities represent high-level behaviors, such as for example reading from or writing into a file, opening a network connection, sending/receiving data, etc. In some cases, there will be a one-to-one correspondence between an activity and a system call, while in others an activity may encompass a sequence of system calls executed in a given order. In what follows, we assume that:

$$\mathcal{A} = \{a_1, a_2, \ldots, a_r\} \tag{11}$$

is the set of all relevant activities observable from an app execution.

The execution flow of an app may follow different paths depending on the input events provided by the user and the context. Let $\sigma = \langle \sigma_1, \ldots, \sigma_k \rangle$ be a sequence of states as defined above. We model the behavior of an app when executed with σ as input as the sequence:

$$\beta(\sigma) = \langle \alpha_i, \ldots, \alpha \rangle, \quad \alpha_i \in \mathcal{A}, \tag{12}$$

which we will refer to as the *behavioral signature* induced by σ.

Behavioral signatures constitute dynamic execution traces generated with usage and context patterns specific to one particular user. Analysis of such traces will be instrumental in determining whether there is evidence of security and/or privacy risks for that particular user. The specific mechanism used for that analysis is beyond the scope of our current work. In general, we assume the existence of a *Risk Assessment Function* (RAF) implementing such an analysis. For example, general malware detection tools based on dynamic analysis could be a natural option here. The case of grayware is considerably more challenging, as the user's privacy preferences must be factored in to resolve whether a behavior is safe or not.

3 Targeted Testing in the Cloud

In this section, we first describe the architecture and the prototype implementation of a cloud-based testing system for targeted malware and grayware based on the models discussed in the previous section. We then provide a detailed description of various experimental results obtained in two key tasks in our system: obtaining triggering models and using them to test a cloned device.

3.1 Architecture and Prototype Implementation

A high level architectural view of our system is shown in Fig. 1. There are two differentiated major blocks: (i) the *evidence generation* subsystem, and (ii) the

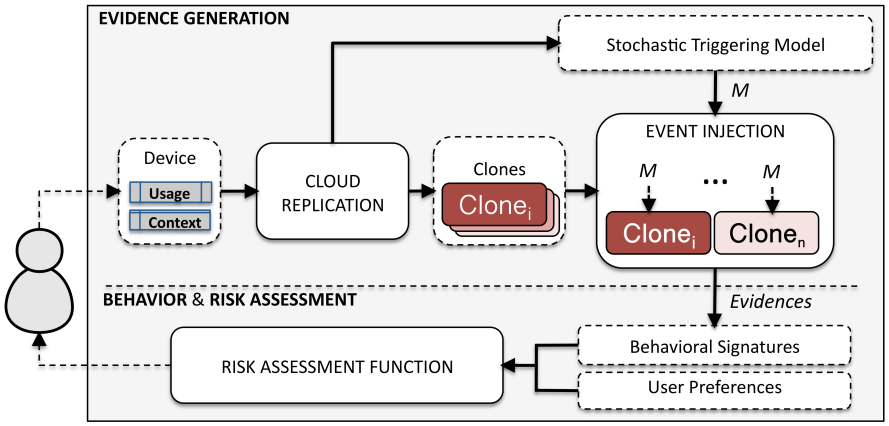


Fig. 1: System architecture and main building blocks.

behavioral modeling and risk assessment subsystem. The first one extracts usage and context traces from the device and generates the stochastic triggering model. This process is carried out by first cloning the user device in the cloud and then injecting the triggering patterns over the clone. The second block extracts the behavioral signatures from the clone(s) and applies the RAF over the evidences collected. We next provide a detailed description of our current prototype implementation.

The experiments have been conducted using a smartphone and a virtual mobile device in the cloud, both running Android OS 2.3. In particular, a Google Nexus One is used for the physical device and an Android emulator [30] for the clones. The device is instrumented with various monitoring tools that collect user events, the context, and the device configuration and transmits them to the cloud. For this purpose, we used a combination of `logcat` and `getevent` tools from the Android Debug Bridge (ADB) [30].

Our proof-of-concept implementation builds on a number of previous works for cloud cloning smartphone platforms [16–19] and for performing behavioral analysis [31, 15]. In the cloud end, a middleware implemented in `Python` processes all inputs received, generates the associated models, and runs the simulation. We inject events and contexts into apps with a combination of a testing tool called `Monkeyrunner` [30] and the Android emulator console [30].

As for the behavioral signatures obtained in the virtual device, we have used an open source dynamic analysis tool called `Droidbox` [31] to monitor various activities that can be used to characterize app behavior and tell apart benign from suspicious behavior [2]. `Droidbox` is based on `TaintDroid` [15] and provides a variety of data about how an app is behaving. We chose 20 relevant activities to characterize app behavior (see Table 1), which include information about calls to the crypto API (`cryptousage`), I/O network and file activity (`opennet`, `sendnet`, `accessedfiles`, etc.), phone and SMS activity

(`phonecalls`, `sendsms`), data exfiltration through the network (`dataleak`), and dynamic code injection (`dexclass`), among others.

Activities				
• `sendsms`	• `servicestart`	• `phonecalls`	• `udpConn`	• `cryptousage`
• `permissions`	• `netbuffer`	• `activities`	• `dexclass`	• `activityaction`
• `dataleak`	• `enfperm`	• `opennet`	• `packageNames`	• `sendnet`
• `recvs`	• `recvnet`	• `recvsaction`	• `fdaccess`	• `accessedfiles`

Table 1: Set of activities ($\mathcal{A}$) monitored from an app execution and used to characterize its behavior.

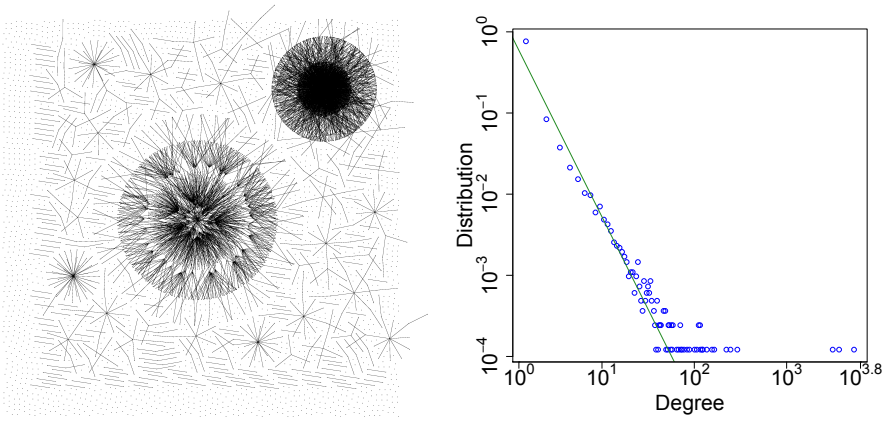
Finally, we implemented a simple yet powerful RAF (*Risk Assessment Function*) for analyzing behavioral signatures. Due to space reasons, we only provide a high-level description of this mechanism. In essence, the scheme is based on a pattern-matching process driven by a user-specified set of rules that identify behaviors of interest according to his security and privacy preferences. Such rules are first-order predicates over the set of activities $\mathcal{A}$, allowing the user to specify relatively complex patterns relating possible activities in a signature through logical connectives. Regardless of this particular RAF, our prototype supports the inclusion of standard security tools such as antivirus packages or other security monitoring components. These can be easily uploaded to the clone and run while the testing carries on.

3.2 Experiment I: The Structure of a Triggering Model

In this first experiment, we monitored all events triggered by a user executing several apps on his device during a representative amount of time. More precisely, we collected traces from the user while interacting with the OS and several apps such as Facebook, YouTube, and Google Maps. We assumed that the events collected were representative enough, as user behavior is generally very redundant. The resulting event set contained about $|S|$ =8K states, distributed over various observations traces of around $|\mathbf{O}| = 37$K states. We then used such traces to estimate the transition matrix using Eq. (8). The obtained Markov chain turned out to have various interesting features. For example, its degree distribution follows a power-law of the form $\mathbf{P}(k) = k^{-\alpha}$ (see Fig. 2) with $\alpha = 2.28$ for $k \geq 2$. This suggests that events and contexts follow a scale-free network [32], which is not surprising. Recall that an edge between two nodes (events) indicates that the destination event occurs after the source event.

A power-law distribution such as the one shown in Fig. 2 reveals that most events have an extremely low number of "neighbors"; i.e., once an event has happened, the most likely ones coming next reduce to about 100 out of the 8K possible. Only a small fraction of all events are highly connected, meaning

that almost any other event is possible to occur after them. For instance, in our traces we found that over half of the states were only connected to just one state. In contrast, one state was found to be connected to more than 4K other states.



(a) (b)

Fig. 2: (a) Markov model representing contextual and kernel input events for a user interacting with an Android platform; (b) Degree distribution, in log-log scale, of the model in (a) as defined in Section 2.2.

These results make sense due to a simple reason: input and context events do depend quite strongly on those issued immediately before. For example, the probability of moving from one place to another nearby is much higher than to a remote place. The same applies to sequences of events, where the probability distribution of the next likely event reflects the way we interact with the app. As we will next see, this structure makes testing extremely efficient.

3.3 Experiment II: Speed of Testing

We performed a number of experiments to measure how fast input events can be injected into an Android application sandbox. Such events include not only input events, but also a variety of context traces comprising phone calls, SMS messages, and GPS locations. We recorded and analyzed the time taken by both the sandbox and the operating system to process each injected event. Our results suggest that the time required to process injected states (input or context events) varies quite strongly depending on the type of state (see Table 2). For instance, it takes around 0.35 seconds, on average, to inject an SMS and process it trough the operating system. In contrast, geolocation events can be injected almost 100 times faster. We also observed a significant difference between the capabilities of the sandbox and the OS running on top of it. For instance, while the sandbox is

able to process about 2800 geolocation states per second, the OS can only absorb around 100 each second. We suspect that this throughput might be improved by using more efficient virtual frameworks, such as for example Qemu for Android x86[2] or ARM-based hardware for the cloud[3].

Automatic Injection		
Injected Event	**Emulator Layer**	**App Layer**
Sensor event	7407.66 events/s	1.26 events/s
Power event	361.77 events/s	19.16 events/s
Geolocation event	2810.15 events/s	111.87 events/s
SMS event	451.27 events/s	0.35 events/s
GSM call/cancel event	1726.91 events/s	0.71 events/s

Human Generated		
Event Type	**Average**	**Peak**
Usage patterns	5 events/s	10 events/s
Context patterns	10 events/s	25 events/s

Table 2: Event injection rates for different types of events over a virtualized Android device (top), and rates generated by real users based on profiling 67 apps [33] (bottom).

For comparison purposes, the lower rows in Table 2 show the average and peak number of events generated by human users, both for usage (e.g., touch events) and context events, as reported in previous works [33].

3.4 Experiment III: Coverage and Efficiency

We finally carried out various experiments to evaluate the performance of our proposal. Our main aim here was measuring the time required to reach an accurate decision by means of simulation. More precisely, we simulated an injection system configured with randomly generated **u** and **t** patterns and with different number of states: $|S| = 100$, 1000, and 4000.

The configuration of each experiment was based on the findings discussed in previous sections, as detailed bellow. First, we generated two types of Markov model chains: (i) one random scale-free network of events using a preferential attachment mechanism as defined by *Barabási-Albert* (BA) [34], and (ii) another random network following the well-known *Erdős-Rényi* (ER) model [35]. Then, we simulated a user providing inputs to a device together with its context at a rate of 10 events per second. We chose this throughput as it is a realistic injection rate (see Table 2).

[2] `www.android-x86.org/`
[3] `http://armservers.com/`

In each experiment, we generated a number of random Markov chains and calculated the cumulative transition probability covered when traversing from one state to another of the chain for the first time. Formally, let:

$$w = \langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle, \qquad s_{i_j} \in S, \tag{13}$$

be a random walk over the chain, with $a_{i_j i_{j+1}} > 0 \; \forall i_j$, and let:

$$T(w) = \{(s_{i_j}, s_{i_{j+1}}) \mid s_{i_j} \in S \setminus \{s_{i_n}\}\}, \tag{14}$$

be the set of all transitions made during the random walk. We define the coverage of w as the amount of transitions seen by w, weighted by their respective probabilities and normalized to add up to one, i.e.:

$$\texttt{Coverage}(w) = \frac{1}{N} \sum_{(p,q) \in T(w)} a_{pq}. \tag{15}$$

The coverage is used to evaluate both the efficiency and the accuracy of our system. On the one hand, it can be used to measure the amount of a user's common actions triggered given a limited period of testing time. Additionally, it also shows how fast the system tests the most common actions. Results for sets of events of various sizes are shown in Fig. 3, where the curves have been averaged over 10 simulations. The results show that the coverage reached when testing networks of sizes $|S| = 100$, 1000, and 4000 states is very satisfactory. Such a good performance is related to the scale-free distribution of states through time, since this property allows to test the most common actions performed by the user very rapidly. Thus, a coverage above 80% is reached in less than two minutes for 100 states, and in approximately 1 hour for 4000 states.

It is important to emphasize that the coverage reported in Fig. 3 corresponds to one test sequence randomly drawn according to the user behavioral model. If the process is repeated or carried out in parallel over various clones, other test sequences may well explore behaviors not covered by the first one. This is illustrated in Table 3, where we show the total testing coverage as a function of the number of clones tested in parallel, each one provided with a different input sequence. Thus, two hours of testing just one clone results in a coverage slightly above 84%. However, if five clones are independently tested in parallel, the overall result is a coverage of around 93% of the total user behavior. This time-memory trade-off is a nice property, allowing to increase the coverage by just testing multiple clones simultaneously rather than by performing multiple test over the same clone.

Reaching a 100% coverage is, in general, difficult due to the stochastic nature of the models. This is not critical, as those behavioral patterns that are left unexplored correspond to actions extremely unlikely to be executed by the user. In practical terms this is certainly a risk, but one relatively unimportant as the presumably uncovered malware instance would not activate for this user except with very low probability.

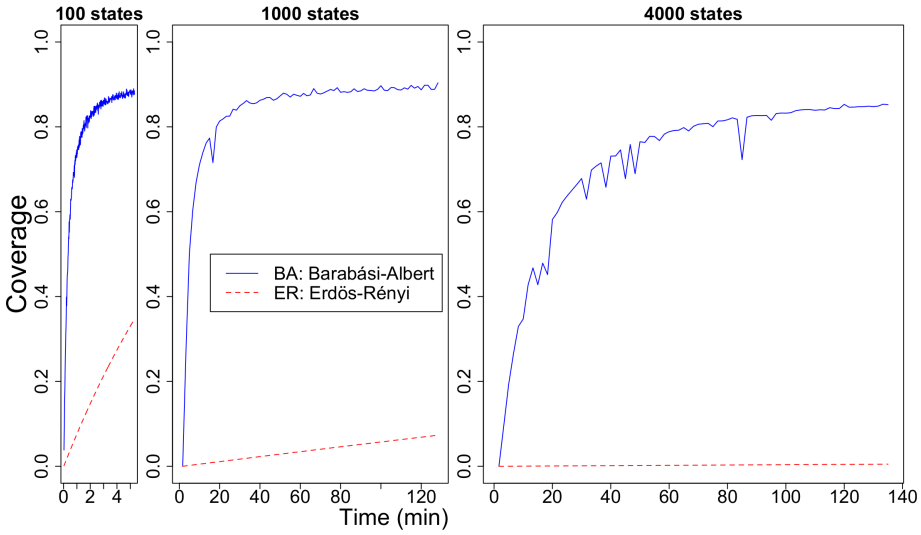


Fig. 3: Efficiency and accuracy of the decision for a Barabási-Albert and Erdős-Rényi network model.

	Number of parallel clones									
	1	2	3	4	5	6	7	8	9	10
10 min.	42.2%	60.6%	68.8%	73.8%	76.9%	79.2%	81.9%	81.8%	82.5%	83.4%
60 min.	79.3%	86.6%	89.1%	90.2%	90.5%	91.1%	91.3%	91.5%	91.7%	95.0%
120 min.	84.3%	87.2%	88.1%	88.5%	93.3%	93.4%	93.6%	93.8%	93.8%	93.9%

Table 3: Testing coverage when running multiple parallel clones given a limited testing time for a network of $|S| = 4000$ states.

4 Case Studies

In this section, we present two case studies illustrating how the injection of user-specific behavioral patterns can contribute to revealing malware with targeted activation mechanisms. We cover *dormant* and *anti-analysis* malware, as these scenarios constitute representative cases of targeted behaviors in current smart devices [2]. For each case, we first provide a brief description of the rationale behind the malware activation condition and then discuss the results obtained after applying the injection strategy presented in this paper. In all cases, the evaluation has been conducted by adapting an open source malware called *Androrat* (Android Remote Access Tool, or RAT) [36] and incorporating the specific triggering conditions.

4.1 Case 1: Dormant Malware/Grayware

Piggybacked malware [37] is sometimes programmed to remain dormant until a specific situation of interest presents itself [38]. This type of malware is eventually activated to sense if the user context is relevant for the malware. If so, then some other malicious actions are executed. For instance, a malware aiming at spying a very specific industrial system, such as the case of Stuxnet, will remain dormant until the malware hits the target system. Similarly, in a Bring-Your-Own-Device (BYOD) context, malware targeting a specific office building can remain dormant until the device is near a certain location.

Typically, malicious apps are activated when the `BOOT_COMPLETED` event is triggered regardless of the context of the infected device. A recent study on Android malware [38] suggests that the tendency is shifting towards more sophisticated activation triggers so as to better align with the malware incentives and the pursued goals. This results in a variety of more complex activation conditions, such as those shown in Table 4.

Wake-up conditions	
User presence	USB connected, screen-on action, accelerator changed, etc.
Location	Location change event, near an address, leaving an area, etc.
Time	A given day and time, after a certain period of time, etc.
Hardware	Power and LED status, KEY action, LOCK event, etc.
Configuration	Apps installed, a given contact/phone number in the agenda, etc.

Table 4: Typical wake-up conditions for malware activation.

We instrumented Androrat to activate the RAT component only when the device is in a certain location. We use a mock location near the Bushehr nuclear plant, simulating a possible behavior for a Stuxnet-like malware. Specifically, the RAT is only activated when the device is near the location: 28.82781 ° (latitude) and 50.89114 ° (longitude). Once the RAT is activated, we send the appropriate commands to exfiltrate ambient and call recordings captured through the microphone, the camera, and the camcorder.

For testing purposes, we built a symbolic model representing the abstract geographic areas of a given user working at the Bushehr plant. Fig. 4 represents the Markov Model chain for the different areas and the transitions between them. For instance, the model represents a user traveling from HOME (c_H) to WORK (c_W) with a probability of $P(c_H|c_W) = 0.7$.

Given the above model, we then injected testing traces drawn from the chain into the sandbox instrumented with Androrat. The sandbox was configured with a generic RAF aiming at identifying when operations involving personal information occur together with network activity. The results show how the malware is not activated until we start injecting mock locations. A few seconds

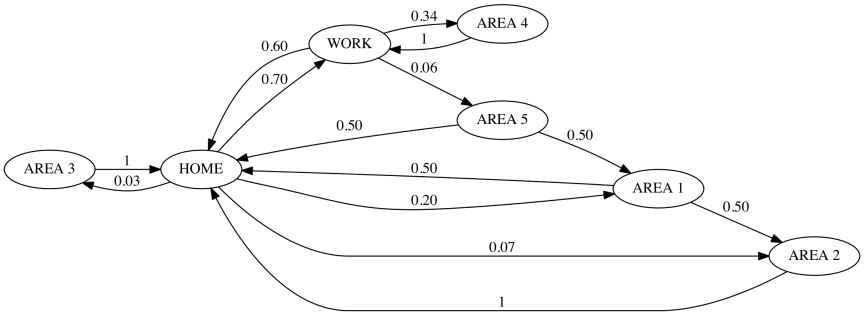


Fig. 4: Markov chain for the location.

after the first injection, the behavioral signature collected reported, as expected, both data leakage (`dataleak`) and network activity (`sendnet`).

We next defined an alternative scenario in which an app accesses the user location and sends an SMS to one of his contacts whenever he is leaving a certain region, such as for instance WORK (c_W). To this end, we implemented an app and tested it against three users with different contexts and concerns about their privacy. The first user has strict privacy policies and visits very frequently the location c_W. The second user has the same policy as the first one but has never visited such a location. Finally, the last user visits c_W as well but has a more flexible privacy policy. For the sake of simplicity, we used the same triggering model described in the previous example for users one and three (see Fig. 4), while the second user has a different Markov chain. Results show that:

- For the first user, the behavioral signature reported data leakage activity (`dataleak`) as well as SMS activity (`sendsms`). As both are in conflict with this user's privacy preferences, this is marked as undesirable behavior.
- In the case of the second user, the model injects locations other than those triggering the grayware component. Consequently, no significant behavioral signature is produced.
- Finally, the events injected for the third user do trigger the grayware component, resulting in data leakage and SMS activity. However, as these are not in conflict with his privacy preferences, no alert is issued.

This example reinforces the view that not only malware activation can be user specific, but that the consequences of such a malware may also be perceived very differently by each user.

4.2 Case 2: Anti-analysis Malware

Malware analysis is typically performed in a virtual sandbox rather than in a physical device due to economic and efficiency factors [2]. These sandboxes often have a particular hardware configuration that can be leveraged by malware

instances to detect that they are being analyzed and deploy evasion countermeasures [11], for example by simply not executing the malicious payload if the environment matches a particular configuration. Sandboxes for smartphone platforms have such artifacts. For instance, the IMEI, the phone number, or the IP address are generally configured by default. Furthermore, other hardware features such as the battery level are typically emulated and kept indefinitely at the same status: e.g., AC on and Charging 50%. Table 5 summarizes some of these features in most Android emulators along with their default value.

HW feature	**Default value**
IMEI	000000000000000
IMSI	012345678912345
SIM	012345678912345
Phone Number	1-555-521-PORT (5554)
Model Number	sdk
Network	Android
Battery Status	AC on Charging 50%
IP Address	10.0.2.X

Table 5: Default hardware configuration for Android emulator.

Hardware features such as those described above can be set prior to launching the sandbox. This will prevent basic fingerprinting analysis, for example by setting random values for each execution. However, smarter malware instances might implement more sophisticated approaches, such as waiting for a triggering condition based on a combination of hardware changes. Motivated by this, we modified Androrat to activate the RAT component only after AC is off and the battery status is different from 50%. Once the RAT is activated, we send appropriate commands to exfiltrate some personal information from the device such as SMSs, call history, etc.

In principle, there are as many triggering conditions as combinations of possible hardware events. Although our framework support injection of all possible hardware events via the Android emulator console [30], for simplicity we restricted our experimentation to the subset of power-related events described in Table 6.

Based on the different power states, we built a model of the battery usage extracted from an actual device when used by a real user. The resulting model is shown in Fig. 5. We then tested Androrat against this model generated using the same RAF configuration used in previous cases. The results show that the behavioral signature not only reported `dataleak` and `sendnet`, but also file activity (`accessedfiles`), thus confirming that the malware activated as it failed to recognize its presence in a sandbox.

status	health	present	AC	capacity
unknown charging discharging not-charging full	unknown good overheat dead overvoltage failure	false true	off on	$0 - 100\%$

Table 6: Different hardware states for power status of the device.

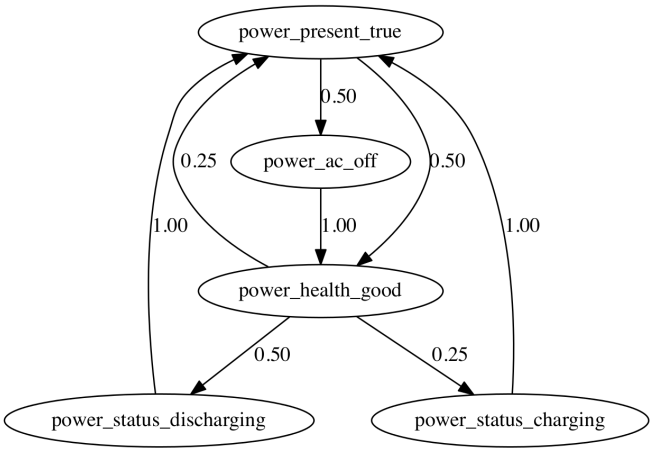


Fig. 5: Markov chain for the battery status.

5 Conclusions

The problem of detecting targeted malware via behavioral analysis requires the ability to reproduce an appropriate set of conditions that will trigger the malicious behavior. Determining those triggering conditions by exhaustively searching through all possible states is a hard problem. In this paper, we have proposed a novel system for mining the behavior of apps in different user-specific contexts and usage scenarios. One of our system's main aims is providing the testing environment (replicas in the cloud) with the same conditions than those the actual device is exposed to. Our experimental results show that modeling such conditions as Markov chains reduces the complexity of the search space while still offering an effective representation of the usage and context patterns. In essence, our system is able to trigger a targeted malware as long as: (i) it also activates in the device; and (ii) the user behavior is appropriately modeled. However, we also anticipate that a more sophisticated adversary could exploit some features

of our model to evade detection. This weakness will be further explored and addressed in future work.

Our approach represents a robust building block for thwarting targeted malware, as it allows the analyst to automatically generate patterns of input events to stimulate apps. As the focus of this paper has been on the design of such a component, we have relied on ad hoc replication and risk assessment components to discuss the quality of our proposal. We are currently extending our system to support: (a) a replication system to automatically generate and test clones of the device under inspection; and (b) a general framework to specify risk assessment functions and analyze behavioral signatures obtained in each clone. Finally, in this paper we have not discussed the potential privacy implications associated with obtaining user behavioral models. Even if such profiles are just used for testing purposes, they do contain sensitive information and must be handled with caution. This and other related privacy aspects of targeted testing will be tackled in future work.

Acknowledgment

We are very grateful to the anonymous reviewers for constructive feedback and insightful suggestions that helped to improve the quality of the original manuscript.

The work of G. Suarez-Tangil, J.E. Tapiador, and P. Peris-Lopez was supported by the MINECO grant TIN2013-46469-R (SPINY: Security and Privacy in the Internet of You). Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission under the agreement No. PCIG11-GA-2012-321980. This work is also partially supported by the TENACE PRIN Project 20103P34XC funded by the Italian MIUR, and by the Project "Tackling Mobile Malware with Innovative Machine Learning Techniques" funded by the University of Padua.

References

1. Juniper: 2013 mobile threats report. Technical report, Juniper Networks (2013)
2. Suarez-Tangil, G., Tapiador, J.E., Peris, P., Ribagorda, A.: Evolution, detection and analysis of malware for smart devices. IEEE Communications Surveys & Tutorials **PP**(99) (November 2013) 1–27
3. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A survey of mobile malware in the wild. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. SPSM '11, New York, NY, USA, ACM (2011) 3–14
4. Zawoad, S., Hasan, R., Haque, M.: Poster: Stuxmob: A situational-aware malware for targeted attack on smart mobile devices (2013)
5. Hasan, R., Saxena, N., Haleviz, T., Zawoad, S., Rinehart, D.: Sensing-enabled channels for hard-to-detect command and control of mobile devices. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ACM (2013) 469–480

6. Raiu, C., Emm, D.: Kaspersky security bulletin. Technical report, Kaspersky (2013) http://media.kaspersky.com/pdf/KSB_2013_EN.pdf.
7. Langner, R.: Stuxnet: Dissecting a cyberwarfare weapon. Security & Privacy, IEEE **9**(3) (2011) 49–51
8. Corporation, S.: Internet security threat report. Technical report, Symantex (2013) http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.
9. Kalige, E., Burkey, D.: A case study of eurograbber: How 36 million euros was stolen via malware. Technical report, Versafe (December 2012)
10. Marquis-Boire, M., Marczak, B., Guarnieri, C., Scott-Railton, J.: You only click twice: Finfishers global proliferation. Research Brief (March 2013) https://citizenlab.org/wp-content/uploads/2013/07/15-2013-youonlyclicktwice.pdf.
11. Rogers, M.: Dendroid malware can take over your camera, record audio, and sneak into google play (March 2014) https://blog.lookout.com/blog/2014/03/06/dendroid/.
12. Capilla, R., Ortiz, O., Hinchey, M.: Context variability for context-aware systems. Computer **47**(2) (2014) 85–87
13. Gianazza, A., Maggi, F., Fattori, A., Cavallaro, L., Zanero, S.: Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. arXiv preprint arXiv:1402.4826 (2014)
14. Chakradeo, S., Reaves, B., Traynor, P., Enck, W.: Mast: Triage for market-scale mobile malware analysis. In: Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks. WiSec '13, New York, NY, USA, ACM (2013) 13–24
15. Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation, USENIX Association (2010) 1–6
16. Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid android: versatile protection for smartphones. In: Proceedings of the 26th Annual Computer Security Applications Conference. (2010) 347–356
17. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the sixth conference on Computer systems. (2011) 301–314
18. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: INFOCOM, 2012 Proceedings IEEE, IEEE (2012) 945–953
19. Zonouz, S., Houmansadr, A., Berthier, R., Borisov, N., Sanders, W.: Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. Computers & Security (2013)
20. Fleck, D., Tokhtabayev, A., Alarif, A., Stavrou, A., Nykodym, T.: Pytrigger: A system to trigger & extract user-activated malware behavior. In: Availability, Reliability and Security (ARES), 2013 Eighth International Conference on, IEEE (2013) 92–101
21. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: Smartdroid: an automatic system for revealing UI-based trigger conditions in Android applications. In: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices. SPSM '12, New York, NY, USA, ACM (2012) 93–104

22. Rastogi, V., Chen, Y., Enck, W.: Appsplayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on Data and application security and privacy. CODASPY '13, New York, NY, USA, ACM (2013) 209–220
23. Jensen, C.S., Prasad, M.R., Møller, A.: Automated testing with targeted event sequence generation. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM (2013) 67–77
24. Liang, C.J.M., Lane, N.D., Brouwers, N., Zhang, L., Karlsson, B., Liu, H., Liu, Y., Tang, J., Shan, X., Chandra, R., et al.: Context virtualizer: A cloud service for automated large-scale mobile app testing under real-world conditions
25. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2013, New York, NY, USA, ACM (2013) 224–234
26. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Proceedings of the 6^{th} European Workshop on System Security (EUROSEC), Prague, Czech Republic (April 2013)
27. Conti, M., Crispo, B., Fernandes, E., Zhauniarovich, Y.: Crepe: A system for enforcing fine-grained context-related policies on android. Information Forensics and Security, IEEE Transactions on **7**(5) (Oct 2012) 1426–1438
28. Norris, J.R.: Markov chains. Number 2008. Cambridge university press (1998)
29. Suarez-Tangil, G., Lobardi, F., Tapiador, J.E., Pietro, R.D.: Thwarting obfuscated malware via differential fault analysis. IEEE Computer (June 2014)
30. Android: Android developers (Visited December 2013) `http://developer.android.com/`.
31. Lantz, P.: Android application sandbox (Visited December 2013) `https://code.google.com/p/droidbox/`.
32. Clauset, A., Shalizi, C.R., Newman, M.E.: Power-law distributions in empirical data. SIAM review **51**(4) (2009) 661–703
33. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Profiledroid: Multi-layer profiling of android applications. In: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking. Mobicom '12, New York, NY, USA, ACM (2012) 137–148
34. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Reviews of modern physics **74**(1) (2002) 47
35. Erdős, P., Rényi, A.: On the evolution of random graphs. Magyar Tud. Akad. Mat. Kutató Int. Közl **5** (1960) 17–61
36. Bertrand, A., David, R., Akimov, A., Junk, P.: Remote administration tool for android devices (Visited December 2013) `https://github.com/DesignativeDave/androrat`.
37. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of piggybacked mobile applications. In: Proceedings of the third ACM conference on Data and application security and privacy, ACM (2013) 185–196
38. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012). (May 2012)