GreatEatlon: Fast, Static Detection of Mobile Ransomware

Chengyu Zheng(✉), Nicola Dellarocca, Niccolò Andronio, Stefano Zanero, and Federico Maggi

DEIB, Politecnico di Milano, Milan, Italy
{chengyu.zheng,nicola.dellarocca,niccolo.andronio,
stefano.zanero,federico.maggi}@polimi.it

Abstract. Ransomware is a class of malware that aim at preventing victims from accessing valuable data, typically via data encryption or device locking, and ask for a payment to release the target. In the past year, instances of ransomware attacks have been spotted on mobile devices too. However, despite their relatively low infection rate, we noticed that the techniques used by mobile ransomware are quite sophisticated, and different from those used by ransomware against traditional computers.

Through an in-depth analysis of about 100 samples of currently active ransomware apps, we concluded that most of them pass undetected by state-of-the-art tools, which are unable to recognize the abuse of benign features for malicious purposes. The main reason is that such tools rely on an inadequate and incomplete set of features. The most notable examples are the abuse of reflection and device-administration APIs, appearing in modern ransomware to evade analysis and detection, and to elevate their privileges (e.g., to lock or wipe the device). Moreover, current solutions introduce several false positives in the naïve way they detect cryptographic-APIs abuse, flagging goodware apps as ransomware merely because they rely on cryptographic libraries. Last but not least, the performance overhead of current approaches is unacceptable for appstore-scale workloads.

In this work, we tackle the aforementioned limitations and propose GreatEatlon, a next-generation mobile ransomware detector. We foresee GreatEatlon deployed on the appstore side, as a preventive countermeasure. At its core, GreatEatlon uses static program-analysis techniques to "resolve" reflection-based, anti-analysis attempts, to recognize abuses of the device administration API, and extract accurate data-flow information required to detect truly malicious uses of cryptographic APIs. Given the significant resources utilized by GreatEatlon, we prepend to its core a fast pre-filter that quickly discards obvious goodware, in order to avoid wasting computer cycles.

We tested GreatEatlon on thousands of samples of goodware, generic malware and ransomware applications, and showed that it surpasses current approaches both in speed and detection capabilities, while keeping the false negative rate below 1.3%.

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 617–636, 2017.
DOI: 10.1007/978-3-319-59608-2_34

1 Introduction

Nowadays there are approximately 1.9 billion smartphone and tablet users worldwide [1], using 3.7 billion devices, a number that is expected to grow to 6.75 billion by 2021 [2]. This widespread diffusion of mobile devices makes the attack surface substantial and the tendency to store sensitive data on mobile devices makes them an attractive target for malware authors.

According to GData [3], in the first half of 2015 more than 1 million infections on Android devices occurred, which means 6,100 newly infected devices every day, a 25% increase since 2014. Out of these infections, a few more than a half are financially motivated. More specifically, in 2015 the most dangerous threats were ransomware, whose number of families doubled in only one year and infected nearly 100,000 distinct users, a five-fold increase since 2014 [4].

Even though there are tools [5] that aim at post-infection recovery, they are effective only against some (known) ransomware families. Moreover, the state-of-the-art approach [6] is imprecise since it is only partially able to recognize certain feature of modern mobile ransomware. HelDroid works by analyzing three main characteristics that belongs to a ransomware, composed by a text, encryption, and locking analyzer. In this paper, we propose how to enhance HelDroid to overcome the limitations that we noticed after about one year of operation on modern ransomware families. More in detail, we modified the static taint analysis tool, on which the encryption detector is based. For instance, preventing decryption flows from being erroneously considered as malicious, lowering the number of false positives. Furthermore, we identified a different set of sources and sinks that allows the detector to identify encryption flows independently of the particular folder that contains the target files and augmented HelDroid for detecting the abuse of admin APIs, which are used by modern ransomware to urge victims to effectively lock the device. In addition to that, we propose a heuristic to statically resolve the method invoked via the most common reflection patterns, even in the presence of lightweight method name obfuscation. Finally, we implemented a pre-filter that aims to reduce the overhead of HelDroid by recognizing goodware.

We tested the resulting system, named GreatEatlon, on thousands of samples including ransomware, generic malware and goodware, using HelDroid as a benchmark.

In summary, the main contributions of this work are:

- a novel encryption-detection approach that can recognize, with good precision, malicious encryption flows, thanks to a generic set of sources and sinks, and by taking into account the nature of the encryption operation (*i.e.,* encryption vs. decryption initiated by the user via UI);
- a static technique, to discover device administration APIs abuse, which is widely used by modern ransomware families;
- a heuristic to detect the most common patterns used by malware to call methods via reflection;
- a lightweight and fast pre-filter able to discard goodware from the analysis;

2 Motivation

In this section, we introduce the problem in more detail, together with some solutions proposed by other researchers along with their limitations, and finally set the goals for this work.

2.1 Ransomware

A ransomware is a particular kind of malware which business model is to extort money from the victims. In order to force the user to pay the ransom, ransomware usually performs actions that limit the ability for the victim to use her device such as screen locking, or encrypting personal files. Mobile ransomware represents a concrete threat that is increasing by about 14.8% per year [7]. Mobile ransomware started with SimpleLocker [8], which is the first family of mobile ransomware that encrypts user's data with a unique key embedded in the binary and subsequently ask for money. In 2015 a more advanced version of SimpleLocker [9] appeared, instead of using a unique key able to decrypt and encrypt, it uses a per-device key.

It is important to note that by default Android's security model does not allow applications to do all kinds of operations. In particular, there are many APIs that are considered as potentially dangerous. In order to let an application to use these APIs, Android requires that, at install time, the user explicitly grants an application all the permissions it needs. Moreover, if an application needs to use the so-called Device Administration APIs (Sect. 3.2), then an additional run-time permission grant dialog is shown to the user, listing the administration policies the application requires, together with a brief message about the associated risk. Since the decision of whether to grant or deny these permissions is made by the user, it is very important that she understands the danger associated with the permission. Unfortunately, in [10] researchers have demonstrated that 84% of the users either do not pay attention to the permissions they grant or do not even know about the existence of them, and only 20% of all participants demonstrated "awareness of permissions and reasonable rates of understanding," choosing at least 70% of right answers to the survey they took. In fact, malware authors exploit this lack of attention to massively obtain permissions and use them to perform malicious actions and to spread quickly.

2.2 State of the Art

Being mobile ransomware a recent problem there are currently two kinds of tools available: commercial removal/cleanup utilities (*e.g.,* Avast Ransomware Removal [5]) and a research prototype that we prosecuted in [6], which offer a more generic approach, mainly based on static analysis.

Ransomware removal/cleanup utilities are specific to each ransomware family, thus it requires a certain effort keep them up to date with the development of new families. Additionally, these utilities are mobile applications that, like any Android app, are restricted by the security model of Android, hence they have

limited functionalities. Therefore, their detection approach is not possible to do anything more than signature checking [11]. Moreover, certain ransomware families exploit high privileges (*e.g.,* device admin API) to kill those processes that are typically associated to common AVs.

The second approach *HelDroid*, proposes a *feature-based* detection mechanism using advanced static-analyses techniques directly on the bytecode extracted from APK files. We envisioned HelDroid deployed on the app-store side to scan submitted application's code and resources in order to discover whether they exhibit one or more characteristics that belong to a ransomware-distinguishing feature set.

The quality of the outcome strongly depends both on the set of extracted features and on the ability of HelDroid to extract them correctly. *HelDroid* recognizes three main characterizing actions than can be used to distinguish ransomware from other kinds of malware or goodware. Namely, it detects whether the app is (1) displaying threatening message, (2) locking the device, and (3) encrypt personal files. Clearly, since HelDroid is based on static analysis, what it actually detects is the presence of code dedicated to implement these features. Such code may or may not be executed at runtime, depending on factors that fall outside the scope of analysis of HelDroid.

After about one year of experience with running[1] HelDroid on thousands of mobile ransomware samples, taken from the VirusTotal daily feed, we concluded that this set of features is good to characterize ransomware, it does not take into account some new features introduced by the most recent families, such as the ability to use highly privileged APIs. For this reason, we augmented the original HelDroid it with new detectors, which will be explained in detail in the following sections.

Moreover, we found that some of the features detectors were not precise enough to detect all possible ways for a ransomware to perform a malicious action. In particular, we refer to the encryption detector, which can easily be circumvented by using the encryption API in a slight different way than the expected one, and to the text detector, which does not take into account the possibility to convey the threatening text through pictures instead of plain text.

2.3 Goals and Challenges

The goal of this work is to improve the ability of some detectors to correctly identify those features that make ransomware distinguishable from other malware and goodware, providing at the same time new ones capable of finding new characteristics. In this way, the updated system will be able to increase the reliability of the outcomes.

The main challenge is to create a solution that is generic enough to be effective even with new samples.

[1] http://ransom.mobi/scans.

3 Approach

In this section, we introduce, at a high level, the novel detectors, postponing the implementation details to Sect. 4. Recall that the process unit of GreatEatlon is the APK file. Therefore, the detectors described in the rest of this section are run on each APK file—in addition to the detectors already present in HelDroid. Each APK file contains three kinds of files: a manifest file (`AndroidManifest.xml`), source code files, and resource files. We used Apktool [12] to decode APK files, but the same information can be extracted in other ways.

As in HelDroid [6], the final output of GreatEatlon is a combination of the detectors. However, the focus of this paper is not how the detectors are combined, but rather on how we improved the original ones.

3.1 Encryption Detector

Ransomware encrypts personal files of a victim, which can be stored in any paths, typically under the SD card tree. These paths can be retrieved (by the malware) in several ways. Therefore, a path- or folder-dependent detection can easily be circumvented. Our solution, instead, is based on the impossibility of the attacker to know the location of the target files in advance. As a result, ransomware will perform at least one folder/file-listing operation, mediated by the OS, to know which files are within inside a specific folder. Therefore, we take advantage of a static taint analysis to track all code flows originating in a "query" (a request sent to the OS to get the list of files contained inside a folder) and ending in one of the encryption-related APIs that Android offers to developers. The OS offers only a few ways to perform such query: a couple of methods from the `File` class and a low-level query that relies on the underlying shell. For this reason, if we are able to detect that there is an information flow starting from one of these methods and ending in a bulk file encryption, then we can reasonably assume that relevant encryption-related operations are made to user's files.

We minimize the chances of false positives by focusing only in those flows that actually perform encryption (and not decryption) because the latter is harmless, and can be performed by a wide variety of benign applications. To this end, we implemented the notion of **conditional flows**. A conditional flow is considered by the taint-analysis engine only if there exists at least one path between the source(s) and the sink(s) that satisfies all the given conditions. In our proof-of-concept prototype, we support conditions on function arguments because this is the minimum requirement for implementing our detection logic, but the concept can be extended further.

The ability for an encryption-related flow to either encrypt or decrypt depends on the value with which the `Cipher` (i.e., the Java class responsible for performing encryption and decryption) is initialized. Therefore, we are interested in defining conditions that based on this value. In particular, the `init()` method (and all its overloads) currently supports only two values: `ENCRYPT_MODE` (i.e., `0x1`) and `DECRYPT_MODE` (i.e., `0x2`). Given that these values are numeric constants, and that both the compiler for the Dalvik virtual machine and

the taint-analysis tool are able to perform constant propagation, we can adopt static conditions to make sure that a given tainted flow is an encryption or decryption one. To satisfy such a condition, an instruction like `cipher.init(ENCRYPT_MODE, ...)` must be called before a sink is reached.

3.2 Device Administration APIs Misuse Detector

Android's security model requires each application using Device Administration APIs to declare a specific set of permissions and components in the manifest file. Ransomware is obviously not exempted from doing so, allowing us discover the misuse of these APIs. More specifically, an app that needs to use device-admin policies must:

- Declare a class extending `DeviceAdminReceiver`, which is a component in charge of receiving and processing specific broadcast messages sent by the system whenever particular events happen (*e.g.,* when the user grants or revokes the privileges to the app).
- Declare a so-called *policy meta-data* XML file containing the list of all security policies that the app wants to use.
- Associate the XML file with the `Receiver` through a `<meta-data>` XML element.

Given these strict requirements, we created a detector that parses the `AndroidManifest` file, looking for the declaration of the appropriate `Receiver`. If found, and if the related meta-data contains dangerous policies (e.g., the ability to change the device unlock password and/or to remotely wipe the device), then it proceeds to analyze the source code.

Whenever the manifest and policy meta-data file analyses are completed and return a positive result, the detector processes the application source code. In this phase, we are interested in discovering if there exists at least a call to one of the potentially harmful methods of `DevicePolicyManager`, the main class that implements the Device Administrator APIs. In particular, the methods of interest are `wipeData()` and `resetPassword()`. To check whether the application calls one of these methods, we inspect the CFG to perform reachability analysis.

Interestingly, in an effort to hinder abuse of such APIs for permanent screen locking, the upcoming major release of Android (7.0, code-named "Nougat") eliminates [13] the possibility of creating device-admin policies to (programmatically) change the pass-code (e.g., PIN, pattern) without user intervention. This will certainly help in the future, but a short-time countermeasure—such as the detector presented in this section—is still required until mass adoption of Android Nougat.

3.3 Reflection Heuristic

Given the static nature of our analyses, all ransomware samples that make use of reflection or other dynamic techniques would not be detected by HelDroid,

causing the detector to produce a wrong outcome. Therefore, we implemented a heuristic that resolves the most common reflective calls and includes them in the CFG. For efficiency, this heuristic is invoked only if the statically built CFG does not trigger any of the GreatEatlon detectors. In particular, GreatEatlon perform a series of forward and backward analyses that recognize the usage of reflection (*i.e.,* method calls like `method.invoke()`) and reconstruct which are the target class and method in case they are obfuscated.

Although far from being general and exhaustive, we our heuristic traces back to the origin of the string that holds the method name, including string-obfuscating transformations—if any (e.g., the original string is "`lockNow`" but the malware author obfuscated it by inserting random chars in between each couple of chars). Since we noticed that the de-obfuscating process typically involves only methods from the `String` class (such as `replace` or `substring`), our detector re-applies any method found along the backward path on the target string. This method is clearly not generic, and can certainly be improved. However, string de-obfuscation is a wide research topic, falling outside the scope of this paper, which *leverages* such program-analysis techniques rather than proposing new ones.

3.4 Text Detector

In its original form implemented in HelDroid [6], this component is responsible for analyzing any ASCII string extracted from the sample (both statically and dynamically), guessing the language, and determining whether the phrases form a threatening message—typical of any ransomware scheme.

In GreatEatlon, we pre-pended a lightweight image-processing phase to this component in order extract text from images, so to make the overall system resilient to evasion (e.g., text rendered into images). In particular, we added an image scanner that inspects all image files shipped in the application resource directories and applies a set of transformations to optimize them for optical character recognition (OCR).

The extracted text is then automatically corrected by a standard spell checker, to remove the obvious errors that may occur during OCR. The resulting, corrected text is then processed with the original text analyzer of HelDroid, which queries a classifier that returns a score indicating the amount of "threatening" sentiment found. If higher than an empirically determined threshold, then the text is considered as threatening, indicating that the sample is likely to be ransomware.

3.5 Lightweight Pre-filtering

To quickly decide whether an application is suspicious, and thus worth spending computing resources to analyze it, we adopt a supervised-learning classification approach. When tackling a classification task it is crucial to design features that best discriminate between goodware and all the rest. A great amount of research work has been done in the area, proposing several static and dynamic

features that characterize malware vs. goodware [14–20]. However, if the goal is malware detection, errors are very costly in either sense (*i.e.,* false positives and negatives).

Instead, we make use of (some of the) features identified by previous work and relax some of these constraints by working on the dual problem (*i.e.,* detecting goodware). Since the pre-filtering is followed by the ransomware-detection pipeline, the cost of a few benign samples mis-classified as suspicious is negligible, because they will be eventually recognized as non ransomware. In other words, we can allow a slight penalty in the pre-filter accuracy in favor of almost perfect precision. We detail the choice of the classification algorithms and the features that we selected in Sect. 4.4. Since we need this phase to be fast, features that can be only extracted at runtime are unsuitable because the extraction would be prohibitively time consuming. Therefore, we focus on features that can be extracted efficiently by parsing the APK files. The output of this phase is a binary decision: "goodware" or "suspicious".

4 Implementation Details

In this section, we explain the technical details of GreatEatlon.

4.1 Encryption Detector

To implement our encryption detection approach, we extended FlowDroid [21] (the state-of-art static taint-analysis tool for Android) to allow the taint-propagation engine to track information flows through files (*e.g.,* a function writes to a file, another function reads such file, and passes the reference to function). Note that FlowDroid can be configured to ignore flows that originate from the user interface, effectively eliminating many false positives due to benign, user-initiate encryption. In particular, we modified FlowDroid to track information flows between (1) `InputStream` (and related classes) linked to the victim files and (2) `Cipher` objects in charge of encrypting them. Ransomware usually reads the original file through a loop, placing the bytes read in one of the parameters passed to the `read()` method, which is usually an array. Given that this parameter is not tainted directly by the ransomware, but it is manipulated internally by the `InputStream`, FlowDroid would not be able to detect this information flow. Luckily, the component that is in charge of deciding whether a particular instruction is involved in taint propagation (namely, one of the "taint wrapper" classes) can be easily extended to override the default taint propagation rules. Hence, we created a custom `TaintWrapper` that taints the parameter that will receive the file's bytes if the underlying `InputStream` is tainted in turn. In this way GreatEatlon is able to taint also the `Cipher` objects that receive the same tainted parameter, and that will eventually perform the encryption.

Conditional Flows. We designed conditional flows to be as generic as possible, in order to allow adding, removing and modifying conditions in a simple way

in the future. In particular, we created a module that "injects" conditions in FlowDroid by reading them from a text file formatted as follows:

```
NUMBER -> <CLASS: RET_TYPE METHOD(PARAMS)>
```

where:

- `NUMBER` is the index for the condition.
- `CLASS` is the fully-qualified class name for the class that declares the method on which we want to check the condition.
- `RET_TYPE` is the method return type.
- `METHOD` is the method name.
- `PARAMS` is the (possibly empty) list of comma-separated method's parameters.

The `NUMBER` token can take any non-negative integer value (typically a sequential number starting from 0 or 1) and is not part of the method signature, but instead it is used by GreatEatlon to decide whether the condition should be considered as an alternative or standalone. In fact, it is possible to specify an *alternative condition* (*i.e.,* a condition composed by two or more sub-conditions that is valid if at least one of the sub-conditions is valid) by using the same value for two or more conditions. In other words, the parser evaluates all conditions with the same index as logical disjunctions and conditions with different indexes as logical conjunctions. Alternative conditions can be useful to specify requirements on a method that has overloads or on multiple different methods.

The `PARAMS` token represents the list of actual parameters used in a method call. Currently, the only allowed values are Java primitive types, that are numbers (both integers and floating-point decimals), Boolean values, and characters, plus `null`, and a custom type indicated by "_". This custom type is essentially a "don't care", meaning that the i-th parameter can take any value (including reference types).

Condition Verification. In general, we could check for condition satisfaction either while performing the taint propagation, or after the taint analysis is completed. These two approaches have different impacts on performance and resource usage. The latter needs to store all the potential source-to-sink paths resulting from the taint analysis to perform the subsequent check. This number can explode if the sample is complex. Moreover, given that FlowDroid does not return the full path, but just a summary (*i.e.,* a path that typically contains only the source, the sink and the intermediate nodes involved in branch decisions), it would be necessary to manually reconstruct the full source-to-sink path to check whether there are some nodes that satisfy the conditions. This implies performing an additional control-flow graph analysis. In the worst case, all the potential paths are reconstructed and visited twice: the first time to perform the taint propagation, and the second time to check for conditions verification. This solution, however, does not require any modification to the taint analysis tool, so it might be helpful to implement the analysis this way if taint analysis is performed by a proprietary or immutable tool.

Instead, when conditions are checked while performing the taint analysis, the control-flow graph is examined only once, by the taint-analysis tool, which would also be in charge of verifying the conditions. This approach is undoubtedly faster than the previous one, but it requires to modify the taint analysis tool source code. Thanks to its open-source nature, we were able to extend InfoFlow, the FlowDroid sub-component that computes the taint analysis, to perform it this way, by modifying the objects that are responsible for taint tracking in order to make them deal with conditions sets. In particular, for each node visited during the taint propagation, we check whether it contributes in satisfying the conditions set. This information is then stored inside the object responsible for containing all data related to the taint, which is propagated to all children of a node when the CFG is explored. We finally modified the `TaintPropagationResults` class, which is responsible for adding paths to the set of results, to allow conditions verification: In this way, whenever a sink node is reached, this component adds the source-to-sink path to the set of results only if the associated condition set has been verified by a previous node.

4.2 Device Administration APIs Misuse Detector

In order to detect misuses of the device-admin APIs, we created a component that starts by analyzing the `AndroidManifest`. Subsequently, we take advantage of FlowDroid for generating CFG and entry-points, because it is designed to deal with this kind of applications and it can be configured to consider or ignore specific callbacks that can be invoked during the application life-cycle. For instance, two interesting entry-points are the `onEnable()` and `onDisable()` methods from class `DeviceAdminReceiver`, which are called by the OS whenever the user grants or revokes device administration rights to the application.

After these setup operations, the tool is ready to analyze the CFG. Since we are interested only in knowing if some methods are called by the sample but not in knowing the exact path, we can perform a simple reachability analysis, which allows us to quickly discover such a method call, if it exists. In particular, we decided to explore the CFG in a breadth-first fashion (BFS), because since the step costs are uniform (*i.e.,* we can assume that visiting a child node has a unitary cost) it can provide the optimal solution, reaching the target node (if it exists) by traversing as few edges as possible. Moreover, we avoid visiting the same node twice. This serves both as an optimization (when a target node is present in the CFG) and to avoid entering an infinite loop it the CFG is not acyclic. A never-terminating analysis, for instance, could occur if a sample contains a suspicious `AndroidManifest`, but it does not actually use any of the potentially dangerous methods, which often happens with (usually benign) applications that require more permissions than needed, or when the target nodes cannot be recognized by the detector, which happens when the sample uses reflection or other kinds of obfuscation.

4.3 Reflection Heuristic

Thanks to manual analyses, we observed that many ransomware applications exploit reflection to call device administration-related methods, which we cannot detect through the above-described procedure. For this reason, we decided to implement a heuristic to detect some common cases, in order to reduce the number of false negatives. In this step, we reuse the CFG generated in Section 4.2 to perform a series of forward and backward analyses in order to discover whether reflection is used and, if this is the case, to try to figure out which method is executed. In particular, we perform a first forward analysis from the application entry-points trying to reach a reflection call, that is an instruction like `method.invoke(...)`, where `method` is an object from class `Method` and represents a Java-callable method. If we find at least one instruction of this kind, it means that the application dynamically calls a method. Unfortunately, this information is not enough to prove that the sample is performing something malicious because we do not know the invoked method yet. To obtain this additional information, we perform a backward analysis whose target is to reach the `method` variable assignment, which usually involves hard-coded strings (because the attacker already knows which is the method to call).

Unluckily we discovered that in a few cases this procedure is not enough because in several samples the hard-coded method name was obfuscated, in order to circumvent those AVs that perform strings analysis. In particular, we observed that attackers manipulate these strings by adding some extra characters to the method name, for instance by transforming the string "resetPassword" to "resLetXPassVUwgXord". Given that the string de-obfuscation is performed by applying some transformations on the string, such as `String.replace`, we, in turn, take advantage of reflection to apply the same modifications to the original string to try to clean it.

4.4 Lightweight Pre-filtering

The design of the filter revolves around the design of the classification features, the automatic feature-selection algorithm, and the choice of the classifier.

Feature Set. Our features can be extracted via simple static analysis. Although some of them are inspired by previous work (e.g., [15,21]), we propose novel features. In particular, features that capture the app behavior, package name heuristics, file types and count, obfuscation, domain name "well-formedness" and reachability, and commands executed through `Runtime.exec()`. To keep the filter lightweight, the majority of our features are either binary (*i.e.,* presence vs. absence) or numeric. The behavioral features (namely, Called APIs and Lightweight Behavioral Features) express runtime behavior of an app, although we match them statically, at the price of a few more false positives, which are perfectly acceptable given the problem setting.

Permission Features (Binary). Android applications are sandboxed within Linux processes, plus an additional layer of permissions that regulate inter-process communication. Permissions [22,23] are known to be abused by malware to escape the sandbox. Indeed, previous research showed that permissions are distributed differently among goodware vs. malware [19,24], and can certainly be used to recognize goodware from "suspicious" applications.

Lightweight Behavioral Features (Binary, Novel). We developed simple reachability heuristics that determine, statically (from the Smali code) whether the application sends SMS at startup (*i.e.,* onStartup), reads phone data at startup, sends data when receiving an SMS, sends SMS to short numbers used in premium services, calls built in utilities (*e.g.,* su, ls, grep, root, chmod), and so forth. Clearly, these features alone are by no means complete nor perfect for malware detection. However, combined with the others, they help in finding suspicious samples.

Other Binary Features (Novel). We calculate some aggregated features from package names, URLs and use of obfuscation. For example, one feature is whether the package name is composed by only one part, whether the domain of the main package name is valid, the presence of URLs whose domain does not match the main package name, whether ProGuard has obfuscated the source code, and so on. We designed this diverse but simple set of features by manually inspecting several malicious and benign samples.

Numerical Features (Novel). We include numeric features such as the number of files in an APK, its size, number of permissions, activities and services, the average class size, the total number of packages and the number of classes contained only in the main package.

Feature Selection. We ended up with a collection of more than 220 attributes. GreatEatlon automatically selects the first 120 most significant features by gain ratio ranking [25]. The choice of gain ratio as information measure is driven by the use of decision trees and random forests as suitable classifier models, as explained in the next section.

Classifier Model and Training. We tested several classification techniques, including decision trees (J48), random forests, support vector machine (SVM), stochastic gradient descent (SGD), decision tables (DT), and rule learners (JRip, FURIA, LAC, RIDOR). We found that the best trade off between time, accuracy and precision is an ensemble classifier that performs majority voting [26] among a J48 decision tree, a random forest and a decision table. Essentially, it chooses the prediction on which most classifiers agree. A relevant aspect of our design is that we incorporate a cost-sensitive wrapper around each classifier to make false positives (non-goodware mis-classified as goodware) count more than false negatives [27]. This is crucial to give more importance to precision. By empirical tests, we found that the cost to assign to mis-classifications of such type in order

to obtain reasonably high accuracy and very high precision ranges between 16 and 20 times the default mis-classification cost.

5 Experimental Evaluation

In this section, we present the experiments that we performed to evaluate GreatEatlon as well as the dataset we used to test it.

5.1 Experiments

We conducted four experiments to evaluate the ability of GreatEatlon to detect file-encrypting ransomware apps, and three experiments to evaluate the performance of the pre-filter. More precisely, **Experiment 1** evaluates the detection precision between GreatEatlon and the state of the art on dataset of manually vetted ransomware apps known to encrypt files. **Experiment 2** is similar to **Experiment 1**, but on a larger dataset, containing *potential* file-encrypting ransomware. **Experiment 3** evaluates the number of false positives on a dataset of benign apps and generic malware samples. **Experiment 4** evaluates the quality of the image scanner. **Experiment 5** and **6** evaluate the precision and speed of the pre-filter, and **Experiment 7** evaluates the impact of the pre-filter on a large-scale scenario.

5.2 Dataset

We have built 5 distinct data sets to evaluate the various characteristics of GreatEatlon:

- The **Ransomware1** dataset, composed by 75 ransomware samples of which 5 were obtained from "Contagio Mobile" dataset [28] and the rest from VirusTotal Intelligence [29]. We manually vetted these samples to ensure that they actually try to surreptitiously encrypt files.
- The **Ransomware2** dataset, composed by samples downloaded from VirusTotal based on the AV labels. In particular, we queried the database for samples with labels containing the most common ransomware family names, or the generic ``crypto'' keyworkd, filtering out samples with less than 5 positive detections. We expect the dataset to contain both the kind of ransomware we want to analyze and other kinds of malware samples, due to the intrinsic imperfection of AVs.
- The **Malware** dataset, composed by 153,982 malware samples, of which 147,145 obtained from the AndRadar project [30] and 6,837 from the AndroTotal repository [31] (having at least 5 positive detections). This dataset contain malware that we used to test precision and speed of the pre-filter.
- The **ThreateningPicture** dataset, which contains screenshots of threatening messages displayed by real ransomware samples, in English and Russian language. In this dataset we included uncommon font faces with handwritten style, so as to test the capabilities of the Tesseract OCR decoder.

- The **Generic** dataset, composed by 1,239 goodware and generic malware samples gathered both from the Google Play store and alternative markets.
- The **AppScale** dataset, taken from AndRadar, MalGenome, Contagio-Minidump, and the top 1,000 APKs submitted to VirusTotal in between Dec 2014 and Jan 2015.

5.3 Experiment 1: GreatEatlon vs. State of the Art (Benchmark)

We compared the precision of the new encryption detectors of GreatEatlon against those implemented in HelDroid using the **Ransomware1** dataset. HelDroid detected only 35 out of 75 ransomware samples, whereas GreatEatlon detected 74 samples. GreatEatlon is able to detect more samples thanks to the customized taint analysis engine. For instance, many samples create target file paths by combining dynamically obtained strings (e.g., file names as a result of a directory listing operation) with hard-coded ones (e.g., default directory names), or by using only hard-coded names. HelDroid is not able to taint fully hard-coded paths. Consequently, even if the malware composes the target path using a mix of hard-coded and dynamically obtained paths, the resulting path will not be tainted because the composition itself would cancel any existing taint. Conversely, GreatEatlon can detect this data flow because the taint is generated only when the application retrieves files in bulk, and not when it obtains a reference to one particular folder.

The false negative was caused by the fact that `MainActivity`, which contain flow sources, is placed as a public inner class of the device-admin class. Unfortunately FlowDroid does not support nested classes, and therefore it is unable to detect flow sources originating from them. Clearly, this is a simple technical limitation of FlowDroid, by no means affecting the conceptual validity of our approach.

5.4 Experiment 2: GreatEatlon vs. State of the Art

We found 11 positives out of 547 analyzed samples. However, only 54 of them were positive to the text detector. If we consider only these 54 samples, we notice that only 43 of them have the `WRITE_EXTERNAL_STORAGE` permission—to write on the SD card (all the aforementioned 11 positives belong to this set). This means that the remaining 504 samples are certainly not file-encrypting ransomware apps. We manually analyzed 10 samples, randomly chosen among the remaining 32 samples confirming that they were true negatives.

5.5 Experiment 3: False Positive Rate

We analyzed the **Generic** dataset to test the false positives rate of GreatEatlon. The results show that our improved detectors do not confuse generic malware with ransomware.

5.6 Experiment 4: Image Scanner Quality

We tested the image scanner on the **ThreateningPicture** dataset. GreatEatlon was able to extract text and classify it as threatening from all the original pictures (*i.e.,* the ones with original font). Instead, if we consider only the images we created by using uncommon font faces, the detector was able to correctly extract the ones with simple symbols, but failed in recognizing the others (e.g., handwritten style). However, we consider thin, handwritten or other fonts of the like more difficult to read for victims, too, hence our assumptions on the reading and understanding ease for threatening text would be no longer satisfied.

Table 1. Precision, accuracy, and area under the ROC curve of different classifiers.

Classifier(s)	Accuracy	Precision	AUC
J48	93.74%	99.4%	0.979
SGD	90.90%	98.9%	0.916
Decision table	91.83%	99.5%	0.986
Random forests	87.18%	99.6%	0.991
J48 + DT + RF	92.75%	99.6%	0.934
J49 + DT + SGD	93.75%	99.6%	0.956
SGD + DT + RF	91.29%	99.6%	0.941

5.7 Experiment 5: Pre-filtering Precision

We evaluated the pre-filter on **Malware** dataset using the standard 10-fold cross-validation approach. We split the dataset in 10 random sub-samples (9 for training, 1 for validation) and repeated this procedure using each sub-sample exactly once per validation. Table 1 shows that the classification capabilities of our pre-filter are very encouraging, especially considering that the training dataset is not homogeneous (*e.g.,* samples from diverse sources and time frames). Notice that the filter alone should not be used as a malware detector! Since our scope is ransomware detection, as opposed to generic malware detection, misclassified innocuous applications would have been analyzed anyways. The goal of our filter is to reduce their amount vastly, and quickly, as showed in the next experiment.

5.8 Experiment 6: Pre-filtering Speed

As training is performed offline, we are interested in measuring the speed of the actual classification. Each APK goes through unpacking, feature extraction, and then the actual classification. Using the **Malware** dataset, we measured that the actual classification has a negligible impact (milliseconds), and unpacking takes 2.484 seconds on average (median 1.922, 3rd quantile 2.814). The feature-extraction step is the core of the pre-filter. Thus, we measured the execution time

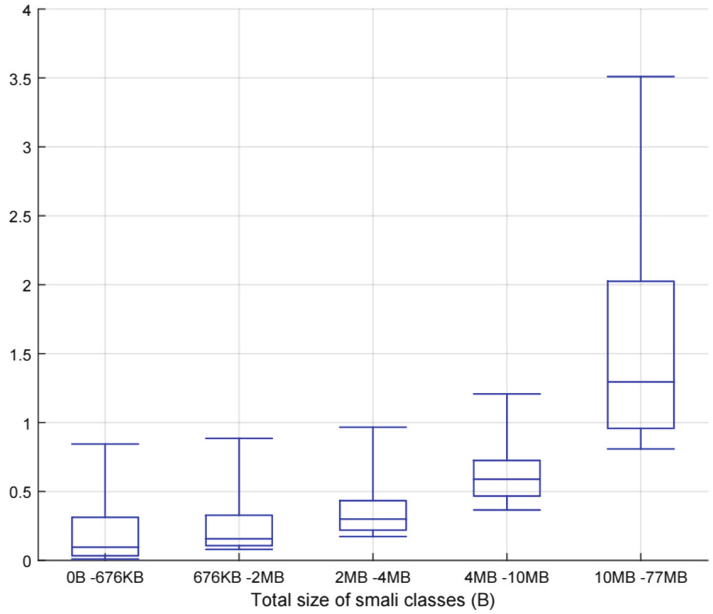

Fig. 1. Pre-filter execution time.

while varying the total size of Smali classes, total Smali classes count, and total files count, and APK size. We found out that time is mainly influenced by the total size of the Smali classes. Therefore, we plot this dependency in Fig. 1. In the worst cases encountered in our large dataset, the feature extraction takes less than 1.5s. Even considering the unpacking, in less than 4 seconds our pre-filter produces an answer.

5.9 Experiment 7: Impact of the Pre-filter on Large Scale Analysis

We measured the response time of HelDroid with and without the pre-filter on 50 distinct random splits of 1,000 samples each from the **AppScale** dataset. Under this scenario, with the pre-filter we pay a small precision penalty but we gain 1.5 to 2.0× on the overall processing time, on average.

6 Limitations

Despite the good performance, GreatEatlon has some limitations, which are described in this section.

6.1 Native Code

GreatEatlon assumes that ransomware will use the Android APIs. Despite the effectiveness of GreatEatlon on the majority of the samples that we analyzed,

it is unable to deal with native machine code, which could be used by malware authors to evade static analyses. Nevertheless, it is possible to discover if a certain sample makes use of native code, or by inspecting the Smali code looking for methods containing the `native` modifier. In this case, it would be appropriate to use an external, dedicated tool to perform the consequent analysis. Therefore, this limitation is not conceptual, but simply technical.

6.2 Conditional Flows

Conditional flows are evaluated while performing the taint analysis. Although in this way we save time and resources, we cannot specify complex conditions, such as conditions related to object fields, or on values known only at runtime. Therefore, if a sample bases its decision of whether to encrypt or decrypt a file on a value that is computed at runtime, GreatEatlon would not be able to detect it. Unluckily, there is no easy solution to both solve this problem with pure static analysis, because the only way to precisely know the value stored inside a certain variable is to watch such variable at runtime.

6.3 Reflection Heuristic

In Sect. 3.3, we anticipated that we designed this heuristic to detect the most common approaches used by ransomware samples. Malware authors could evade it by encrypting strings or other hard-coded values and decrypt them at runtime, as soon as the application needs them. In this scenario, the only way to retrieve those values would be to decrypt them (which requires to know the encryption key), to retrieve the memory dump from a dynamic analysis or to use dynamic techniques such as Harvester (see Sect. 7). Generally, the ultimate solution would be to use a fully dynamic-analysis approach, because it would allow to know with certainty which is the exact signature of the methods that the sample calls, despite of the calling technique.

6.4 Image Scanner

Our image scanner assumes that each threatening text unit is contained inside a single picture. In the future, though, ransomware samples could split the threatening text in multiple images, rendering them in a sort of grid at runtime and composing the complete message in a sort of "mosaic picture," or an animated sequence. In this scenario, GreatEatlon would probably be not able to extract an amount of text significant enough to trigger the text analyzer, specially if the complete threatening message is split in a great number of "tiles".

In order to allow the image scanner to extract meaningful text, we would need to pre-process the layout and Java files to an additional component that is capable of deciding whether a given APK contains such "mosaic" pictures, if that is the case, to reconstruct the final picture. Whenever the resulting image is reconstructed, it can be submitted to our image scanner, that would treat it as a traditional picture and extract any text it contains.

7 Related Work

Ransomware Detection. HelDroid represents the state of the art in the field of static ransomware detection and is the ground on which GreatEatlon is based. HelDroid mainly contributed with three techniques: the text analyzer, which is in charge of deciding whether a given string should be considered as threatening or not, the lock detector, which is able to detect active action made by ransomware that are trying to lock the user out of his phone, and the encryption detector, which is able to detect whether an application is trying to do an unsolicited encryption operations. In particular, we improved the encryption detection system and we added a lightweight pre-filter that is able to recognize goodwares, and discard it from the analysis queue.

Runtime Values Extraction. Harvester [32] is a new system intended to dynamically extract runtime values. Since it is based on a cyclical combination of slicing and code execution, it can extract values regardless of any possible encryption, obfuscation or other anti-analysis techniques applied to them. Given its extraction capabilities, we think that this tool could be integrated into GreatEatlon to replace the heuristic we developed, in order to improve both the device administrator abuse and the encryption detectors as well as FlowDroid's recall (as demonstrated by paper authors, Harvester improved FlowDroid detection by roughly three times).

8 Conclusions

Ransomware is a real threat for mobile devices and is expected to grow in the next years. As a countermeasure against this threat, we propose an approach for detecting encryption-capable apps based on a customized static taint analysis tool, allowing it to accept or discard taint flows based on a set of static conditions. We also designed and developed a new component to detect device administration API abuse, a feature that is present in the newest ransomware families as well as in other recent, non-ransomware malware families.

Our experiments show that GreatEatlon can identify more encryption-capable apps than the state of the art, while maintaining a low false positive rate. Moreover, the new device administration API abuse detector allow us to identify modern ransomware families with better precision. In particular, in the experiment relative to the pre-filter show that is possible to detect goodware with around 99% accuracy.

We believe that merging these new components with the (already good) text analyzer and lock detector of HelDroid can lead to improved and fast detection of modern mobile ransomware families, making GreatEatlon the most advanced mobile ransomware detection tools. If we could also include some external tools such as Harvester or other runtime values extractor, then we would be able to deal with obfuscated, encrypted or other kinds of evasive samples, too.

Finally, we provide a publicly accessible website, which allows other security researchers or end users to submit their samples, to focus on prevention and fight the mobile ransomware threat.

References

1. Statista: Number of smartphone users worldwide from 2014 to 2019, August 2015. http://www.statista.com/
2. Ericcson: Mobility report, February 2016. http://www.ericsson.com/
3. G Data: G data mobile malware report (2015). https://www.gdatasoftware.com/
4. K Lab: The volume of new mobile malware tripled in 2015, February 2016. http://www.kaspersky.com/
5. Avast Software: Avast ransomware removal, June 2014. https://play.google.com/
6. Andronio, N., Zanero, S., Maggi, F.: HelDroid: dissecting and detecting mobile ransomware. In: Bos, H., Monrose, F., Blanc, G. (eds.) RAID 2015. LNCS, vol. 9404, pp. 382–404. Springer, Cham (2015). doi:10.1007/978-3-319-26362-5_18
7. Spreitzenbarth Mobile Security and Forensics: Summary of the year 2015, January 2016. http://forensics.spreitzenbarth.de/
8. Symantec: Simplocker: first confirmed file-encrypting ransomware for android, June 2014. http://www.symantec.com/
9. Avast: Mobile crypto-ransomware simplocker now on steroids, February 2015. http://www.symantec.com/
10. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: user attention, comprehension, and behavior. In: SOUP'S 2012 Proceedings of the Eighth Symposium on Usable Privacy and Security, no. 3 (2012)
11. ESET: Eset simplocker decryptor, August 2014. http://www.eset.com/
12. Apktool v2.0.3. https://github.com/iBotPeaches/Apktool
13. Venkatesan, D.: Android nougat prevents ransomware from resetting device passwords, July 2016. http://www.symantec.com/connect/blogs/android-nougat-prevents-ransomware-resetting-device-passwords
14. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. NDSS **25**(4), 50–52 (2012)
15. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: effective and explainable detection of android malware in your pocket. In: NDSS (2014)
16. 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, pp. 13–24. ACM (2013)
17. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: Andromaly: a behavioral malware detection framework for android devices. J. Intell. Inf. Syst. **38**(1), 161–190 (2012)
18. Apvrille, L., Apvrille, A.: Pre-filtering mobile malware with heuristic techniques. In: Proceedings of GreHack (2013)
19. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: IEEE Symposium on Security and Privacy (SP) 2012, pp. 95–109. IEEE (2012)
20. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., Platzer, C.: Andrubis-1,000,000 apps later: a view on current android malware behaviors. In: Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS) (2014)
21. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 259–269 (2014)

22. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627–638. ACM (2011)
23. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: user attention, comprehension, and behavior. In: Proceedings of the Eighth Symposium on Usable Privacy and Security, p. 3. ACM (2012)
24. Andrubin. https://anubis.iseclab.org
25. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. Elsevier, Amsterdam (2011)
26. Jarvis, K.: Cryptolocker ransomware. Viitattu **20**, 2014 (2013)
27. Domingos, P.: Metacost: a general method for making classifiers cost-sensitive. In: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 155–164. ACM (1999)
28. Contagio mobile. http://contagiominidump.blogspot.it/
29. Virustotal. https://virustotal.com/
30. Lindorfer, M., Volanis, S., Sisto, A., Neugschwandtner, M., Athanasopoulos, E., Maggi, F., Platzer, C., Zanero, S., Ioannidis, S.: AndRadar: fast discovery of android applications in alternative markets. In: Dietrich, S. (ed.) DIMVA 2014. LNCS, vol. 8550, pp. 51–71. Springer, Cham (2014). doi:10.1007/978-3-319-08509-8_4
31. Maggi, F., Valdi, A., Zanero, S.: Andrototal: a flexible, scalable toolbox and service for testing mobile malware detectors. In: Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM). ACM, November 2013
32. Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E.: Harvesting runtime values in android applications that feature anti-analysis techniques. In: Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS) (2016)