G. Suarez-Tangil, J. E. Tapiador, F. Lombardi and R. D. Pietro, "Alterdroid: Differential Fault Analysis of Obfuscated Smartphone Malware," in IEEE Transactions on Mobile Computing, vol. 15, no. 4, pp. 789-802, 1 April 2016

DOI: 10.1109/TMC.2015.2444847

ALTERDROID: Differential Fault Analysis of Obfuscated Smartphone Malware

Guillermo Suarez-Tangil, Juan E. Tapiador, Flavio Lombardi, Roberto Di Pietro

Abstract—Malware for smartphones has rocketed over the last years. Market operators face the challenge of keeping their stores free from malicious apps, a task that has become increasingly complex as malware developers are progressively using advanced techniques to defeat malware detection tools. One such technique commonly observed in recent malware samples consists of hiding and obfuscating modules containing malicious functionality in places that static analysis tools overlook (e.g., within data objects). In this paper, we describe ALTERDROID, a dynamic analysis approach for detecting such hidden or obfuscated malware components distributed as parts of an app package. The key idea in ALTERDROID consists of analyzing the behavioral differences between the original app and a number of automatically generated versions of it, where a number of modifications (*faults*) have been carefully injected. Observable differences in terms of activities that appear or vanish in the modified app are recorded, and the resulting differential signature is analyzed through a pattern-matching process driven by rules that relate different types of hidden functionalities with patterns found in the signature. A thorough justification and a description of the proposed model are provided. The extensive experimental results obtained by testing ALTERDROID over relevant apps and malware samples support the quality and viability of our proposal.

Index Terms—Computer security, Malware, Mobile computing

1 INTRODUCTION

Smartphones present a number of security and privacy concerns that are, in many respects, even more alarming than those existing in traditional computing environments [1]. Most smartphone platforms are equipped with multiple sensors that can determine user location, gestures, moves and other physical activities, to name a few. Smartphones also feature high-quality audio and video recording capabilities. Sensitive pieces of information that can be captured by these devices could be easily leaked by malware residing on the smartphone. Even apparently harmless capabilities have swiftly turned into a potential menace. For example, access to the accelerometer or the gyroscope can be used to infer the location of screen taps and, therefore, to guess what the user is typing (e.g., passwords or message contents) [2]. Similarly, the Radio Data System (RDS) embedded in most AM/FM channels can be exploited to inject attacks on Software Defined Radio (SDR) systems [3].

A major source of security problems is precisely the ability to incorporate third-party applications from available online markets. Thus, security measures at the market level constitute a primary line of defense [4]. Many market operators carry out a revision process over submitted apps that involves some form of security testing. Official details about such revisions remain unknown, but the constant presence of malware in many markets and recent research studies [5] suggest that operators cannot afford to perform an exhaustive analysis over each app submitted for release to the general public. This is further complicated by the fact that determining which applications are malicious and which are not is still a formidable challenge, particularly for the so-called *grayware*—namely, apps that are not fully malicious but that constitute a threat to the user security and privacy.

• *G. Suarez-Tangil and J.E.Tapiador are with the Dept. of Computer Science, Universidad Carlos III de Madrid, 28911 Leganes, Madrid, Spain. E-mail: guillermo.suarez.tangil@uc3m.es (G. Suarez-Tangil), jestevez@inf.uc3m.es (J.E. Tapiador).*
F. Lombardi is with IAC-CNR, via dei Taurini 19, 00185, Rome, Italy. E-mail: flavio.lombardi@cnr.it (F. Lombardi).
Roberto Di Pietro is with Bell Labs, Cyber Security Research, 91620 Nozay, Paris, France. He is also with Maths Dept. Univ. of Padua, Italy. E-mail: roberto.di_pietro@alcatel-lucent.com (R. Di Pietro)

1.1 Obfuscated Smartphone Malware

The rapid growth of smartphone sales has come hand in hand with a similar increase in the number and sophistication of malicious software targeting these platforms. For example, according to the mobile threat report published by Juniper Networks in 2012, the number of unique malware variants for Android increased by 3325.5% during 2011 and by 614% between 2012 and 2013 [6]. Smartphone malware has become a rather profitable business due to the existence of a large number of potential targets and the availability of reuse-oriented malware development methodologies that make exceedingly easy to produce new samples.

Malware analysis is a thriving research area with a substantial amount of still unsolved problems [7], [6], [8]. In the case of smartphones, the impressive growth both in malware and benign apps is making increasingly unaffordable any human-driven analysis of potentially dangerous apps. This has consolidated the need for intelligent analysis techniques to aid malware analysts in

their daily functions. Furthermore, smartphone malware is becoming increasingly stealthy [9] and recent specimes are relying on advanced code obfuscation techniques to evade detection by security analysts [10]. For instance, *DroidKungFu* has been one of the major Android malware outbreaks. It started on June 2011 and has already at least six known different variants. It has been mostly distributed through official or alternative markets by piggybacking the malicious payload into a variety of legitimate applications. Such a payload is encrypted into the app's assets folder and decrypted at runtime using a key stored in a local variable and located at one class. Another remarkable example is *GingerMaster*, the first malware using root exploits for privilege escalation on Android 2.3. The main payload was stored as PNG and JPEG pictures in the assets file, which were interpreted as code once loaded by a small hook within the app.

More sophisticated obfuscation techniques, particularly in code, are starting to materialize (e.g., stego-malware [11]). These techniques and trends create an additional obstacle to malware analysts, who see their task further complicated and have to ultimately rely on carefully controlled dynamic analysis techniques to detect the presence of potentially dangerous pieces of code.

1.2 Overview and Contributions

In this paper we describe ALTERDROID, a tool for detecting, through reverse engineering, obfuscated functionality in components distributed as parts of an app package. Such components are often part of a malicious app and are hidden outside its main code components (e.g. within data objects), as code components may be subject to static analysis by market operators. The key idea in ALTERDROID consists of analyzing the behavioral differences between the original app and an altered version where a number of modifications (*faults*) have been carefully introduced. Such modifications are designed to have no observable effect on the app execution, provided that the altered component is actually what it should be (i.e., it does not hide any unwanted functionality). For example, replacing the value of some pixels in a picture or a few characters in a string encoding an error message should not affect the execution. However, if after doing so it is observed that a dynamic class loading action crashes or a network connection does not take place, it may well be that the picture was actually a piece of code or the string a network address or a URL.

At high level, ALTERDROID has two differentiated major components: fault injection and differential analysis. The first one takes a candidate app—the entire package—as input and generates a fault-injected one. This is done by first extracting all components in the app and then identifying those suspicious of containing obfuscated functionality. Such an identification is done on an anomaly-detection basis by comparing specific statistical features of the component's contents with a predefined model for each possible type of resource (i.e., code, pictures and video, text files, databases, etc.). Faults are then injected into candidate components, which are subsequently repackaged, together with the unaltered ones, into a new app. This process admits simultaneous injection of different faults into different components and it is driven by a search algorithm that attempts to identify where the obfuscated functionality is hidden. Both the original and the fault-injected apps are then executed under identical conditions (i.e., context and user inputs), and their behavior is monitored and recorded in the form of two behavioral signatures. Such signatures are merely sequential traces of the activities executed by the app, such as for example opening a network connection, sending or receiving data, loading a dynamic component, sending an SMS, interacting with the file system, etc. Both behavioral signatures are then treated as in a string-to-string correction problem, in such a way that computing the Levenshtein (edit) distance between them returns the list of observable differences in terms of insertions, deletions, and substitutions. Such a list, called the differential signature, is finally matched against a rule-set where each rule encodes a relationship between the type of presumably hidden functionality and certain patterns in the differential signature.

Our prototype implementation of ALTERDROID builds on a number of Android open source tools that facilitate tasks such as extracting components [12], repackaging them back into an app [13], and analyzing dynamic behavior [14]. The present ALTERDROID base platform does not have a fully comprehensive set of fault injection operators and differential rules. In fact, ALTERDROID is designed and built to allow ease of tailoring and flexibility in functionality addition. Required extensions depend on the kind of usage the proposed system is built for. In order to build a production system, of course, the entire set of possible operators has to be created. However, this is out of the scope of present paper, aimed at showing and discussing benefits and limitations of the proposed approach rather than proving its completeness of suitability for production usage in its present shape.

The main contributions of this paper can be summarized in what follows:

- We introduce the notion of differential fault analysis for detecting obfuscated malware functionality in smartphone apps.
- We provide simple yet powerful enough models for fault injection operators, behavioral signatures and rule-based analysis of differential behavior.
- We describe the functional components of ALTERDROID, a prototype implementation of our differential fault analysis model for Android apps. The system includes instantiations for key tasks such as identifying components to be fault-injected and a search-based approach to track down obfuscated components in an app. Moreover, ALTERDROID's functional architecture supports distributed deployment of different modules, which allows running

various analysis tasks in parallel and also potentially offloading them to the cloud.
- We illustrate our approach by providing a step-by-step analysis of three relevant Android malware samples that incorporate hidden functionality in repackaged apps: `DroidKungFu`, `AnserverBot`, and `GingerMaster`.
- We evaluate the performance of our approach over a number of malware samples found in the wild. Specifically, we use ALTERDROID to analyze around 10000 apps from a malware repository (VirusShare), an unofficial Android Market (Aptoide), and Google Play (GP).
- Finally, we provide an open-source version of ALTERDROID[1] to foster further research in automated tools for advanced smartphone malware analysis.

The rest of this paper is organized as follows. In Section 2 we introduce the formal models for fault injection and differential analysis. Section 3 describes ALTERDROID's architecture and its key functional components, and provides an overview of our proof-of-concept implementation. Subsequently in Section 4 we discuss the analysis of three Android malware samples with ALTERDROID and present a performance evaluation. Section 5 provides an overview of related work in this area and compares ALTERDROID to other proposals targeting the problem of Android malware detection. Finally, in Section 6 we conclude the paper by summarizing our main contributions and discussing limitations and directions for future research.

2 A Differential Fault Analysis Model

This section introduces the theoretical background used in ALTERDROID to:

- inject faults into apps;
- represent behavioral differences between apps;
- deduce properties from such behavioral differences considering injected faults and observed differences.

The overall dynamics of the differential fault analysis process (i.e., the mechanism governing which faults are injected and where) is external to this model and will be discussed in Section 3.

2.1 Fault Injection Model

An app $\mathcal{P}$ can be seen as a collection of components

$$\mathcal{P} = \{c_1, c_2, \ldots, c_k\}. \tag{1}$$

A component can be composed of a number of classes (i.e., code), but also other resources that are dynamically accessed, such as for example asset files. Components have a type, such as for example code, picture, video, database, etc. A type function $\tau(c)$ can be defined that returns the type of component c.

1. Code and documentation can be downloaded from http://www.seg.inf.uc3m.es/~guillermo-suarez-tangil/Alterdroid/

Fault conditions can be injected into an app by altering one or more of its components. If $\mathbb{C}$ is the set of all possible app components, a Fault Injection Operator (FIO) is a transformation

$$\begin{aligned} \Psi^{c_i} &: 2^{\mathbb{C}} \rightarrow 2^{\mathbb{C}} \\ \Psi^{c_i}(\mathcal{P}) &= \mathcal{P} \setminus \{c_i\} \cup \{\Psi(c_i)\}. \end{aligned} \tag{2}$$

That is, $\Psi^{c_i}(\mathcal{P})$ returns a modified version of $\mathcal{P}$ where component c_i has been replaced by $\Psi(c_i)$. Depending on the functionality of c and on the nature of the modifications introduced by Ψ, replacing c by $\Psi(c)$ may (or may not) translate into observable differences in the execution of $\mathcal{P}$.

In this paper, we restrict ourselves to FIOs that make alterations to data components only, not to instructions. Data components include the value of variables found in the code and also asset files such as databases, pictures, and audio and video files. We will abuse notation and write $\tau(\Psi^{c_i})$ for $\tau(c_i)$; i.e., we consider that the type of a FIO is the type of all components it can be applied to.

FIOs can be arbitrarily complex and, in some cases, their operation may depend on the type and/or current value of the component to be altered. However, some simple FIOs treat components as bit strings, such as for example:

- $\text{rrep}^c(\cdot)$: replaces the value of component c for a randomly chosen bit string.
- $\text{zero}^c(\cdot)$: replaces the value of component c for a string of zeros of the same length.
- $\text{rmut}^c_j(\cdot)$: flips the j-th bit of of component c.

The above FIOs are rather generic. In some cases, we might want to define datatype-specific operators. These will allow modifying specific data objects (e.g., multimedia files) in a syntax-preserving way, when the focus is on changing the content without rendering the object unusable.

2.2 Modeling Differential Behavior

A key task in our system is the analysis of the behavioral differences between an original app and a slightly modified version of it after applying a FIO. We next introduce a model to represent traces of activities and differences between such traces.

2.2.1 Behavioral Signatures

An app interacts with the platform where it is executed by requesting services through a number of system calls. These define an interface for apps that need to read/write files, send/receive data through the network, place a phone call, etc. Rather than focusing on low-level system calls, in this paper we will describe an app's behavior through the *activities* it executes (see also [15]). In some cases, there will be a one-to-one correspondence between a behavioral activity and a system call, while in others a behavioral activity will encompass a sequence of

system calls executed in a given order. In what follows, we assume that

$$\mathbb{A} = \{a_1, a_2, \ldots, a_n\} \tag{3}$$

is a set of all relevant and observable activities an app can execute.

The execution flow of an app $\mathcal{P}$ may follow different paths depending on its inputs. We group such inputs into two main classes:

- A sequence $\mathbf{u}$ of user-provided inputs, such as for example those acquired through the touchscreen.
- A sequence $\mathbf{t}$ of contexts, defining the state of the environment when the execution takes place. Each context (state) is represented by a set of variables that provide the app with information such as current location, time, energy level, temperature, etc.

We will denote by $\mathcal{P}(\mathbf{u}|\mathbf{t})$ the execution of $\mathcal{P}$ with user inputs $\mathbf{u}$ in context $\mathbf{t}$.

The observable behavior resulting from the execution of $\mathcal{P}(\mathbf{u}|\mathbf{t})$ is summarized in a *behavioral signature* $\sigma[\mathcal{P}(\mathbf{u}|\mathbf{t})]$, this being a time series given by

$$\sigma[\mathcal{P}(\mathbf{u}|\mathbf{t})] = \langle s_1, s_2, \ldots, s_n \rangle, \quad s_i \in \mathbb{A}. \tag{4}$$

Notice that the adopted signature model does not take into account the duration of each behavioral activity or the time elapsed between (each two of) them, but only their relative ordering. We will abuse notation and omit the associated app and its inputs when it is irrelevant or clear from context.

Finally, we will denote by $\text{len}(\sigma)$ the length of signature σ, defined as the number of activities in the series.

2.2.2 Differential Signatures

We are interested in analyzing the differences between two observed behaviors given by their respective behavioral signatures. We approach this problem as one of string-to-string correction, where differences are respresented as the minimum number of edit operations needed to transform one signature into the other. Given a behavioral signature $\sigma = \langle s_1, s_2, \ldots, s_n \rangle$, we define the next three families of signature transformation operators (STO) for all $a \in \mathbb{A}$ and $i \in [1, n]$:

- $\mathsf{Ins}_i^a(\sigma) = \langle s_1, \ldots, s_i, a, s_{i+1}, \ldots, s_n \rangle$
- $\mathsf{Del}_i^a(\sigma) = \langle s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n \rangle$
- $\mathsf{Sub}_i^a(\sigma) = \langle s_1, \ldots, s_{i-1}, a, s_{i+1}, \ldots, s_n \rangle$

Let

$$\mathbb{O} = \bigcup_{i,a} \left(\mathsf{Ins}_i^a \cup \mathsf{Del}_i^a \cup \mathsf{Sub}_i^a \right) \tag{5}$$

be the set of all possible STOs. Given two behavioral signatures σ_1 and σ_2, we define the *differential signature* $\Delta(\sigma_1, \sigma_2)$ as an ordered sequence of STOs

$$\Delta(\sigma_1, \sigma_2) = \langle o_1, o_2, \ldots, o_k \rangle \quad o_i \in \mathbb{O} \tag{6}$$

such that

$$o_k \circ o_{k-1} \circ \cdots \circ o_1(\sigma_1) = \sigma_2, \tag{7}$$

where $o_i \circ o_j$ denotes de composition of STOs o_i and o_j. In other words, the differential signature $\Delta(\sigma_1, \sigma_2)$ provides a sequence of insertions, deletions, and substitutions that transforms σ_1 into σ_2. Notice that, in general, $\Delta(\sigma_1, \sigma_2) \neq \Delta(\sigma_2, \sigma_1)$.

For the purposes of this work, we are interested in minimal differential signatures, i.e., sequences of minimum length. The most straightforward way to compute the minimal differential signature is by computing the Levenshtein distance [16] (also known as edit distance) between σ_1 and σ_2, assuming that all operators have equal cost [17]. This computation returns not only the distance, but also the optimal differential signature.

2.3 Analyzing Differential Signatures

Let

$$\mathcal{P}' = \Psi(\mathcal{P}) = \Psi_r^{c_r} \circ \Psi_{r-1}^{c_r}{}^{1} \circ \cdots \circ \Psi_1^{c_1}(\mathcal{P}) \tag{8}$$

be the app resulting after the sequential application of FIOs $\Psi_1, \ldots, \Psi_r$ to components $c_1, \ldots, c_r$ of app $\mathcal{P}$. Let $\sigma[\mathcal{P}]$ and $\sigma[\Psi(\mathcal{P})]$ be the behavioral signatures obtained after executing $\mathcal{P}$ and $\Psi(\mathcal{P})$ under the same conditions[2], and let $\Delta(\sigma[\mathcal{P}], \sigma[\Psi(\mathcal{P})])$ be their differential signature. The analysis model used in this paper is based on deducing properties of $\mathcal{P}$ from the presence or absence of certain *patterns* in $\Delta(\sigma[\mathcal{P}], \sigma[\Psi(\mathcal{P})])$ and the properties of the FIO Ψ. We next describe these two elements in detail.

2.3.1 FIO Classes

We identify two broad classes of FIOs:

- A FIO Ψ^{c_i} is said to be *indistinguishable* if $\Delta(\sigma[\mathcal{P}], \sigma[\Psi^{c_i}(\mathcal{P})]) = \emptyset$ for all apps $\mathcal{P}$ containing component c_i. In other words, a FIO is indistinguishable if it does not affect the execution flow of any app and, therefore, the behavioral signatures before and after applying it coincide.
- A FIO Ψ^{c_i} is said to be *distinguishable* if $\Delta(\sigma[\mathcal{P}], \sigma[\Psi^{c_i}(\mathcal{P})]) \neq \emptyset$ for all apps $\mathcal{P}$ containing component c_i. Thus, distinguishable FIOs always manifest as nonempty differential signatures.

In what follows, the predicate $\text{ind}(\Psi^{c_i})$ models this property:

$$\text{ind}(\Psi^{c_i}) = \begin{cases} \texttt{true} & \text{if } \Psi^{c_i} \text{ is indistinguishable} \\ \texttt{false} & \text{otherwise} \end{cases} \tag{9}$$

2.3.2 Properties of Differential Signatures

Patterns in differential signatures are modeled as first-order logical predicates upon which Boolean formulae can be defined. Thus, analyzing a differential signature reduces to evaluating a number of Boolean formulae linked to properties of the app and the FIO, i.e.,

$$\mathcal{P} \text{ has property } x \iff \Phi_x\big(\Psi, \Delta(\sigma[\mathcal{P}], \sigma[\Psi(\mathcal{P})])\big) = \texttt{true}. \tag{10}$$

2. That is, the same sequence of user inputs and contexts.

We consider two basic predicates:

- $\mathsf{equal}(\Delta_1, \Delta_2) = \texttt{true}$ iff $\Delta_1 = \Delta_2$, where Δ_1 and Δ_2 are differential signatures. Notice that the empty set is a valid differential signature.
- $\mathsf{contains}(\Delta, o) = \texttt{true}$ iff $\Delta = \langle o_1, o_2, \ldots, o_k \rangle$ and $\exists o_j \in \Delta$ such that $o_j = o$.

Standard symbols will be used for Boolean formulae, including quantifiers ($\exists$, $\forall$), negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$).

2.3.3 Examples

We next illustrate the concepts introduced above through a number of examples.

Example 1. Assume that c_{icon} is an icon image used by an app $\mathcal{P}$ in the GUI. Modifying some pixels of such an icon, or even replacing it by another valid icon should not affect at all the execution flow of $\mathcal{P}$. If nonetheless the icon is replaced and the modified app behaves differently from the original app under exactly the same conditions, it can be deduced that the original icon contained some *functionality*, for instance a piece of compiled code masqueraded as an icon. This intuition can be generalized through the following rule (hidden functionality in component, or HFC):

$$\mathbf{R_{HFC}} : c \in \mathcal{P} \text{ contains hidden functionality} \iff \mathsf{ind}(\Psi^c) \wedge \neg\mathsf{equal}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(\mathcal{P})]), \emptyset\big),$$

where $\Psi^c(P)$ is the FIO that replaces icon c in $\mathcal{P}$ by another valid icon.

Example 2. A more specific case of the situation discussed in the previous example occurs when modifying a component c results in the absence of a dynamic loading action, which is used to load code pieces into memory. In such a case, it may be possible that c contains hidden code that is dynamically loaded. The following rule captures this:

$$\mathbf{R_{CDC}} : c \in \mathcal{P} \text{ contains dynamic code} \iff \mathsf{ind}(\Psi^c) \wedge \exists\, i : \mathsf{contains}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(\mathcal{P})]), \mathsf{Del}_i^{dex\_load}\big).$$

Example 3. Let v be a variable such that its content should have no influence on the program flow. For example, v could be a string containing an error message that may be displayed at some point. Such strings have been broadly used in existing malware to hide URLs that point to services from where the malware can download further code, receive instructions, send data, etc. To avoid detection, the string is often obfuscated and the URL is only revealed at execution time after applying some transformation. Thus, any modification of the string such that the URL is damaged will likely result on the impossibility of establishing a connection. The following rule captures this intuition:

$$\mathbf{R_{URL}} : v \in \mathcal{P} \text{ contains an URL} \iff \mathsf{ind}(\Psi^v) \wedge \exists\, i : \mathsf{contains}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^v(\mathcal{P})]), \mathsf{Del}_i^{net}\big).$$

Example 4. Similarly to the cases discussed above, it may be possible to find out whether a component c leaks information from a number of sensors (e.g., accelerometer, GPS, etc.) if, after modifying it, the differential signature lacks an access to such a sensor and a network connection:

$$\mathbf{R_{SDL}} : c \in \mathcal{P} \text{ leaks sensor data} \iff \mathsf{ind}(\Psi^c) \wedge \Big(\exists\, i_1 : \mathsf{contains}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(\mathcal{P})]), \mathsf{Del}_{i_1}^{acc}\big) \vee \exists\, i_2 : \mathsf{contains}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(\mathcal{P})]), \mathsf{Del}_{i_2}^{gps}\big) \vee \vdots \Big) \wedge \exists\, j : \mathsf{contains}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(\mathcal{P})]), \mathsf{Del}_j^{net}\big).$$

3 ALTERDROID: DIFFERENTIAL FAULT ANALYSIS OF OBFUSCATED APPS

We next describe ALTERDROID, our approach to studying obfuscated malware code based on the differential fault analysis model discussed above. The high level architecture of ALTERDROID is shown in Fig. 1 (see [18]). There are two differentiated major blocks:

1) The first one generates a number of fault-injected apps. This process is carried out by first extracting all app components and identifying those of interest (ColS[3]), i.e., those components suspicious of containing hidden functionality. An iterative process then selects candidate Cols and injects faults into them. Both modified and unmodified components are then repackaged together into a new app[4].
2) The second block generates stimuli (user inputs and context) for both apps and executes them, generating a pair of behavioral signatures. The differential signature is then computed and matched against a database of patterns to identify the presence of hidden functionality.

We next provide a detailed description of the key modules of ALTERDROID and the current prototype implementation.

3.1 Identifying Components of Interest

The first step in the analysis of an app is identifying *components of interest* (Cols), i.e., parts of an app suspicious of containing hidden functionality. Such components will be later fault injected according to some strategy in order to analyze the resulting behavior.

We say that a component c of type $\tau(c)$ in an app $\mathcal{P}$ is of interest if it does not fit a model $\mathcal{M}_{\tau(c)}$ defined for all components of type $\tau(c)$. In our current version of ALTERDROID, models measure statistical features only, such as for example the expected entropy, the byte distribution, or the average size. Such features are computed from a dataset of components of the same type, such

3. We denote as ColS (with capital S) the set of Components of Interest Cols

4. Note this does not require the source code of the app to be available

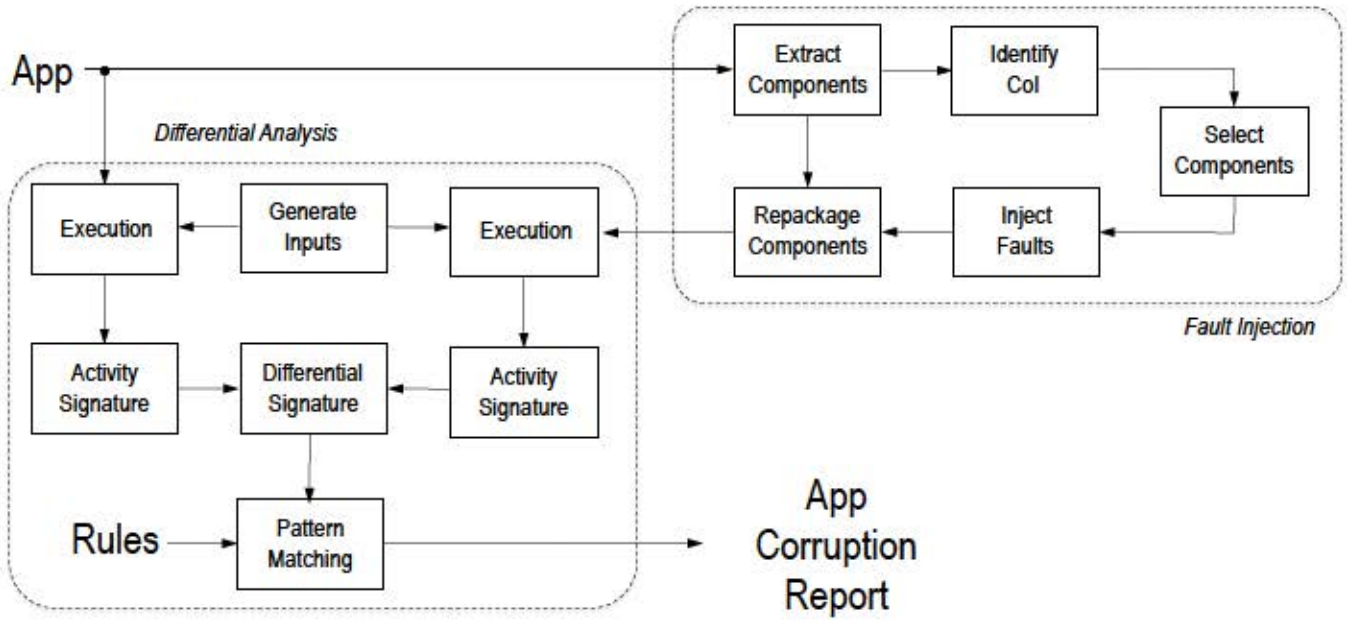

Fig. 1. ALTERDROID architecture.

as text files, pictures, code, etc. For each model $\mathcal{M}$, we assume a Boolean function $\text{test}(c, \mathcal{M})$ that returns `true` if c complies with $\mathcal{M}$, and `false` otherwise. For example, if $\mathcal{M}$ is a byte distribution, then $\text{test}(c, \mathcal{M})$ could be a goodness-of-fit test (e.g., χ^2) between $\mathcal{M}$ and c's byte distribution. More formally

$$c \in \text{CoIS}(\mathcal{P}) \iff \text{test}(c, \mathcal{M}_{\tau(c)}) = \texttt{false}. \quad (11)$$

In our experience, such simple models suffice to spot the most common—and rather simple—obfuscation methods observed in smartphone malware, including code camouflaged as supplementary multimedia files, connection data hidden in text variables, etc.

ALTERDROID also supports an exhaustive analysis mode in which some additional components may be considered CoIs even if they comply with their type model. In this mode, a component is considered CoI if it is CoI as defined above, or if there exists an indistinguishable operator for it. Formally

$$c \in \text{CoIS}(\mathcal{P}) \iff \big(\text{test}(c, \mathcal{M}_{\tau(c)}) = \texttt{false}\big) \text{ or } \big(\exists\, \Psi^c \,:\, \text{ind}(\Psi^c)\big). \quad (12)$$

The rationale for including this mode is to also check components for which we know in advance that alterations do not translate into noticeable differences. This is very useful for detecting more sophisticated obfuscation methods that try to evade detection by carefully modifying the code so that it fits the statistical model of the component. As a side effect, however, the exhaustive analysis mode may end up with a large set of CoIs (CoIS).

The algorithm shown in Fig. 2 describes the process discussed above to identify the CoIS in ALTERDROID.

Input:
App: $\mathcal{P} = \{c_1, c_2, \ldots, c_k\}$
Set of type normality models: $\{\mathcal{M}_1, \mathcal{M}_2, \cdots, \mathcal{M}_n\}$
Set of FIOs: $\{\Psi_1, \Psi_2, \cdots, \Psi_m\}$
Mode: `normal` / `exhaustive`
Procedure:
1. CoIS $\leftarrow \emptyset$
2. **For each** $c \in \mathcal{P}$ **do**
3. **if** [$\text{test}(c, \mathcal{M}_{\tau(c)}) = \texttt{false}$] or [(mode = `exhaustive`) and $(\exists\, \Psi_i : \tau(\Psi_i) = \tau(c))$] **then**
4. CoIS $\leftarrow$ CoIS $\cup \{c\}$
5. **return** CoIS

Fig. 2. Algorithm for obtaining CoIS from an app.

3.2 Generating Fault-injected Apps

Components of interests identified in the previous stage are injected with faults and reassembled, together with the remaining app components, to generate a faulty app $\mathcal{P}'$. This process can generate several fault-injected apps, as there are multiple ways of applying different FIOs to different components in the set of CoIs. In ALTERDROID, fault-injected apps are generated one at a time and sent for differential analysis. If no evidence of malicious behavior is found in the differential analysis, the fault injection process is invoked again to generate a different faulty app, and so on.

Assume that $\text{CoIS} = \{c_1, \ldots, c_n\}$ and that for each $c_i \in \text{CoIS}$ there is a set of FIOs $\mathcal{F}_i = \{\Psi_{i_1}^{c_i}, \ldots, \Psi_{i_{m_i}}^{c_i}\}$ that can be applied to c_i. (Recall that FIOs can be quite specific and, therefore, not all FIOs are applicable to all components.) All possible fault-injected apps can be generated by a naïve strategy that applies each FIO to each component one at a time, producing the sequence of apps

$$\Psi_{i_1}^{c_1}(\mathcal{P}), \ldots, \Psi_{i_{m_1}}^{c_1}(\mathcal{P}), \ldots, \Psi_{i_1}^{c_n}(\mathcal{P}), \ldots, \Psi_{i_{m_n}}^{c_n}(\mathcal{P}). \quad (13)$$

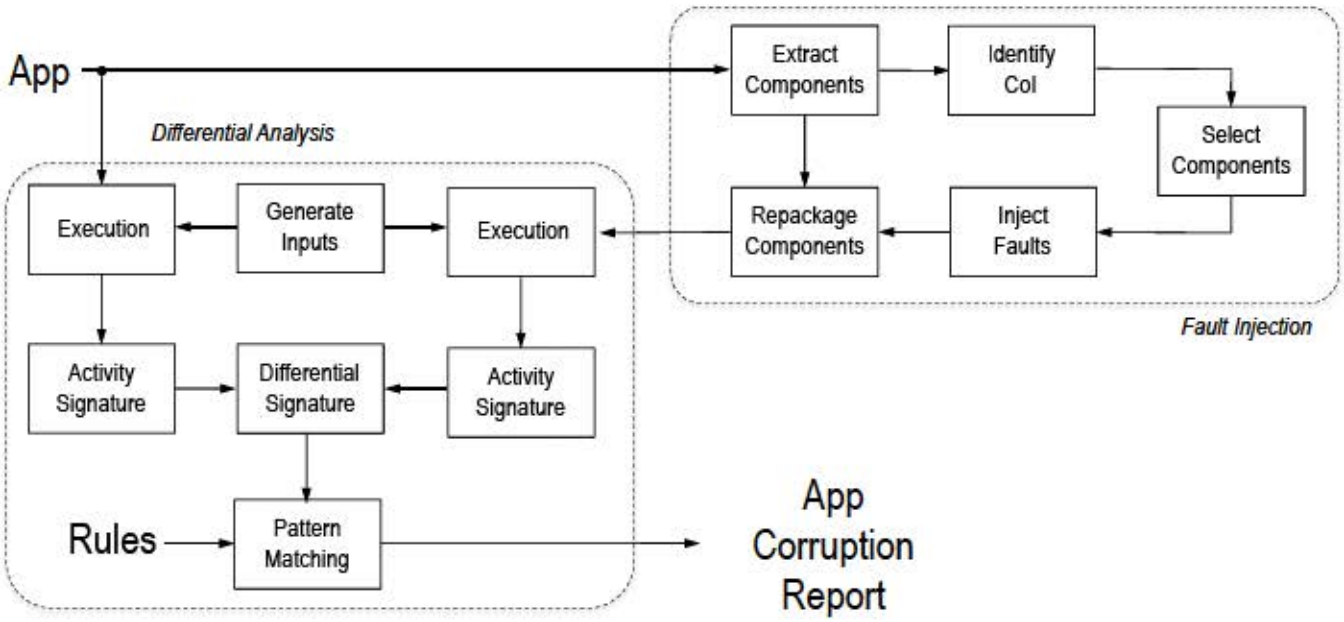

Fig. 1. ALTERDROID architecture.

as text files, pictures, code, etc. For each model $\mathcal{M}$, we assume a Boolean function test$(c, \mathcal{M})$ that returns `true` if c complies with $\mathcal{M}$, and `false` otherwise. For example, if $\mathcal{M}$ is a byte distribution, then test$(c, \mathcal{M})$ could be a goodness-of-fit test (e.g., χ^2) between $\mathcal{M}$ and c's byte distribution. More formally

$$c \in \mathsf{ColS}(\mathcal{P}) \iff \mathsf{test}(c, \mathcal{M}_{\tau(c)}) = \texttt{false}. \quad (11)$$

In our experience, such simple models suffice to spot the most common—and rather simple—obfuscation methods observed in smartphone malware, including code camouflaged as supplementary multimedia files, connection data hidden in text variables, etc.

ALTERDROID also supports an exhaustive analysis mode in which some additional components may be considered CoIs even if they comply with their type model. In this mode, a component is considered CoI if it is CoI as defined above, or if there exists an indistinguishable operator for it. Formally

$$c \in \mathsf{ColS}(\mathcal{P}) \iff \big(\mathsf{test}(c, \mathcal{M}_{\tau(c)}) = \texttt{false}\big) \text{ or } \big(\exists\ \Psi^c\ :\ \mathsf{ind}(\Psi^c)\big). \quad (12)$$

The rationale for including this mode is to also check components for which we know in advance that alterations do not translate into noticeable differences. This is very useful for detecting more sophisticated obfuscation methods that try to evade detection by carefully modifying the code so that it fits the statistical model of the component. As a side effect, however, the exhaustive analysis mode may end up with a large set of CoIs (ColS).

The algorithm shown in Fig. 2 describes the process discussed above to identify the ColS in ALTERDROID.

Input:
App: $\mathcal{P} = \{c_1, c_2, \ldots, c_k\}$
Set of type normality models: $\{\mathcal{M}_1, \mathcal{M}_2, \cdots, \mathcal{M}_n\}$
Set of FIOs: $\{\Psi_1, \Psi_2, \cdots, \Psi_m\}$
Mode: `normal` / `exhaustive`
Procedure:
1. ColS $\leftarrow \emptyset$
2. **For each** $c \in \mathcal{P}$ **do**
3. **if** [test$(c, \mathcal{M}_{\tau(c)}) =$ `false`] or [(mode = `exhaustive`) and $(\exists\ \Psi_i\ :\ \tau(\Psi_i) = \tau(c))$] **then**
4. ColS $\leftarrow$ ColS $\cup \{c\}$
5. **return** ColS

Fig. 2. Algorithm for obtaining ColS from an app.

3.2 Generating Fault-injected Apps

Components of interests identified in the previous stage are injected with faults and reassembled, together with the remaining app components, to generate a faulty app $\mathcal{P}'$. This process can generate several fault-injected apps, as there are multiple ways of applying different FIOs to different components in the set of CoIs. In ALTERDROID, fault-injected apps are generated one at a time and sent for differential analysis. If no evidence of malicious behavior is found in the differential analysis, the fault injection process is invoked again to generate a different faulty app, and so on.

Assume that ColS $= \{c_1, \ldots, c_n\}$ and that for each $c_i \in$ ColS there is a set of FIOs $\mathcal{F}_i = \{\Psi_{i_1}^{c_i}, \ldots, \Psi_{i_{m_i}}^{c_i}\}$ that can be applied to c_i. (Recall that FIOs can be quite specific and, therefore, not all FIOs are applicable to all components.) All possible fault-injected apps can be generated by a naïve strategy that applies each FIO to each component one at a time, producing the sequence of apps

$$\Psi_{i_1}^{c_1}(\mathcal{P}), \ldots, \Psi_{i_{m_1}}^{c_1}(\mathcal{P}), \ldots, \Psi_{i_1}^{c_n}(\mathcal{P}), \ldots, \Psi_{i_{m_n}}^{c_n}(\mathcal{P}). \quad (13)$$

Input:
App: $\mathcal{P}$
ColS $= \{c_1, c_2, \ldots, c_n\}$
Set of FIOs: $\mathcal{F} = \{\Psi_1, \Psi_2, \cdots, \Psi_m\}$
Procedure:
1. maliciousComp $\leftarrow$ null
2. **For each** FIO Ψ_j **do**
3. $\quad \mathcal{P}'_j \leftarrow \mathcal{P}$
4. $\quad$ **For each** $c_i \in$ ColS **do**
5. $\quad\quad$ **if** Ψ_j is applicable to c_i **then**
6. $\quad\quad\quad \mathcal{P}'_j = \Psi_j^{c_i}(\mathcal{P})$
7. $\quad$ **if** DiffAnalysis$(\mathcal{P}, \mathcal{P}', \Psi_j) \neq \emptyset$ **then**
8. $\quad\quad$ maliciousComp $\leftarrow$ SearchComponent$(\Psi_j, \mathcal{P}, \text{ColS}, 1, n)$
9. **return** maliciousComp

Function SearchComponent$(\Psi_j, \mathcal{P}, \text{ColS}, min, max)$
1. $\mathcal{P}'_j \leftarrow \mathcal{P}$
2. **For** $i = min$ **to** max **do**
3. $\quad$ **if** Ψ_j is applicable to c_i **then**
4. $\quad\quad \mathcal{P}'_j = \Psi_j^{c_i}(\mathcal{P})$
5. **if** DiffAnalysis$(\mathcal{P}, \mathcal{P}', \Psi_j) \neq \emptyset$ **then**
6. $\quad$ **if** $min = max$ **then**
7. $\quad\quad$ **return** c_{min}
8. **else**
9. $\quad\quad$ SearchComponent$(\Psi_j, \mathcal{P}, \text{ColS}, min, (max - min)/2)$
10. $\quad\quad$ SearchComponent$(\Psi_j, \mathcal{P}, \text{ColS}, (max - min)/2), max)$

Fig. 3. Algorithm for injecting faults and searching for malicious components after differential analysis.

Thus, there are $\sum_{j=1}^{n} m_j$ possible fault-injected apps, one for each possible component-FIO pair.

Although ALTERDROID implements several distinguishable FIOs, all FIOs tested in our experiments are indistinguishable. This allows for a more efficient fault injection process based on the fact that the composition of indistinguishable FIOs is an indistinguishable FIO. Consequently, if the same FIO is applied to multiple components and there is hidden functionality in just one of them, the resulting app will behave exactly as if just the malicious component would have been fault injected. The resulting fault injection process is as follows:

1) For each FIO Ψ_j, generate $\mathcal{P}'_j$ by applying it to all $c_i \in$ ColS

$$\mathcal{P}'_j = \mathbf{\Psi}_j(\mathcal{P}) = \Psi_{i_j}^{c_1} \circ \Psi_{i_j}^{c_2} \circ \cdots \circ \Psi_{i_j}^{c_n}(\mathcal{P}), \qquad (14)$$

where $\Psi_j^{c_i}$ is the identity operator if Ψ_j is not applicable to c_i. The resulting P'_j is sent for differential analysis with respect to the original $\mathcal{P}$.

2) If there is one $\mathcal{P}'_j$ such that the differential analysis spots malicious behavior, the component responsible for it can be identified by searching over all $c_i \in$ ColS with just the corresponding FIO Ψ_j. This process can be done in logarithmic time by ordering all components and recursively applying Ψ_j to half of them, rather than in linear time by just applying Ψ_j to each $c_i \in$ ColS in turn.

The overall process, which is entwined with the differential analysis stage discussed later, is summarized in the algorithm shown in Fig. 3. Notice that in this description the process stops when just one malicious component is identified. Extending the algorithm so that it searches for all of them is straightforward.

Input:
Apps: $\mathcal{P}$ and $\mathcal{P}'$
FIO Ψ
Set of rules: $\mathcal{R} = \{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_p\}$

Procedure:
1. $(\mathbf{u}, \mathbf{t}) \leftarrow$ GenUsagePatterns$(\mathcal{P})$
2. $\sigma \leftarrow$ GenBehavioralSig$(\mathcal{P}, \mathbf{u}, \mathbf{t})$
3. $\sigma' \leftarrow$ GenBehavioralSig$(\mathcal{P}', \mathbf{u}, \mathbf{t})$
4. $\Delta(\sigma, \sigma') \leftarrow$ ComputeDiffSig(σ, σ')
5. matchingRules $\leftarrow \emptyset$
6. **For each** $\mathbf{R}_i \in \mathcal{R}$ **do**
7. $\quad$ **if** match$(\mathbf{R}_i, \Psi, \Delta(\sigma, \sigma'))$ **then**
8. $\quad\quad$ matchingRules $\leftarrow$ matchingRules $\cup \{\mathbf{R}_i\}$
9. $\quad$ **end-if**
10. **end-for**
11. **return** matchingRules

Fig. 4. Algorithm DiffAnalysis for generating differential signatures and identifying matching rules.

3.3 Applying Differential Analysis

Differential analysis between a candidate fault-injected app and the original app is carried out following the model described in Sections 2.2 and 2.3. The process comprises the following steps:

1) Generate an appropriate usage pattern $\mathbf{u}$ and context $\mathbf{t}$ [19], [20] to feed both apps and extract their behavioral signatures, $\sigma[\mathcal{P}(\mathbf{u}|\mathbf{t})]$ and $\sigma[\mathcal{P}'(\mathbf{u}|\mathbf{t})]$. Both the original and the fault-injected app are tested under the same conditions and using the same inputs. Note that this assumes that the execution of an app is completely deterministic.
2) Generate the differential signature $\Delta(\sigma[\mathcal{P}(\mathbf{u}|\mathbf{t})], \sigma[\mathcal{P}'(\mathbf{u}|\mathbf{t})])$ from the behavioral signatures obtained above.
3) Apply sequentially all rules $\mathbf{R}_i$ over $\Delta(\sigma[\mathcal{P}(\mathbf{u}|\mathbf{t})], \sigma[\mathcal{P}'(\mathbf{u}|\mathbf{t})])$ and return those for which a match is obtained.

The process is summarized in the algorithm in Fig. 4.

3.4 Implementation

We next describe our prototype implementation of ALTERDROID, including the currently available operators for extracting the components of interest, generating fault injected apps, and a rule-set used for differential signature matching.

3.4.1 *Prototype Implementation*

ALTERDROID is implemented using Java and Python components and relies on a number of Android open source tools for specific tasks. App components are extracted using `Androguard` [12]. After fault injection, components are repackaged into a modified app using `ApkTool` [13]. `Monkey` [21] is used to generate a common sequence of events to interact with both the original app and the fault-injected app. These events should be generated specifically for each test to intelligently drive the GUI exploration [19], [20], i.e., to test code implementing different functionalities of the app. In its

current version, ALTERDROID uses `Monkey` to generate 5 classes of input events: *activity launch*, *service launch*, *action buttons*, *screen touch*, and *text input*.

Each app is then executed in a controlled environment using the stream of events generated above. For this purpose, we use `Droidbox` [22], a sandbox that allows monitoring various features related to the execution during a fixed, user-given amount of time. In order to generate behavioral signatures, ALTERDROID monitors the execution of 11 different activities:

- *crypto*: generated when calls to the cryptographic API are invoked;
- *net-open*, *net-read*, *net-write*: associated with network I/O actitivites (opening a connection, receiving, and sending data);
- *file-open*, *file-read*, *file-write*: associated with file system I/O activities (opening, reading, and writing);
- *sms*, *call*: generated whenever a text message or a phone call is sent or received;
- *leak*: generated whenever the app leaks private information, as determined by `Taintdroid` [23]; and
- *dexload*: generated when an app loads native code.

Finally, our prototype allows performing analysis tasks in parallel. We presently limit our implementation to a small number of CoI models, FIO operators, and differential matching operators. Nonetheless, our architecture allows security experts to further extend this and configure their own operators based on their experience.

3.4.2 CoI Models

ALTERDROID currently supports the following models for identifying CoIs:

- `EXEFileMatch`. This model analyzes components of type Dalvik Executable Format (`DEXFileMatch`), Application Package file format (`APKFileMatch`), and Executable and Linkable Format (`ELFFileMatch`), i.e., $\tau(c) = \langle DEX, APK, ELF \rangle$. The model defined for these components is based on the magic number defined in the file header.
- `ImgFileMatch`. This model analyzes components of type picture, such as PNG, JPG, or GIF images, i.e., $\tau(c) = \langle PNG, \cdots, JPG \rangle$. This is based on the magic number defined in the file header, similarly to the model above.
- `EncryptedOrCompressedMatch`. This model matches any file whose entropy, measured at the byte level, exceeds a given threshold. In such a case, the file is considered to contain random or encrypted information and, therefore, is selected for fault analysis. We set the current threshold to 3.9. Such value was chosen after measuring the entropy of several files before and after being encrypted with DES.
- `ExtensionMismatch`. This model identifies files such that their magic numbers do not match the file extension. For instance, we found several *APK* files with *DB* extension and several encrypted files with *JPG* extension. We currently support two submodels: `ImgFileExtensionMismatch` and `APKFileExtensionMismatch`.
- `TextScriptMatch`. This model analyzes components that match any ASCII text executable file, i.e., $\tau(c) = Script$. This model is also based on the magic number defined in the file header.

All CoIs described above are implemented in Python. The set can be easily extended to incorporate additional models by simply adding the corresponding module.

FIO	Type	Targeted CoIs	ind.

`GenericFMutation`	Any file	`ImgExtensionMismatch`
		`EncryptedOrCompressed`
		`APKFExtensionMismatch`
			✓
			–
			✓

`ImgFileChange`	Any image	`ImgFileMatch`	✓
`ScriptFileChange`	Non-compiled program	`TextScriptMatch`	×
`APKFileChange`	Android app	`APKFileMatch`	×
`DEXFileChange`	Dalvik executable	`DEXFileMatch`	×
`ELFFileChange`	Executable and linkable	`ELFFileMatch`	×

TABLE 1
FIOs implemented in ALTERDROID's current version and their corresponding CoIs (ind. = indistinguishable).

3.4.3 Fault Injection Operators

FIOs in ALTERDROID are strongly typed. This pevents syntactic errors during the execution of the modified app. For instance, if a generic FIO randomly modifies chosen bits of a JPEG without considering the file structure, it may end up with a malformed picture that could cause the app to crash during execution. We currently support the following FIOs (see also Table 1):

Name	Contains	Rule
R_{NBC}	Network Behavior Component	$\exists\, i_1 :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_1}^{net-open})$ $\vee \exists\, i_2 :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_1}^{net-read})$ $\vee \exists\, i_3 :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_2}^{net-write})$
R_{FBC}	File Behavior Component	$\exists\, i_1 :$ cont$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_1}^{file-open})$ $\vee \exists\, i_2 :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_1}^{file-read})$ $\vee \exists\, i_3 :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_2}^{file-write})$
R_{DLC}	Data Leak	$\exists\, i$: contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_i^{leak})$
R_{SBC}	SMS Behav.	$\exists\, i$: contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_i^{sms})$
R_{PBC}	Payload	$\exists\, i$: contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_i^{dexload})$
R_{UPC}	Update Payload	$\exists\, i_1 :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_1}^{net-read})$ $\wedge \exists\, i :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_i^{dexload})$
R_{CBC}	Crypto	$\exists\, i$: contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_i^{crypto})$
R_{CPC}	Crypto Payload	$\exists\, i_1 :$ contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_{i_1}^{crypto})$ $\wedge \exists\, i$: contains$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \text{Del}_i^{dexload})$
R_{HFC}	Hidden Func	$\neg$ equal$(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^c(P)]), \emptyset)$

TABLE 2
Basic indistinguishable differential rules implemented.

- `ImgFileChange`. This FIO changes a number of pixels of image file components. The FIO type matches components of type `ImgFileMatch`. This is an indistinguishable FIO due to the nature of the changes

and the type of component. Thus, although the image resulting from the injection will be different, this change should not alter the app execution flow.

- `EXEFileChange`. This FIO replaces the file with a well-formed *APK*, *DEX* or *ELF* file that effectively does nothing, equivalent to a NOP (no-operation) injection. This change should cause a different behavior in the resulting differential signature as the former *EXE* file has been replaced. Thus, this FIO is distinguishable.
- `ScriptFileChange`. This FIO replaces the file with a valid NOP script. It only matches components of type `ScriptFileChange`. This FIO is also distinguishable.
- `GenericFileMutation`. It randomly changes several bytes of a file. This FIO is applied when there is no information about the file type and its structure, e.g., when injecting faults to encrypted files (`EncryptedOrCompressedMatch`) or when the file extension does not match its magic number `ExtensionMismatch`. This FIO might be distinguishable or indistinguishable, depending on the file.

As in the case of CoI models, FIOs are implemented in Python and provided with ALTERDROID's current version. Again, the set can be easily extended with additional FIOs by adding the corresponding module.

3.4.4 Differential Rules

The basic set of differential rules incorporated in ALTERDROID comprises the 9 rules shown in Table 2. They all apply to indistinguishable FIOs and cover the most common examples of obfuscated functionality: network activity, file activity, data leakage, SMS activity, hidden payloads, update attacks, cryptographic activity, cryptographic payloads, and generic hidden functionality.

To reduce the complexity of the search space, all basic rules apply to indistinguishable FIOs. However, for the sake of completeness our implementation incorporates several distinguishable FIOs, and new rules can be further added to match them. For instance, given an app that incorporates a DEX program used to enhance photos taken from the camera, we can use a rule to check whether this CoI actually does just that or not.

Thus, if after applying a FIO over this component the differential signature shows, for instance, changes in network activity, we may suspect that the CoI contained other functionality piggybacked on the DEX.

Formally, given `DEXFileMach` $\in$ `CoIS` and its corresponding distinguishable FIO (i.e., `DEXFileChange`), the following rule captures this intuition:

$$\mathbf{R_{DEX}} : dex \in \mathcal{P} \text{ contains NET activity} \iff \neg\mathsf{ind}(\Psi^{dex}) \wedge \exists\, i : \mathsf{contains}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^{v}(\mathcal{P})]), \mathsf{Del}_i^{net}\big).$$

4 EVALUATION

We next report a number of experimental results obtained with our prototype implementation of ALTERDROID. These results illustrate how our system can be used by market operators and security analysts to facilitate the analysis of complex obfuscated mobile malware. We first present the results of testing ALTERDROID against two datasets of smartphone malware samples found in the wild, including a performance analysis of the entire differential fault analysis process. We finally discuss in more detail three representative case studies.

4.1 Analytical Results

We tested ALTERDROID against a dataset composed of around 10000 apps retrieved from the following repositories: Aptoide (AP) alternative market[5], VirusShare (VS)[6] and Google Play (GP) [7]. Every app was executed over a time span of 120 seconds—current malware is generally quite eager to run their payloads promptly [18], so this time suffices to activate most malicious payloads.

Table 3 provides a summary of the obtained experimental results, including the average time required for analyzing one app (this includes the time for extracting CoIS and injecting faults into each component). Further, repackaging time, testing time as well as differential signature creation and analysis are included as well.

When analyzing the distribution of CoIs throughout the apps in our datasets, we observed that some apps have a fairly large amount of CoIs. For instance, some apps contain over 5K pictures (`ImgFileMatch`). Conversely, we found many others with less than 10 CoIs. On average, our experiments show that there are about 146, 284, 410 CoIs per app in VS, AP, and GP respectively, as shown in Table 3. Note that the number of CoIs from AP is twice the number of CoIs from VS. Similarly, the number of CoIs in GP is significantly higher than in VS and AP. In any case, the amount of potentially malicious components is significant and the time required to manually analyze each of them is affordable.

Finally, our results report the number of apps matching against the rules implemented in our prototype. For instance, we could identify 220 apps reporting components containing SMS functionality ($\mathbf{R_{SCC}}$) from all 2.9K samples in VS. Conversely, we could not find any $\mathbf{R_{SBC}}$ rule in Aptoide nor in GP (see Table 3). One alarming result is that we found a significant number of apps (669) reporting components containing data leakage functionality ($\mathbf{R_{DLC}}$) in AP. However, our results show that GP contains a much lower number of apps reporting data leakage functionality.

One interesting aspect of ALTERDROID is that it can inject all selected FIOs at once. Furthermore, ALTERDROID allows performing several analyses concurrently. In fact, our current experimental setup allows the execution of 15 Android instances in parallel. Thus, this simple optimization strategy reduces the average execution time

5. http://aptoide.com/
6. http://virusshare.com/
7. http://play.google.com/apps/

		VS	AP	GP
Sum.	No. Apps	2 913	2 994	4 000
	Avg. No. CoIS	145.6	284.4	409.9
	Avg. No. FIOs	138.3	273.5	397.3
CoIS	ImageFileMatch	397 248	813 754	1 586 379
	EncOrCompressed	16 687	35 293	45 781
	ImgExtensionMismatch	5 771	5 246	3 130
	DEXFileMatch	2 827	2 995	4 007
	APKFileMatch	1 087	58	220
	APKExtensionMismatch	517	39	183
FIOs	ImageFile	397 248	813 754	1 586 379
	GenericMutationFile	5 714	5 237	2 698
Rules	No. R_{FBC}	2 802	2 962	3 961
	No. R_{NBC}	2 773	2 929	3950
	No. R_{HFC}	1 971	669	95
	No. R_{SBC}	220	0	0
–	Average Overhead	584.51 s	666.67 s	567.99 s

TABLE 3
Analysis of the VS (VirusShare), AP (Aptoide), and GP (Google Play) datasets. The number of CoIs and FIOs is summarized (Sum.) on average per app and also given on absolute value. The number of rules matching (NAC and DLC) is also given in absolute value, and the overhead is given in seconds on average per app.

	#Apps	TP	TN	TPR	TNR
DKF	34	33	n/a	97.06%	n/a
ASB	187	186	n/a	99.47%	n/a
GM	4	3	n/a	75%	n/a
GM+	4	4	n/a	100%	n/a
Gray	16	16	n/a	100%	n/a
Good	81	n/a	81	n/a	100%

TABLE 4
Accuracy evaluation against existing malware, grayware, and goodware apps. True Positives, True Negatives, and their ratios are defined as expected.

per app at 32.62, 44.44 and 37.87 seconds for VS, AP, and GP respectively.

One challenge we faced when analyzing apps from AP is identifying whether some behaviors were malicious or not. Many legitimate apps are not fully malicious but carry out activities that may constitute a privacy risk for some users. During our analysis, most such suspicious behaviors were related with accessing local data and exfiltrating it over the network. We did not analyze in detail whether this was an intrinsic behavior of the app caused by the fault-injection process, for example because the app contained an integrity check. Nonetheless, this indicates that the app was behaving suspiciously and therefore it is worth analyzing.

4.2 Accuracy

From all apps tested above, we selected 300 known obfuscated malicious samples, grayware [6], and goodware and evaluated the accuracy of ALTERDROID. More precisely, we tested more than 200 variants of DKF, ASB, and GM and about 100 legitimate apps from GP. Every app was executed over a time span of 120 seconds except for the GM+ ones, requiring 1200 seconds. Table 4 summarizes the experimental results obtained and shows the detection rates. A usual measure of accuracy (sensitivity) is the True Positive Rate (TPR); that is, the percentage of functionality-hiding apps (malware, grayware) correctly identified as such. Another relevant measure of accuracy (specificity) is the True Negative Rate (TNR), which accounts for the percentage of goodware apps correctly identified as not containing hidden functionality.

Our experiments show that ALTERDROID performs very well, especially when dealing with obfuscated malware (DKF, ASB and GM). In fact, a significant number of rules per app matched the aforementioned differential signatures containing suspicious behaviors, such as network (R_{NBC}) or data leakage (R_{DLC}) activity (see tables 2 and 3). Additionally, no false positive was produced in the above tests on goodware (i.e. TNR reached 100%). The overall accuracy (i.e. $(TP+TN)/(Total\#Apps)$) was around 99%. The only case where TPR drops below 97% (GM) was related to the short time given to dynamic analysis. This was corrected by just increasing it. In particular, when the increased time window was adopted (GM+), the TPR achieved 100%.

4.3 Performance

The time taken by the entire differential analysis process depends on the number of different fault-injected apps to be explored, the time required to generate each of them, and the time taken by the differential analysis over each one:

$$t = n_{\text{faultApps}} \cdot t_{\text{genFaultApp}} \cdot t_{\text{diffAnalysis}} \tag{15}$$

As for the first term, if $|\text{CoIS}| = n$ and there are m FIOs, the fault injection algorithm shown in Fig. 3 generates $O(m + \log n)$ different fault-injected apps to be analyzed. Each one of those apps has been injected with at most n faults, one per component. The time t_{FIO} required to inject one fault depends on the specific FIO, although most of them run in constant time or are linear in the size of the component to be fault-injected. Finally, differential analysis requires:

- Executing the two apps. In ALTERDROID this is done by a component which admits as input the time t_{exec} during which the app will be executed. In our experiments, we determined that around 2 minutes suffice for most malware samples in our dataset.
- Obtaining the differential signature, which reduces to computing an edit distance between the two behavioral signatures. If these signatures have lengths s_1 and s_2, then this process takes $O(s_1 \cdot s_2)$ steps.
- Pattern-matching the differential signature with the rule-set, which takes $O(|\mathcal{R}|)$.

Apart from t_{exec}, the two most critical parameters affecting the total analysis time are n and m, as defined above (i.e., the number of CoIs and FIOs, respectively).

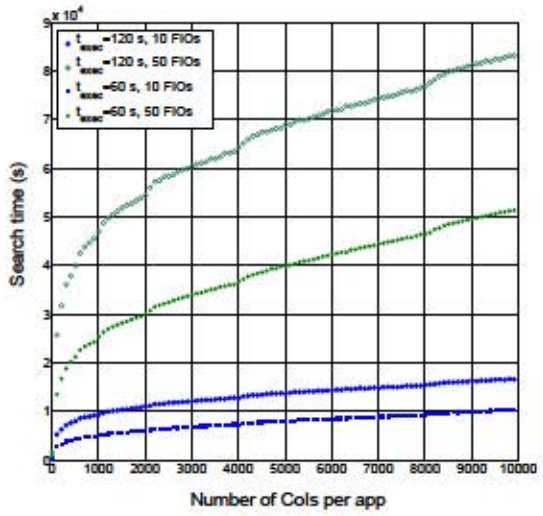

Fig. 5. Average execution time of the SearchComponent algorithm for different number of FIOs and dynamic analysis time.

Fig. 5 shows the average execution time of the SearchComponent identification algorithm at the core of ALTERDROID for different values of n, m, and t_{exec}. For example, the analysis of an app containing 100 CoIs for which 10 FIOs are applicable, and executing each fault-injected app 120 s, will require around 5 minutes. This time increases to 2.5 hours and 4.5 hours if the app contains 1K or 10K CoIS, respectively. If we decrease the dynamic execution time of each app to 60 s, these figures reduce to 2.7 minutes, 1.3 hours, and 2.9 hours, respectively.

4.4 Case Studies

We finally provide a more detailed discussion on the analysis of three relevant malicious apps found in Android markets. These three samples constitute representative cases as they incorporate obfuscation techniques of various degrees of sophistication [10], as well as some malicious features common in malware for smart devices [6] such as aggressive privilege escalation exploits, C&C-like functions [24] and information leakage.

For these three cases, we analyze their CoIS, we inject various faults into such components, and perform the resulting differential analysis. The findings discussed below about these three malware families are in accordance with previous reports, including those undertaken by Jiang and Zhou in [10].

Finally, we discuss several cases of recent apps that use different obfuscation techniques.

4.4.1 DroidKungFu (DKF)

DKF's main goal is to collect details about the infected Android device, including the IMEI (International Mobile Station Equipment Identity) number, phone model, and OS version. It is mostly distributed through open or alternative markets via repackaging—that is, by piggybacking the malicious payload into various legitimate applications. Apps infected with DKF are distributed together with a root exploit hidden within the app's assets, namely, Rage Against the Cage (RAC) [25]. To hinder static analysis, this encrypted payload is only decrypted at runtime.

We fed one DKF variant to ALTERDROID, which first extracted the variant's CoIS, injected various faults into these components, and then applied differential analysis by executing the resulting app and comparing it to the original. The sample contained about 170 resource files, including 153 PNG files, 6 MP3 files, 2 XML files, 1 DEX file, and an RSA key file. All these assets were, in principle, suspected of containing obfuscated functionality.

Figure 6 (top left) shows the differential behavior reported by ALTERDROID over a two-minute period. Activities launched by the original piggybacked app correspond to the full plot, while the behavior after fault injection is indicated by the green (legitimate app) and black (DKF) squares. ALTERDROID revealed that a text file pertaining to the assets was randomly modified. We later identified this file as the component containing the RAC exploit and found that disabling the malware's access to such functionality prevented it from establishing a network connection (net-open, net-write), leaking information through it (leak), and later performing some I/O operations (file read). This analysis agrees with previously reported results about DKF.

In the case of DKF, applying standalone static detection techniques was not sufficient by itself to identify malicious payloads without human-driven inspection. This is due to the way the malware obfuscates its core components. Specifically, each variant uses a different encryption key hidden throughout the code. Even when we attempted to apply standalone dynamic analysis, this technique only gave a rough notion of the app's holistic behavior. In fact, the behavior introduced by DKF is strongly entwined with the original code of the repackaged app such that some of its key activities, like network connections, might be easily seen as normal.

4.4.2 AnserverBot (ASB)

The ASB specimen we analyzed is similar to the first versions of DKF in terms of sophistication and distribution strategy. However, ASB introduces an update component that enables it to retrieve at runtime secondary payloads and the latest C&C URLs from public blogs. It also incorporates advanced anti-analysis methods to avoid detection: on the one hand, ASB introduces an integrity component to check if the app has been modified, while on the other, it piggybacks the main payload in native runnable code. Furthermore, ASB obfuscates its internal classes and methods, and partitions the main payload into two different parts: while one is installed, the other is dynamically loaded without actually being installed. Specifically, ASB hides one of these components into the assets folder under the names `anservera.db` or

Fig. 6. Activities of DKF, GM and ASB during a time span of 120 seconds.

`anserverb.db`. In addition, it inserts a new component named `com.sec.android.provider.drm` that executes a root exploit known as *Asroot*.

As in the case of DKF, we observed that all ASB variants contain a non-negligible amount of candidate CoIs. The specimen we examined had about 78 resource files, including 54 image files, 1 database file, 1 DEX file, and a ZIP file. After a few iterations of the fault injection process, ALTERDROID positively identified the actual payload within the database file, as well as the behavior related to this component. More precisely, it triggered this CoI after observing a mismatch between the magic number of the file (APK) and the actual extension of the database. In fact, when a fault is injected into the database file, ASB's integrity check naturally aborts its execution and produces a result similar to that expected from the original app.

Figure 6 (bottom) shows the exhibited differential behavior over two minutes. ASB first establishes a network connection (net-open, net-write) after loading the main payload (file-read, dex-load). After that, it continues reading data that it finally leaks out. Interestingly, the legitimate app uses the network as well, although it does not leak any personal information.

4.4.3 GingerMaster (GM)

GM is the first known malware to use root exploits for privilege escalation on Android 2.3. Its main goal is to exfiltrate private information such as the device ID (IMEI number, MSI number, and so on) or the contact list stored in the phone. GM is generally repackaged with a root exploit known as *GingerBreak* [26], [27], which is stored as a PNG and a JPEG asset file. Right after infecting the device, GM connects to the C&C server and fetches new payloads.

We analyzed a GM sample with around 60 asset resources, 30 of which were photos in different formats. Of those images, ALTERDROID identified four as strongly suspicious. A detailed analysis later revealed that they were malformed PNGs that also contained several ASCII scripts. ALTERDROID was also able to determine that such malformed image files play a key role in triggering the payloads piggybacked into the legitimate app, including the ASCII scripts.

Figure 6 (top right) shows the differential behavior exhibited over a two-minute period when Alterdroid injected such images with faults. GM started execution of a service that performs some I/O operations (file-read, file-write) before finally leaking private information through the network (net-write, leak). Again, even when the malicious components were hidden, Alterdroid was able to differentiate them and help identify the underlying malicious behavior.

4.4.4 Other Recent Specimens

We have analyzed some of the most recent specimens hitting both official and unofficial markets. Although obfuscation techniques and algorithms might vary, results confirm that malware keeps hiding payloads within app resources such as images or XML files. The most significant analyzed specimens were:

- **Emmental**: this malware sample targets users of several banks worldwide, collecting one-time passwords used to authorize transactions. Apps infected with Emmental are distributed together with an initial configuration containing a phone number where certain SMSs are sent and several Command and Control (C&C) URLs. To hinder static analysis, this configuration is only decrypted at runtime using

Blowfish. According to a report from Trend Micro [28], Emmental was still active as of 2014.

- **Gamex**: this specimen introduces an update component that enables it to retrieve new payloads, at runtime, from a C&C server. Its main goal is to exfiltrate private information such as the device ID (IMEI number, MSI number, and so on). Gamex [29] obfuscates the main payload using XOR operations while hiding it into the app resources—specifically, a file called *logos.png*.
- **SmsSpy**: this malware is similar to Emmental in terms of sophistication and distribution strategy [30]. It also uses Blowfish to encrypt its payload and hinder analysis. The payload is generally stored in a file called *data.xml* and the decryption key is hard-coded in the app code.

4.5 Discussion

Although current malware is relatively naïve, more sophisticated obfuscation techniques—particularly in code—are starting to materialize. Cryptography is one recurrent technique used by malware developers. Nonetheless, we believe that malware could be already using other advanced techniques for hiding their components such as, for instance, steganography [11]. This technique would allow them to conceal their malicious components within other objects of the code. This is specially critical when these components are hidden within distinguishable components.

In practice, any detection mechanism can be evaded, especially if its internals are well known. In the case of ALTERDROID, an attacker can run it on his own malware and then progressively adapt the sample so as to minimize the chances of the obfuscated payload being detected. For instance, the attacker can try to modify the payload in such a way that the component where it is included is not identified as a COI. In addition, it is well known that dormant or targeted functionality/malware is the Achille's heel for any approach involving dynamic analysis. In this regard, we believe that the approach introduced in ALTERDROID is relevant in the context of the never-ending battle between malware developers and detection mechanisms. However, addressing some limitations of current dynamic analysis techniques is left for future work.

5 Related Work

A substantial amount of recent work has addressed the problem of analyzing malware in smartphones using a variety of techniques [6]. Static analysis techniques are well known in traditional malware detection and have recently gained popularity as efficient mechanisms for market protection [15], [31], [32]. However, these techniques fail to identify malicious components when they are obfuscated or embedded separately from the code (e.g., hidden into an image) [33], [34], [35]. Approaches based on dynamic code analysis [36] are promising, but current works [37], [20], [38], [39] only provide an holistic understanding of the behavior of an app. This feature challenges the identification of grayware and the attribution of malicious behavior to components of the app. Thus, these approaches tend to miss their identification and further human (costly) efforts are required to dissect each malware sample, understand its rationale, and identify their payloads as shown by Zhou and Jiang in [10], [40]. For instance, XManDroid [41] extends Android's security architecture to prevent privilege escalation attacks at runtime based on security policies. Malware detection depends on such previously defined policies. Thus, inadequate policy rules can result in both overlooking grayware and affecting functionality of legitimate applications. Furthermore, the definition of these policies does not allow identifying hidden or obfuscated functionality as we do in this paper.

Recent work aims at detecting obfuscated malware by mining identifiable static features such as cryptographic functions [42]. However, Schrittwieser et al. [43] demonstrate the incompleteness of these and other semantic-aware detectors [44] by means of "covert computation." As for the various ways to obfuscate or locate obfuscated code in binary data, [45] describes the most relevant steganographic and steganalytic techniques, including active [46], [47] and passive wardens. These *wardens* are used in this paper to deploy semantic-aware FIOs to sanitize COIs, eliminating hidden information and detecting where it is hidden.

Fuzz Testing or Fuzzing is a technique commonly used for providing inputs when testing software for security purposes [48]. Fuzzing has been recently gaining popularity for automating the dynamic analysis of apps in smartphones [19], [20], [49], [38]. Basically, Fuzzing aims at providing different streams of events to the app for further monitoring the behavior of the device. Fuzzing was originally proposed to find software crashes or unexpected behaviors by deliberately introducing faulty inputs. Our approach is similar to Fuzzing, but focuses on the manipulation of a program's components rather than its inputs.

Fault injection analysis has been widely used for software assurance against fault tolerance [50], [51]. This paper extends an early version of this work [18], where differential fault injection analysis is introduced and discussed. Together with our previous work, differential fault analysis is a novel approach compared to existing works aiming at analyzing malware in smartphones.

Finally, our differential fault analysis approach can be integrated on top of any system aiming at reconstructing apps' behavior, such as CopperDroid [39] or Targetdroid [52]. As these approaches were not available during the initial phase of our development, we instead implemented several state-of-the-art techniques to automatically extract the behavior of monitored apps. Nonetheless, other monitoring systems can be further plugged into ALTERDROID to extend the number of monitored features or to better detect reactive malware

[52].

6 Conclusions

In this paper we have presented ALTERDROID, a framework for malware analysis based on the notion of differential fault analysis. We have described its architecture and provided a formal model of differential fault analysis. Additionally, we have presented an open-source prototype implementation of ALTERDROID with a versatile design that can be the basis for further research in this area.

Differential fault analysis in the way implemented by ALTERDROID is a powerful and novel dynamic analysis technique that can identify potentially malicious components hidden within an app package. Additionally, empowering dynamic analysis with a fault injection approach can be used to differentiate "gray" from legitimate behavior when analyzing *grayware*. This is a good complement to static analysis tools, more focused on inspecting code components but possibly missing pieces of code hidden in data objects or just obfuscated. Finally, we believe that differential fault analysis is an effective technique to detect *stegomalware*—malware using advanced hiding methods such as steganography. As future work, we are currently extending ALTERDROID to support differential fault analysis over distinguishable components such as those involving Dex bytecode.

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful comments.

This work was partially supported by the MINECO grant TIN2013-46469-R (SPINY: Security and Privacy in the Internet of You) and the CAM Grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Big Data, and Risks).

References

[1] Y. Wang, K. Streff, and S. Raman, "Smartphone security challenges," *IEEE Computer*, vol. 45, no. 12, pp. 52–58, 2012.

[2] L. Cai and H. Chen, "Touchlogger: inferring keystrokes on touch screen from smartphone motion," in *Proc. USENIX*, ser. HotSec'11, Berkeley, CA, USA, 2011, pp. 9–9.

[3] E. Fernandes, B. Crispo, and M. Conti, "Fm 99.9, radio virus: Exploiting fm radio broadcasts for malware deployment," *IEEE TIFS*, 2013.

[4] T. Vidas and N. Christin, "Sweetening android lemon markets: Measuring and combating malware in application marketplaces," in *Proc. ACM*, ser. CODASPY '13. ACM, 2013, pp. 197–208.

[5] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, 2012.

[6] G. Suarez-Tangil, J. E. Tapiador, P. Peris, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Comms. Surveys & Tut.*, vol. 16, no. 2, pp. 961–987, May 2014.

[7] M. Rangwala, P. Zhang, X. Zou, and F. Li, "A taxonomy of privilege escalation attacks in android applications," *Int. J. Secur. Netw.*, vol. 9, no. 1, pp. 40–55, Feb. 2014.

[8] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *Proc. ACM*, ser. WiSec '13. New York, NY, USA: ACM, 2013, pp. 13–24.

[9] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day Android malware detection," in *Proc.*, ser. MobiSys '12. ACM, 2012, pp. 281–294.

[10] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109.

[11] G. Suarez-Tangil, J. E. Tapiador, and P. Peris-Lopez, "Stegomalware: Playing hide and seek with malicious components in smartphone apps," in *INSCRYPT 2014*, December 2014.

[12] A. Desnos and et al., "Androguard: Reverse engineering, malware and goodware analysis of android applications," https://code.google.com/p/androguard/, Visited Feb.2015.

[13] Panxiaobo, "Apktool: A tool for reverse eng. android files," https://code.google.com/p/android-apktool/, Visited Feb. 2015.

[14] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29.

[15] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 1, pp. 1104–1117, 2014.

[16] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *S. Physics Doklady*, vol. 10, p. 707, 1966.

[17] T. Kumazawa and T. Tamai, "Counter example-based error localization of behavior models," in *Proc.*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 222–236.

[18] G. Suarez-Tangil, F. Lombardi, J. E. Tapiador, and R. Di Pietro, "Thwarting obfuscated malware via differential fault analysis," *IEEE Computer*, vol. 47, no. 6, pp. 24–31, June 2014.

[19] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing UI-based trigger conditions in Android applications," in *Proc. ACM*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 93–104.

[20] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proc. ACM*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 209–220.

[21] Android, "Android developers," Visited Feb. 2015, http://developer.android.com/.

[22] Google, "Droidbox: Android application sandbox," https://code.google.com/p/droidbox, 2012.

[23] W. Enck, P. Gilbert, B.-G. Chun, and al., "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. USENIX*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[24] R. Hasan, N. Saxena, T. Haleviz, S. Zawoad, and D. Rinehart, "Sensing-enabled channels for hard-to-detect command and control of mobile devices," in *Proc. ACM SIGSAC*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 469–480.

[25] C-skill, "Rage against the cage," https://github.com/bibanon/android-development-codex/wiki/rageagainstthecage, 2011.

[26] C. Skill, "Gingerbreak," http://c-skills.blogspot.hk/2011/04/yummy-yummy-gingerbreak.html, 2011.

[27] M. Zheng, M. Sun, and J. C. Lui, "Droidray: A security evaluation system for customized android firmwares," in *Proc. ACM*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 471–482.

[28] D. Sancho, F. Hacquebord, and R. Link, "Finding holes: Operation emmental," Trend Micro, Tech. Rep., 2014, http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-finding-holes-operation-emmental.pdf.

[29] Symantec, "Android.gamex," http://www.symantec.com/securityresponse/writeup.jsp?docid=2012-051015-1808-99.

[30] F-secure, "Smsspy," https://www.f-secure.com/weblog/archives/00002202.html.

[31] M. Lindorfer, S. Volanis, A. Sisto, and al., "Andradar: Fast discovery of android applications in alternative markets," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. LNCS, S. Dietrich, Ed., 2014, vol. 8550, pp. 51–71.

[32] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. NDSS*, February 2014.

[33] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM CCS*. ACM, 2003, pp. 290–299.

[34] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proc. ACM SIGSAC*, ser. ASIACCS, 2013, pp. 329–334.

[35] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Trust and Trustworthy Computing*. Springer, 2013, pp. 169–186.

[36] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, "Mobile application testing: A tutorial," *Computer*, vol. 47, no. 2, pp. 46–55, Feb 2014.
[37] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2012.
[38] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security*, 2014.
[39] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *NDSS Symp.* Internet Society, February 2015.
[40] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: bare-metal analysis-based evasive malware detection," in *Proc. USENIX SEC'14.*, 2014, pp. 287–301.
[41] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," Tech. Universitat Darmstadt, Tech. Rep., 2011.
[42] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *Proc. ACM*, ser. CCS '12. ACM, 2012, pp. 169–182.
[43] S. Schrittwieser, S. Katzenbeisser, P. Kieseberg, M. Huber, M. Leithner, M. Mulazzani, and E. Weippl, "Covert computation: hiding code in code for obfuscation purposes," in *Proc. 8th ACM SIGSAC*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 529–534.
[44] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*, May 2005, pp. 32–46.
[45] J. Blasco Alís, "Information leakage and steganography: detecting and blocking covert channels," Ph.D. dissertation, Universidad Carlos III de Madrid, 2012.
[46] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil, "Eliminating steganography in internet traffic with active wardens," in *5th Intl. Worksh. on Information Hiding*, ser. IH '02. London, UK, UK: Springer-Verlag, 2003, pp. 18–35.
[47] E. Li and S. Craver, "A square-root law for active wardens," in *Proceedings of the thirteenth ACM multimedia workshop on Multimedia and security.* New York, NY, USA: ACM, 2011, pp. 87–92.
[48] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance.* Artech House, 2008.
[49] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero, "Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications," *arXiv preprint arXiv:1402.4826*, 2014.
[50] J. Gray, "Why do computers stop and what can be done about it?" in *Symposium on reliability in distributed software and database systems.* Los Angeles, CA, USA, 1986, pp. 3–12.
[51] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "On fault representativeness of software fault injection," *Software Engineering, IEEE Transactions on*, vol. 39, no. 1, pp. 80–96, Jan 2013.
[52] G. Suarez-Tangil, M. Conti, J. E. Tapiador, and P. Peris-Lopez, "Detecting targeted smartphone malware with behavior-triggering stochastic models," in *ESORICS 2014*, ser. LNCS, vol. 8712. Springer International Publishing, 2014, pp. 183–201.

Guillermo Suarez-Tangil is Teaching Assistant in the Computer Security (COSEC) Lab at Universidad Carlos III de Madrid. His research focuses on security in smart devices, intrusion detection, event correlation, and cyber security.

Juan E. Tapiador is Associate Professor of Computer Science in the Computer Security (COSEC) Lab at Universidad Carlos III de Madrid, Spain. His main research interests are in computer/network security and applied cryptography.

Flavio Lombardi is a Researcher at IAC-CNR and Adjunct Professor of Computer Science at Dept. of Mathematics and Physics, Roma Tre University. His main research interests focus on: cloud security; GPGPU computing; virtualization; S&P for mobile and distributed systems.

Roberto Di Pietro is with Bell Labs, Cyber Security Research. He is also with Maths Dept. at University of Padua, Italy. His main research interests include: security and privacy for wireless systems; cloud and virtualization security; security and privacy for distributed systems; applied cryptography; computer forensics, and role mining for access control systems (RBAC).