HelDroid: Dissecting and Detecting Mobile Ransomware

Nicoló Andronio, Stefano Zanero, and Federico Maggi(✉)

DEIB, Politecnico di Milano, Milano, Italy
nicolo.andronio@mail.polimi.it,
{stefano.zanero,federico.maggi}@polimi.it

Abstract. In ransomware attacks, the actual target is the human, as opposed to the classic attacks that abuse the infected devices (e.g., botnet renting, information stealing). Mobile devices are by no means immune to ransomware attacks. However, there is little research work on this matter and only traditional protections are available. Even state-of-the-art mobile malware detection approaches are ineffective against ransomware apps because of the subtle attack scheme. As a consequence, the ample attack surface formed by the billion mobile devices is left unprotected.

First, in this work we summarize the results of our analysis of the existing mobile ransomware families, describing their common characteristics. Second, we present HelDroid, a fast, efficient and fully automated approach that recognizes known and unknown scareware and ransomware samples from goodware. Our approach is based on detecting the "building blocks" that are typically needed to implement a mobile ransomware application. Specifically, HelDroid detects, in a generic way, if an app is attempting to lock or encrypt the device without the user's consent, and if ransom requests are displayed on the screen. Our technique works without requiring that a sample of a certain family is available beforehand.

We implemented HelDroid and tested it on real-world Android ransomware samples. On a large dataset comprising hundreds of thousands of APKs including goodware, malware, scareware, and ransomware, HelDroid exhibited nearly zero false positives and the capability of recognizing unknown ransomware samples.

1 Introduction

Theorized back in 1996 [1], ransomware attacks have now become a reality. A typical ransomware encrypts the files on the victim's device and asks for a ransom to release them. The miscreants implement various extortion tactics (as explained in Sect. 2), which are both simple and extremely effective. In the "best" case, the device is locked but data is actually left in place in untouched; in the worst case, personal data is effectively encrypted. Therefore, even if the malware is somehow removed, in absence of a fresh backup, the victims have no other choice than paying the requested ransom to (hope to) regain access to their data. McAfee Labs [2] and the FBI [3] recently concluded that the ransomware trend

H. Bos et al. (Eds.): RAID 2015, LNCS 9404, pp. 382–404, 2015.
DOI: 10.1007/978-3-319-26362-5_18

is on the rise and will be among the top 5 most dangerous threats in the near future.

In parallel, mobile malware is expanding quickly and steadily: McAfee Labs recently reported a 100 % growth in Q42014 since Q42013 [2, p.28], VirusTotal receives hundred of thousands of Android samples every week[1], making them the fourth most submitted file type. Unfortunately, mobile devices are not immune by ransomware. A remarkable wave infected over 900,000 mobile devices in a single month *alone* [4]. Moreover, Kaspersky Labs [5] tracked a another notable mobile campaign, revealing a well-structured distribution network with more than 90 hosts serving the malicious APKs, more than 400 URLs serving the exploits, one controller host, and two traffic-driving networks. Alarmingly, the cyber criminals are one step ahead of the defenders, already targeting mobile users. Given the wide attack surface offered by mobile devices along with the massive amount of sensitive data that users store on them (e.g., pictures, digital wallets, contacts), we call for the need of mobile-specific ransomware countermeasures. Our goal in this paper is to make a first step in this direction.

Current Solutions. To the best of our knowledge, current mitigations are commercial cleanup utilities implementing a classic signature-based approach. For example, SurfRight's HitmanPro.Kickstart [6] is a bootable USB image that uses a live-forensics approach to look for artifacts of known ransomware. Other tools such as Avast's Ransomware Removal [7] (for Android) release the ransomed files by exploiting the naïve design of certain families (i.e., SimpLocker) to recover the encryption key, which fortunately is not generated on a per-infection basis. The research community knows very well that such approaches lack of generality. Also, they are evidently limited to known samples, easy to evade, and ineffective against new variants. From the users' perspective, signature-based approaches must be constantly updated with new definitions, and are rarely effective early.

Research Gap. To our knowledge, no research so far have tackled this emerging threat. Even state-of-the-art research approaches (e.g., [8]), which demonstrated nearly-perfect detection and precision on non-ransomware Android malware, recognized only 48.47 % of our ransomware dataset (see Sect. 8). The reason is because ransomware schemes are essentially mimicry attacks, where the overall maliciousness is visible only as a *combination* of legitimate actions. For instance, file encryption or screen locking alone are benign, while the combination of *unsolicited* encryption and screen locking is certainly malicious. Thus, it is not surprising that generic malware-detection approaches exhibit a low recall.

Proposed Approach. After manually analyzing a number of samples of Android ransomware variants from all the existing families, our key insight is to recognize specific, distinctive features of the ransomware tactics with respect to all other malware families — and, obviously, to goodware. Specifically, our

[1] https://www.virustotal.com/en/statistics/.

approach is to determine whether a mobile application attempts to *threaten* the user, to *lock* the device, to *encrypt* data — or a combination of these actions.

We implemented HelDroid to analyze Android applications both statically and dynamically. In particular, HelDroid uses static taint analysis and lightweight emulation to find flows of function calls that indicate device-locking or file-encryption behaviors. Our approach to detecting threatening behavior — a core aspect of ransomware — is a learning-based, natural language processing (NLP) technique that recognizes menacing phrases. Although most of our analysis is static, the threatening-text detector does execute the sample in case no threatening text is found in the static files. This allows to support off-band text (e.g., fetched from a remote server).

Overall, HelDroid is specific to the ransomware schemes, but it does not rely on any family in particular. Moreover, the detection features are parametric and thus adaptable to future families. For instance, the taint-analysis module relies on a single configuration file that lists interesting sources and sinks. Similarly, the threatening-text detector supports several languages and new ones can be added with little, automatic training.

Evaluation Results. We tested HelDroid on hundreds of thousands of samples including goodware, generic malware, and ransomware. HelDroid correctly detected all the ransomware samples, and did not confused corner-case, *benign* apps that *resembled* some of the typical ransomware features (e.g., screen locking, adult apps repackaged with *disarmed* ransomware payload). Overall, HelDroid outperformed the state-of-the-art approach for Android malware detection (see Sect. 8). HelDroid performed well also against unknown ransomware samples, missing only minority of cases where the language was not supported out of the box. This was easily fixed with 30 min of work (i.e., find a textbook in Spanish and re-train the NLP classifier). The detection heuristics of HelDroid exhibited only a dozen of false positives over hundreds of thousands non-ransomware apps.

Prototype Release. We provide access to HelDroid through an API (on top of which we implemented a simple Android client), and release our dataset for research purposes: http://ransom.mobi.

Original Contributions. In summary:

- We are the first at looking at the ransomware phenomenon against mobile devices. We provide a retrospective view of the past two years and distill the characteristics that distinguish mobile ransomware from goodware (and from other malware).
- We propose three generic indicators of compromise for detecting Android ransomware activity by recognizing its distinguishing features. The novel aspects of our approach include a text classifier based on NLP features, a lightweight Smali emulation technique to detect locking strategies, and the application of taint tracking for detecting file-encrypting flows.

– We implement (and evaluate) our approaches for the Android platform and open them to the community as a JSON-based API service over HTTP. This is the first public research prototype of its kind.

2 Background and Motivation

Fascinatingly, the idea of abusing cryptography to create extortion-based attacks was first theorized and demonstrated back in 1996 [1]. The authors defined the concept of *cryptovirus* as a *"[malware] that uses public key [...] to encrypt data [...] that resides on the host system, in such a way that [...] can only be recovered by the author of virus"*.

Based on this definition, *ransomware* can be seen as an advanced, coercive cryptovirus. Coercion techniques, also seen in various *scareware* families, include threatening the victim of indictments (e.g., for the detention of pornographic content, child pornography), violation of copyright laws, or similar illegal behavior. In pure scareware, the cyber crooks exploit the fear and do not necessarily lock the device or encrypt any data. In pure ransomware, before or after the threatening phase the malware actually locks the device and/or encrypts sensitive content until the ransom is paid, usually through money transfer (e.g., MoneyPak, MoneyGram) or crypto currencies. Although digital currency was not used in practice back in 1996, curiously, Young and Yung [1] foresaw that *"information extortion attacks could translate in the loss of U.S. dollars if electronic money is implemented."* Notably, CryptoLocker's main payment mechanism is, in fact, Bitcoin [9,10].

2.1 Motivation

Noticing the rapid succession of new families of mobile ransomware, as summarized in Table 1, we downloaded and manually reverse engineered a few samples for each family, noticing three, common characteristics. From this manual analysis we hypothesize that these independent characteristics are representative of the typical mobile ransomware scheme and can be combined in various ways to categorize a sample as scareware, ransomware, or none of the previous.

Device Locking. All families doing device locking use one among these three techniques. The main one consists in asking for device-administration rights and then locking the device. Another technique is to superimpose a full-screen alert dialog or activity. The third technique consists in trapping key-pressure events (e.g., home or back button), such that the victim cannot switch away from the "lock" screen.

Data Encryption. Some samples do not actually have any encryption capability, even if they claim so; alternatively, they may include encryption routines that are however never called. To our knowledge, only the SimpLocker family

Table 1. Timeline of known Android ransomware or scareware families (we exclude minor variants an aliases). E = Encrypt, L = Lock, T = Threaten.

First Seen	Name	Extort	E	L	T	Target and notes
May 2014	Koler (Reveton) [5]	$300	✗	✓	✓	Police-themed screen lock; localized in 30 countries; spreads via SMS
Jun 2014	Simplocker [12]	$12.5[a]	✓	✓	✓	All files on SD card; uses hardcoded, non-unique key
Jun 2014	Svpeng [13]	$200	✗	✓	✓	Police-themed screen lock
Aug 2014	ScarePackage [14]	$100	✗	✓	✓	Can take pictures and scan the device for banking apps or financial details
Early 2015	New Simplocker [11]	$200	✓	✓	✓	Per-device keys; advanced C&C

[a]Corresponding to, approximately, 260 UAH.

currently implements file encryption. In the first version, the encryption key was hardcoded, whereas the second version [11] generates a per-device key. Nevertheless, samples of this family never call any decryption routine, arguably leaving data permanently unavailable even after payment (unless unlocking is performed through a separate app).

Threatening Text. All current families display threatening messages of some sort. We noticed that families localized in English rely on MoneyPak for payments, whereas families localized in Russian accept credit cards as well.

2.2 Goals and Challenges

Having considered the threat posed by ransomware, the potential attack surface comprising billions of Internet-connected devices and the limitations of current countermeasures, our high-level goal is to overcome the downsides of signature-based approaches, and recognize both known and novel ransomware variants robustly by generalizing the insights described in Sect. 2.1.

Achieving our goal is challenging. Recognizing variants requires a robust model of their characterizing features that has both generalization and detection capabilities, in order to catch both new and known ransomware implementations, possibly in an adaptive way. For example, the task of modeling and recognizing threatening text must account for localization, creating a model that can be quickly re-adapted to new languages *before* new ransomware campaigns start spreading. Similar observations apply to other characterizing features.

2.3 Scope and Assumptions

Although ransomware detection is by no means tied exclusively to the mobile world, in this work we focus on Android ransomware. Mobile ransomware is

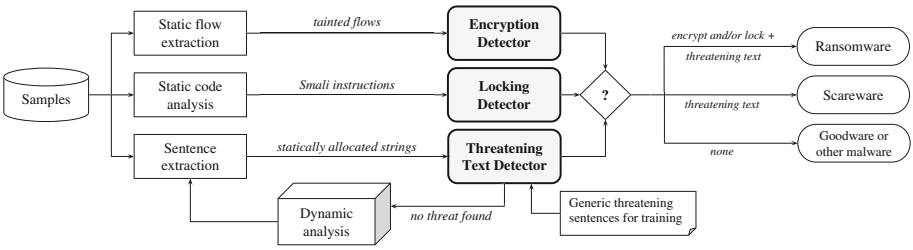

Fig. 1. Android samples are statically analyzed for extracting "artifacts" that typical of ransomware tactics (encryption, locking, threatening). If no threatening text is found, off-band text is analyzed by collecting strings allocated by the sample while running in an instrumented sandbox.

indeed evolving quickly, with 5 families in less than one year (May 2014–Jan 2015), and self-replicating capabilities since the first release.

We focus on challenges that are unique to the ransomware-detection problem (i.e., detecting locking strategies, encryption operations and threatening messages). In this work, we set aside: related problems already tackled by current or past research such as (anti-)evasion techniques, or other aspects that are typical of malicious software *in general*). In Sect. 7 we discuss the impact of our choices.

3 HelDroid's Approach

In this section we describe how, at a *conceptual* level, HelDroid analyzes each Android APK file to decide whether it is a ransomware sample.

As summarized in Fig. 1, we employ three, independent detectors, which can be executed in parallel. Each detector looks for a specific indicator of compromise typical of a ransomware malware. The **Threatening Text Detector** uses text classification to detect coercion attempts (Sect. 3.1). If the result of this classifier is positive, but the others are not, we label the sample as "scareware". This means that the application limits itself to displaying some threatening text to convince the victim in doing some action. If also the **Encryption Detector** (Sect. 3.2) and/or the **Locking Detector** (Sect. 3.3) are triggered, this means that the application is actively performing either action on the infected device. In this case, we label the sample as "ransomware". We designed deterministic decision criteria based on static analysis to detect encryption or locking operations. Note that if the **Threatening Text Detector** is not triggered, the sample is discarded and cannot be considered as ransomware or scareware. Although these three detectors could be combined in other ways (e.g., by including weighting), in this work we consider the presence of threatening text as mandatory for a ransomware author to reach her goal. This aspect is discussed thoroughly in Sect. 7.

3.1 Threatening Text Detector

The goal of this analysis is to recognize menacing phrases in statically and dynamically allocated strings (i.e., sequences of printable characters).

Text Extraction. HelDroid first extracts and analyzes static strings by parsing the disassembled code and resource files (e.g., assets, configuration files). If HelDroid detects no threatening text, then it analyzes dynamically allocated strings: It runs the sample in a sandbox, captures a network traffic dump (i.e., PCAP), decodes application-layer protocols (e.g., HTTP) and extracts strings from the resulting data. The sandbox that we employ also extracts strings allocated dynamically (e.g., as a result of a decryption), but none of the current samples used these measures.

Text Classification. To estimate whether a string contains threatening sentences, we use a natural language processing (NLP) supervised classifier. We train it on generic threatening phrases, similar to (and including) those that typically appear in ransomware or scareware samples. More precisely, we train the classifier using phrases labeled by us as *threat*, *law*, *copyright*, *porn*, and *money*, which typically appear in scareware or ransomware campaigns. Note that no ransomware samples are actually needed to train our classifier: All we need are the sentences. As opposed to being able to isolate a sample, knowing the sentences early is easy (e.g., by taking a screenshot or by leveraging reports given by the first victims).

This phase is further detailed in Sect. 4.1. Its **output** is a ternary decision: "ransomware" threatening text (i.e., accusing the user and asking for payment), "scareware" text (i.e., accusing the user), or "none".

Localization. Our NLP classifier supports localization transparently: It tells whether a given sentence is "threatening" in any of the languages on which it has been trained on. In the unlucky case where localized training phrases are unavailable for training, in Sect. 6.2 we show, as a proof of concept, that these can be easily obtained by running automatic translators on existing sentences found in known ransomware or scareware.

Other Sources of Text. From a technical point of view, the text can be displayed via other means than strings (e.g., images). However, we focus on the core problem, which is that of deciding whether a *text* contains threatening phrases. As discussed in Sect. 7, extracting text from images and videos is easily performed with off-the-shelf OCR software. Recall that, among the goals of the attacker, the ransom-requesting message must be readable and understandable by the victim: It is thus against his or her goals to try to evade OCRs, making the text difficult to read as a side effect.

3.2 Encryption Detector

We check whether the (disassembled) code of the sample under analysis contains traces of unsolicited file-encryption operations.

Unsolicited file-encryption operations are usually implemented by reading the storage (e.g., external storage), looping over the files, invoking encryption routines on each of them, and deleting the original files. Therefore, we are interested in finding execution flows that originate from file-reading operations and terminate into encryption routines. To this end, we rely on a fast, static taint-analysis technique to track flows originating from functions that access the storage (e.g., `getExternalStorageDirectory()`), ending into functions that write encrypted content and delete the original files (e.g., `CipherOutputStream`, `delete()`). We are well aware that a malware author can embed cryptographic primitives rather than using the Android API. Fortunately, recent research [15,16] has already tackled this problem.

Details aside, the **output of this phase** is a binary decision on whether there are significant traces of unsolicited file-encryption operations or not.

3.3 Locking Detector

We check if the application under analysis is able to lock the device (i.e., to prevent navigation among activities). This can be achieved in many ways in Android, including the use of the native screen locking functionality, dimming, immortal dialogs, and so forth. Focusing on the most common techniques that we encountered in real-world Android ransomware we designed a series of heuristics based on lightweight emulation, which can be extended to include other locking techniques in the future.

The most common technique to enact device locking consists in inhibiting navigation among activities through the *Home* and *Back* buttons. This is achieved by handling the events that originate when the user clicks on such buttons on the phone and preventing their propagation. The net result is that the ransomware application effectively forces the device to display an arbitrary activity. Another technique consists in asking the user to let the application become a device administrator, thus allowing it to lock the device. This functionality is part of Android and is normally used for benign purposes (e.g., remote device administration in enterprise scenarios).

To detect if any of these locking technique is executed, we implemented a static code-analysis technique, described in Sect. 4.3. Essentially, we track each Dalvik instruction, including method calls, and check whether there exists an execution path that matches a given heuristic. We created one heuristic per locking strategy. For example, we verify whether the event handler associated to the *Home* button returns always `true`, which means that the event handling cannot propagate further, resulting in a locked screen.

Details aside, the **output of this phase** is a binary decision on whether there are significant traces of device-locking implementations or not.

The overall **final output of** HELDROID, obtained by aggregating the outputs of the three detectors, is a ternary decision: ransomware, scareware, or none.

4 System Details

This section describes the details of HelDroid. The technical implementation details are glanced in Sect. 5.

4.1 Threatening Text Detector Details

We use a supervised-classification approach that works on the text features extracted as follows:

1. **Language Detection:** a simple frequency analysis determines the language of the text (see Sect. 5.1 for the implementation details).
2. **Sentences Splitting:** we use a language-specific segmenter that splits the text into *sentences*.
3. **Stop-word Removal:** we remove all stop words (e.g., "to", "the", "an", "and").
4. **Stemming:** we reduce words to their *stems* (e.g., "fishing," "fished," and "fisher" become "fish").
5. **Stem Vectors:** We map each sentence to a set of *stem vectors*, which are binary vectors that encode which stems are in the sentence.

In *training* mode, each stem vector t is stored in a training set T. At *runtime*, the stem vectors obtained from the app under analysis are used to query the classifier, which answers "ransomware," "scareware," or "other," based on the following scoring algorithm.

Scoring. As suggested in the text-classification literature [17], scoring is based on the cosine similarity $s(x,t) \in [0,1]$ between the query stem vector x and every $t \in T$. Since we operate in a boolean space, it can be reduced to $s(\hat{x},\hat{t}) = \frac{|\hat{x} \cap \hat{t}|}{\sqrt{|\hat{x}|} \cdot \sqrt{|\hat{t}|}}$, where $\hat{x}$ and $\hat{t}$ are the stem sets (i.e., the set data structures that contain strings denoting each stem), which is computed in $O(\min(|\hat{x}|, |\hat{t}|))$.

To score the entire text x, the classifier categorizes its sentences $\forall c \in x$ by maximizing the cosine similarity $s(c,t) \ \forall t \in T$. We denote the score of the best-scoring sentence $c^\star$ as $m(c^\star)$. The best score is calculated within each category. We actually computes two scores, $m(c^\star)_{\text{money}}$ for the best-scoring sentences about "money," and $m(c^\star)_{\text{accusation}}$ for other "accusation" sentences (i.e., threat, law, copyright, porn).

Decision. We label the text as "scareware" if $m_{\text{accusation}}$ exceeds a threshold, and "ransomware" if also m_{money} exceeds. The threshold is set adaptively based on the minimum required score for a sentence to be considered relevant for our analysis. The idea is that short sentences should have a higher threshold, since it is easier to match a greater percentile of a short sentence; instead, longer sentences should have a lower threshold, for the same reason.

Setting thresholds is typically a problematic, yet difficult-to-avoid part of any heuristic-based detection approach. Setting one single threshold is easier, but

makes the decision more sensitive to changes (i.e., one single unit above the threshold could signify a false detection). Therefore, we set *bounds* rather than single threshold values, which we believe leave more room for customization. By no means we claim that such bounds are good for future ransomware samples. As any heuristic-based system, they must be adjusted to keep up with the evolution of the threat under consideration. However, by setting them on the known ransomware samples of our dataset, our experiments show that HelDroid can detect also never-seen-before samples. More details are in Sect. 5.2.

4.2 Encryption Detector Details

Using a static taint-tracking technique, we detect file encryption operations as flows from `Environment.getExternalStorageDirectory()` (1 source) to the `CipherOutputStream` constructor, `Cipher.doFinal` methods, or its overloads (8 sinks). Clearly, tracked flows can involve other, intermediate function calls (e.g., copy data from filesystem to memory, then pass the reference to the buffer to an encryption function, and finally write on the filesystem).

An explanatory example taken from a real-world ransomware sample[2] follows: The underlined lines mark the tracked flow. More sources and sinks can be flexibly added by simple configuration changes, although our results show that the aforementioned ones are enough for current families.

Listing 1.1. Flow source of an encryption operation

```
.class public final Lcom/free/xxx/player/d;

  #...

  .method public constructor <init>(Landroid/content/Context;)V ...

  # getExternalStorageDirectory is invoked to get the SD card root

    invoke-static {},Landroid/os/Environment;->getExternalStorageDirectory()Ljava/io/File;
    move-result-object v0
    invoke-virtual {v0}, Ljava/io/File;->toString()Ljava/lang/String;
    move-result-object v0
    new-instance v1, Ljava/io/File;
    invoke-direct {v1, v0}, Ljava/io/File;-><init>(Ljava/lang/String;)V

    # This invocation saves all files with given extensions in a list
    # and then calls the next method

    invoke-direct {p0, v1}, Lcom/free/xxx/player/d;->a(Ljava/io/File;)V
    return-void
  .end method

  .method public final a()V
    # ...

    # A new object for encryption is instantiated with key
    # 12345678901234567890

    new-instance v2, Lcom/free/xxx/player/a;
    const-string v0, "12345678901234567890"
    invoke-direct {v2, v0}, Lcom/free/xxx/player/a;-><init>(Ljava/lang/String;)V ...

    # If files were not encrypted, encrypt them now

    const-string v3, "FILES_WERE_ENCRYPTED"
    invoke-interface {v2, v3, v0}, Landroid/content/SharedPreferences;->getBoolean(Ljava/lang/String;Z)Z
    move-result v2
    if-nez v2, :cond_1
    invoke-static {}, Landroid/os/Environment;->getExternalStorageState()Ljava/lang/String;
    move-result-object v2
    const-string v3, "mounted"

    # ...

    # Inside a loop, invoke the encryption routine a on file v0, and
    # delete it afterward

    invoke-virtual {v2, v0, v4}, Lcom/free/xxx/player/a;->a(Ljava/lang/String;Ljava/lang/String;)V

    new-instance v4, Ljava/io/File;
    invoke-direct {v4, v0}, Ljava/io/File;-><init>(Ljava/lang/String;)V
    invoke-virtual {v4}, Ljava/io/File;->delete()Z

    # ...
  .end method
.end class
```

[2] MD5: `c83242bfd0e098d9d03c381aee1b4788`.

Listing 1.2. Flow sink of an encryption operation.

```
.class public final Lcom/free/xxx/player/a;

  # ...

  .method public final a(Ljava/lang/String;Ljava/lang/String;)V
    .locals 6

    # A CipherOutputStream is initialized and used to encrypt the file
    # passed as argument, which derives from an invocation to

    new-instance v0, Ljava/io/FileInputStream;
    invoke-direct {v0, p1}, Ljava/io/FileInputStream;-><init>(Ljava/lang/String;)V
    new-instance v1, Ljava/io/FileOutputStream;

    invoke-direct {v1, p2}, Ljava/io/FileOutputStream;-><init>(Ljava/lang/String;)V
    iget-object v2, p0, Lcom/free/xxx/player/a;->a:Ljavax/crypto/Cipher;

    const/4 v3, 0x1
    iget-object v4, p0, Lcom/free/xxx/player/a;->b:Ljavax/crypto/spec/SecretKeySpec;

    iget-object v5, p0, Lcom/free/xxx/player/a;->c:Ljava/security/spec/AlgorithmParameterSpec;

    invoke-virtual {v2, v3, v4, v5}, Ljavax/crypto/Cipher;>init(ILjava/security/Key;Ljava/security/spec/AlgorithmParameterSpec;)V

    new-instance v2, Ljavax/crypto/CipherOutputStream;
    iget-object v3, p0, Lcom/free/xxx/player/a;->a:Ljavax/crypto/Cipher;
    invoke-direct {v2, v1, v3}, Ljavax/crypto/CipherOutputStream;-><init>(Ljava/io/OutputStream;Ljavax/crypto/Cipher;)V

    # ...
  .end method
.end class
```

If any of these flows are found, HelDroid marks the sample accordingly.

4.3 Locking Detector Details

As a proof of concept, we implement a detection heuristic for each of the three most common screen-locking techniques found in Android ransomware.

- Require **administration privileges** and call `DevicePolicyManager.lockNow()`, which forces the device to act as if the lock screen timeout expired.
- **Immortal Activity.** Fill the screen with an activity and inhibit navigation through back and home buttons by overriding the calls to `onKeyUp` and `onKeyDown`. Optionally, the activity cover the software-implemented navigation buttons if the application declares the `SYSTEM_ALERT_WINDOW` permission.
- **Immortal Dialog.** Show an alert dialog that is impossible to close and set a flag in the window parameters.

Detecting whether an app calls the `lockNow` method is easy. We start from searching for the specific permission bit (`BIND_DEVICE_ADMIN`) in the manifest. If found, we parse the Smali assembler code of the application until we find a call to the `lockNow` method.

For the immortal activity technique we are interested in the handling of the `onKeyDown` and `onKeyUp` methods, which are called when a key is pressed or released. They accept as first argument a parameter `p1` containing the numeric code of target key; their return value determines whether the event is considered handled or not (i.e., whether to pass the same event to other underlying View components). An example[3] follows.

[3] MD5 `b31ce7e8e63fb9eb78b8ac934ad5a2ec`.

Listing 1.3. Locking operation example.

```
.method public onKeyDown(ILandroid/view/KeyEvent;)Z
  .locals 1

  # p1 = integer with the key code associated to the pressed key.

  const/4 v0, 0x4 # 4 = back button
  if-ne p1, v0, :cond_0
  iget-object v0, p0, Lcom/android/x5a807058/ZActivity;->q:Lcom/android/zics/ZModuleInterface;

  if-nez v0, :cond_0
  iget-object v0, p0, Lcom/android/x5a807058/ZActivity;->a:Lcom/android/x5a807058/ae;

  # we track function calls as well invoke-virtual {v0},

  Lcom/android/x5a807058/ae;->c()Z :cond_0

  const/4 v0, 0x1 # True = event handled -> do not forward
  return v0
.end method
```

We first locate the `onKeyDown` and `onKeyUp` methods and parse their Smali code. Then we proceed by performing a lightweight Smali emulation. Essentially, we parse each statement and "execute" it according to its semantic. The goal is to verify the existence of an execution path in which the return value is *true*. We examine those `if` statements that compare `p1` with constant integer values. Our emulation technique tracks function calls as well.

Similarly, we detect immortal dialogs by checking if `FLAG_SHOW_WHEN_LOCKED` is set when calling `Landroid/view/Window;->setFlags` in an any `AlertDialog` method, usually in the constructor, and that the same dialog is marked as uncancelable via `setCancelable(false)`.

The immortal activity and dialog techniques can be implemented with a Window instead of an Activity or Dialog object, but we consider this extension exercise for the reader.

5 Implementation and Technical Details

This section describes the relevant technical details of HelDroid.

5.1 Natural Language Processing

We implement the **Threatening Text Detector** on top of *OpenNLP*, a generic, extensible, multi-language NLP library. The sentence splitter and the stemmer [18] are language specific: Adding new languages simply requires training on an arbitrary set of texts provided by the user. For example, we added Russian by training it on a transcript of the *XXVI Congress of the CPSU and Challenges of Social Psychology*[4] and a Wikipedia article about law[5]. In addition, Sect. 6.2 we show how to add new languages to the threatening text classifier.

Our stop-words lists come from the *Stop-words Project*[6]. The language identification is performed with the Cybozu open-source library [19], released and maintained since 2010.

[4] http://www.voppsy.ru/issues/1981/816/816005.htm.
[5] https://ru.wikipedia.org/wiki/.
[6] https://code.google.com/p/stop-words/.

5.2 Text Classification Thresholding

To determine whether the score m of a sentence with respect to the accusation or money categories we proceed as follows. More formally, we want to determine whether $m_{\text{accusation}}$ or m_{money} exceed a threshold. In doing this, we account for the contribution of all sentences (and not only the best scoring ones).

For example, consider the sentences: *"To unlock the device you need"* ($m = 0.775$), m = *"to pay 1,000 rubles"* ($m = 0.632$), and *"Within 24 h we'll unlock your phone"* ($m = 0.612$). The maximum score is 0.775, but since there are other relevant sentences this value should be increased to take them into account. To this end, we increase the score m as follows:

$$\hat{m} = m + (1 - m) \cdot \left(1 - e^{-\sum_{i=1}^{n} (s(c_i) - t(c_i))}\right)$$

where $s(c) - t(c)$ is capped to zero, n is the number of sentences in that category set, c_i the i-th sentence in the stem vector c, and $t : c \mapsto [0, 1]$ is an adaptive threshold function.

Let us pretend for a moment that $t(c)$ is not adaptive, but set to 0.6. Then the sum of $s(c) - t(c)$ is $0.032 + 0.012 = 0.044$. As you can see, $\hat{m}$ is not very different from m because the scores of second and third sentence are just slightly above their detection threshold.

Instead, the idea behind $t(c)$ is that short sentences should have a higher threshold, since it is easier to match a greater percentile of a short sentence; instead, longer sentences should have a lower threshold, for the dual reason:

$$t(c) = \tau_{max} - \gamma(c) \cdot (\tau_{max} - \tau_{min}),\ \gamma(c) = \frac{\sum_{c_i \in c} c_i - \sigma_{min}}{\sigma_{max} - \sigma_{min}}$$

with $\gamma(c)$ capped in $[0, 1]$. The summation yields the number of 1 s in the stem vector of sentence c. σ_{min} and σ_{max} are constants that represent the minimum and maximum number of stems that we want to consider: sentences containing less stems than σ_{min} will have the highest threshold, while sentences containing more stems than σ_{max} will have the lowest threshold. Highest and lowest threshold values are represented by τ_{min} and τ_{max}, which form a threshold bound.

These parameters can be set by first calculating the score of all the sentences in the training set. Then, the values are set such that the classifier distinguishes the ransomware in the training set from generic malware or goodware in the training set. Following this simple, empirical procedure, we obtained: $\tau_{min} = 0.35$, $\tau_{max} = 0.63$, $\sigma_{min} = 3$, and $\sigma_{max} = 6$.

5.3 Dynamic Analysis

If no threatening text is found in statically allocated strings, we attempt a last-resort analysis. In an emulator, we install, run and let the sample run for 5'. After launching the app, our emulator follows an approach similar to the one adopted by TraceDroid [20]: It generates events that simulate user interaction,

rebooting, in/out SMS or calls, etc. Aiming for comprehensive and precise user-activity simulation and anti evasion is out from our scope. From our experience, if the C&C server is active, in a few seconds the sniffer captures the data required to extract the threatening text, which is displayed almost immediately.

From the decoded application-layer traffic (e.g., HTTP), HelDroid parses printable strings. In addition to parsing plaintext protocols from network dumps, every modern sandbox (including the one that we are using) allows to extract strings passed as arguments to functions, which are another source of threatening text. Although we do not implement OCR-text extraction in our current version of HelDroid, we run a quick pilot study on the screenshots collected by TraceDroid. Using the default configuration of `tesseract` we were able to extract all the sentences displayed on the screenshots.

5.4 Static Code Analysis

We extract part of the features for the **Threatening Text Detector** by parsing the manifest and other configuration files found in the APK once uncompressed with `akptool`[7]. We compute the remaining ones by enumerating count, type or size of files contained in the same application package.

However, the most interesting data requires an analysis of the app's Dalvik code in its Smali[8] text representation generated by `apktool`. For the **Locking Detector**, instead of using *SAAF* [21], which we found unstable in multi-threaded scenarios, we wrote a simple emulator that "runs" Smali code, tailored for our needs. To keep it fast, we implemented the minimum subset of instructions required by our detector.

For the **Encryption Detector** we need precise flows information across the entire Smali instruction set. For this, we leveraged *FlowDroid* [22], a very robust, context-, flow-, field-, object-sensitive and lifecycle-aware static taint-analysis tool with great recall and precision. Source and sink APIs are configurable.

6 Experimental Validation

We tested HelDroid, running on server-grade hardware, against real-world datasets to verify if it detected known and new ransomware variants and samples. In summary, as discussed further in Sect. 8, it outperformed the state-of-the-art research tool for Android malware detection.

6.1 Datasets

We used a diverse set of datasets (Table 2), available at http://ransom.mobi.

[7] https://code.google.com/p/android-apktool/.
[8] https://code.google.com/p/smali/.

Table 2. Summary of our datasets. VT 5+ indicates that samples that are marked according to VirusTotal's positive results. VT top 400 are on Dec 24th, 2014.

Name	Size	Labelling	Apriori content	Use
AR	172,174	VT 5+	55.3 % malware + 44.7 % goodware	FP eval.
AT	12,842	VT 5+	68.2 % malware + 31.8 % goodware	FP eval.
MG	1,260	Implicit	100 % malware	FP eval.
R1	207	VT 5+	100 % ransomware + scareware	NLP training
R2	443	VT 5+	100 % ransomware + scareware	Detection
M1	400	VT top 400	100 % malware	FP eval.

Goodware and Generic Malware. We obtained access to the **AndRadar (AR)** [23] dataset, containing apps from independent markets (Blackmart, Opera, Camangi, PandaApp, Slideme, and GetJar) between Feb 2011 and Oct 2013. Moreover, we used the public **AndroTotal (AT)** API [24] to fetch the apps submitted in Jun 2014–Dec 2014. Also, we used the **MalGenome (MG)** [25] dataset, which contains malware appeared in Aug 2010–Oct 2011.

We labeled each sample using VirusTotal, flagging as *malware* those with 5+/56 positives. The **AR** and **AT** datasets do not contain any ransomware samples. The **MG** dataset contains only malware (not ransomware).

Last, the **Malware 1 (M1)** dataset contains the top 400 malicious Android applications as of Dec 2014, excluding those already present in the rest of our datasets and any known ransomware.

Known Ransomware (sentences for Text-Classifier Training). We need a small portion of sentences obtained from true ransomware samples. During the early stages of a malware campaign, samples are not always readily available for analysis or training. Interestingly, our text-classifier can be trained regardless of the availability of the sample: All it needs is the threatening text, which is usually easy to obtain (e.g., from early reports from victims).

We built the **Ransomware 1 (R1)** dataset through the VirusTotal Intelligence API by searching for positive (5+) Android samples labeled or tagged as *ransomware*, *koler*, *locker*, *fbilocker*, *scarepackage*, and similar, in Sep–Nov 2014. We manually verified that at least 5 distinct AV programs agreed on the same labels in **R1** (allowing slight lexical variations). In this way, we excluded outliers caused by naming inconsistencies, and could be reasonably safe that the resulting 207 samples were true ransomware. The training is performed only once, offline, but can be repeated over time as needed. We manually labeled sentences (e.g., threat, porn, copyright) from the **R1** dataset, totaling 51 English sentences and 31 Russian sentences.

Unknown Ransomware. Similarly, we built the **Ransomware 2 (R2)** dataset for samples appeared in Dec 2014–Jan 2015. This dataset is to evaluate HelDroid on an arbitrary, never-seen-before, dataset comprising ransomware — and possibly

other categories of malware. Aposteriori, we discovered that this datasets contains interesting corner-case apps that resemble some of the typical ransomware features (e.g., screen locking, adult apps repackaged with *disarmed* ransomware payload), making this a particularly challenging test case.

6.2 Experiment 1: Detection Capability

HelDroid detected all of the 207 ransomware samples in **R1**: 194 with static text extraction, and the remaining 13 by extracting the text in live-captured web responses from the C&C server. However, this was expected, since we used **R1** for training. Thus, this experiment showed only the correctness of the approach.

We tested the true predictive capabilities of HelDroid on **R2**, which is disjoint from **R1**. Among the 443 total samples in **R2**, 375 were correctly detected as ransomware or scareware, and 49 were correctly flagged as neither. Precisely, the following ones were actually true negatives:

- 14 Badoink + 15 PornDroid clones (see below);
- 6 lock-screen applications to modify the system's look &feel;
- 14 benign, adware, spyware, or other non-ransomware threats.

Badoink and PornDroid are benign applications sometimes used as hosts of ransomware payload. HelDroid correctly only flagged the locking behavior. We installed and used such samples on a real device and verified that they were not performing any malicious operation apart from locking the device screen (behavior that was correctly detected). An analysis of network traffic revealed that the remote endpoint of all web requests issued during execution was unreachable, resulting in the application being unable to display the threatening web page.

The last 19 samples are known to AV companies as ransomware, but:

- 11 samples use languages on which HelDroid was not trained (see below).
- 4 samples contain no static or dynamically generated text, thus they were disarmed, bogus or simply incorrectly flagged by the commercial AVs.
- 4 failed downloading their threatening text because the C&C server was down. Strictly speaking, these samples can be safely considered as being disarmed. Manual analysis revealed that these samples belong to an unknown family (probably based on repackaged PornDroid versions).

False Negative Analysis. We focused on the samples that were not detected because of the missing language models. As a proof of concept we trained HelDroid on Spanish, by translating known threatening text from English to Spanish using Google Translator, adding known non-threatening Spanish text, and running the training procedure. The whole process took less than 30 min. After this, all previously undetected samples localized in Spanish were successfully flagged as ransomware.

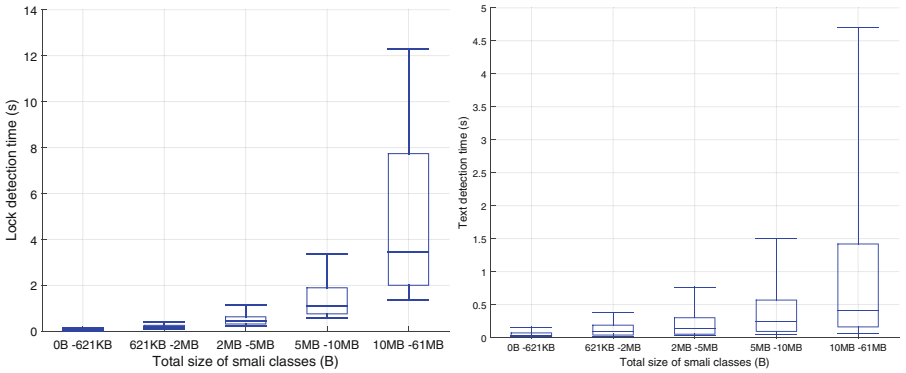

Fig. 2. Lock-detection (left) and text-classification (right) time as function of Smali class size (whiskers at the 9^{th} and 91^{st} percentiles).

6.3 Experiment 2: False Positive Evaluation

A false positive is a generic malware or a goodware sample flagged as ransomware. We first evaluated HelDroid on **M1** (generic malware, no ransomware). No sample in **M1** was flagged by HelDroid as ransomware.

We extended this experiment to the other datasets containing goodware and generic malware (i.e., **AR**, **AT**, **MG**). In the **AR** dataset, which contained both malware and goodware, HelDroid correctly reported zero ransomware samples, whereas in the **AT** dataset only 2 and 7 samples out of 12,842 were incorrectly flagged as ransomware and scareware, respectively. Manual investigation revealed that the 2 false ransomware samples were actually a benign sample and a generic trojan, respectively. Actually, both samples had a locking behavior that was correctly caught by HelDroid. The reason why these were flagged as ransomware is because they contained text localized in all major languages (most of which were different than those currently implemented in HelDroid), which brought the text classifier in a corner case. The 7 false scareware comprised 6 benign apps and 1 Leadbolt adware sample. In all cases, the source of error was an significant amount of text containing threatening-, porn-, law- or copyright-related keywords. Last, in the **MG** dataset, none of the malware samples was incorrectly flagged as ransomware or scareware.

However, we can conclude that the rate of false positives is minuscule compared to the size of the datasets. Moreover, the majority of false positives are actually known goodware, which can be pre-filtered easily with whitelisting.

6.4 Experiment 3: Detection Speed

We measured the speed of each detector component on 50 distinct random splits of **AR** with 1,000 samples each. Figure 2(a) and (b) show that text classification is extremely fast in all cases, while locking strategies detection is the main bottleneck, yet under 4 s on average. The encryption-detection module always took milliseconds.

If HelDroid must invoke the external sandbox to extract dynamically generated text, this takes up to 5 min in our implementation, but this is unavoidable for dynamic analysis. As we showed, however, this happens for a very limited number of samples.

7 Limitations and Future Work

Our results show that HelDroid has raised the bar for ransomware authors. However, there are limitations, which we describe in this section, that we hope will stimulate further research.

Portability. Although we focus on the mobile case, ransomware is a general problem. Porting the HelDroid *approach* to the non-mobile world is non-trivial but feasible. The **Threatening Text Detector** would be straightforward to port, as it only assumes the availability of text. For example, it could be applied *as it* is for filtering scareware emails. The toughest parts to port are those that assume the use of a well-defined API (e.g., for encryption or locking operations). Indeed, a malware author could evade our system by using native code or embedding cryptographic primitives, making porting much more complex. However, the progress on static program analysis (e.g., [26,27]) and reverse engineering (e.g., [28]) of native binary code have produced advanced analysis tools that would ease porting HelDroid to other settings, including the detection of cryptographic primitives in binary code [15,16]. The principles behind our detection modules do not change; only their implementation does.

One last discussion point regards the inspection site. For mobile applications, which are typically vetted prior or upon installation (e.g., by the distributing marketplace, on the device using call-home services such as Google App Verify), HelDroid works "as is." For non-mobile applications that do not follow this distribution model, HelDroid should be integrated into the operating system, in a trusted domain (e.g., kernel, driver). In this application scenario it is crucial that the system is allowed to block the currently executing code to prevent the malicious actions to continue. In HelDroid's terms, this means that the encryption and locking indicators of compromise should have high priority, to avoid cases in which the malware *first* silently encrypts every file and *then* displays the threatening text (when it is already too late).

Internationalization. As we proved in **Experiment 1** by quickly adding Spanish support, we designed HelDroid such that supporting other languages is a trivial task. Languages such as Chinese or Japanese, however, would be trickier than others to implement, due to significant differences in stemming and phrase structure. Fortunately, research prototypes such as Stanford's CoreNLP [29] that support (for instance) Chinese NLP makes this extension feasible with just some engineering work.

Evasion. In addition to the use of native machine code, which we already mentioned above, a simple yet naïve evasion to the static-analysis part of our approach (**Encryption Detector** and **Locking Detector**) consists of a benign APK that dynamically loads the code carrying out the actual attacks [30]. First, we note that this technique can be counter evaded by intercepting the loaded payload and analyzing it in a second round, as previous research have demonstrated [31]. Second, we note that this evasion mechanism is common to any static-based approach, and thus is not specific to HelDroid.

A more interesting discussion regards the threatening text. Text can be displayed via other means than strings (e.g., images, videos, audio), delivered out of band (e.g, e-mail) or obfuscated. A first mitigation, that we partially address, consists in using a sandbox that dumps dynamically allocated text, thus coping with obfuscated strings as well as encrypted application protocols (e.g., HTTPS). For example, Andrubis tracks decryption routines and allow the analyst to access the decrypted content.

Regarding image- or video-rendered text, state-of-the-art optical character recognition (OCR) techniques could be used. Although evasion techniques — such as those used in CAPTCHAs — can be mounted against OCR, the goal of the attacker is to make the text clear and easy to read for the victim, setting a limit to them; also, previous research demonstrated the fallacy of even the most extreme text-distortion techniques adopted by CAPTCHAs [32]. Regarding out-of-band text, our current implementation of HelDroid does not cope with it, although applying our text classifier to incoming email messages is trivial. In general, this strategy may be in contrast with the attacker's goal, that is to ensure that the victim receives the ransom-requesting message. Displaying this message synchronously is an advantage for the attacker, whereas out-of-band communication *alone* is ill suited to the task. For example, the victim may not read email or junk-mail filters could block such messages.

An even more interesting evasion technique is a mimicry attack on our text classifier, which we think is possible. In a nutshell, the attacker must be able to write a text containing a disproportionally large number of unknown words, unusual punctuation or many grammar errors. Unusual punctuation and grammar errors could be mitigated with some lexical pre-processing an advanced corrector. Interestingly, the most recent families (e.g., CBT-Locker) show that the attackers tend to write "perfect" messages, arguably prepared by native speakers, in order to sound more legitimate. After all, careful wording of threatening messages is essential to all social engineering-based attacks.

Future Work. In addition to addressing the aforementioned limitations, future research could focus on designing ransomware-resistant OSs. For example, in the case of Android, calls to encryption routines should be explicitly authorized by the users on a per-file basis. This is not trivial from a usability viewpoint, especially for long sequences of calls. Moreover, many applications may use encryption for benign purposes, making this goal even more challenging.

8 Related Work

Malware Detection. There exist several malware detection approaches, including static [8,33], dynamic [34], and hybrid [25] techniques. DREBIN [8] and MAST [33] are particularly related to our work. DREBIN aims at detecting malware statically, with a 94 % accuracy and 1 % false positives: It gathers features such as permissions, intents, used APIs, network addresses, etc., embeds them in a vector space and trains a support vector machine to recognize malware. MAST relies on multiple correspondence analysis and statically ranks applications by suspiciousness. Thanks to this ranking, it detects 95 % of malware at the cost of analyzing 13 % of goodware.

Unfortunately, generic approaches to malware detection seem unsuitable for ransomware. We tested DREBIN on our **R2** dataset of ransomware. Although DREBIN outperformed AVs, HelDroid outperformed DREBIN (which detected only 48.47 % of the ransomware samples). Even the authors of DREBIN, which we have contacted, in their paper state that their approach is vulnerable to mimicry attacks. Ransomware is a type of mimicry attack, because it composes benign actions (i.e., encryption, text rendering) toward a malicious goal.

Ransomware Detection. To the best of our knowledge, our paper is the first research work on mobile ransomware. The work by Kharraz et al. [35], published after the submission of HelDroid, is the first to present a thorough study on Windows ransomware. After analyzing 1,359 belonging to 15 distinct ransomware families, they present a series of indicators of compromise that characterize ransomware activity at the filesystem layer. This approach, in addition to being focused entirely on the Windows operating system, is complementary to ours. Indeed, we look at how ransomware behaves at the application level, whereas [35] focuses on the low level behavior.

Previous work focused on the malicious use of cryptography for implementing ransomware attacks [1,36]. However, no approaches exist for the explicit detection of this class of malware.

9 Conclusions

A single mobile ransomware family has already affected nearly one million of users [4] in one month. Judging by the most recent families [11] and their rapid evolution pace, this threat will arguably become more and more dangerous, and difficult to deal with. Before HelDroid, the only available tools were signature based, with all of the disadvantages this entails. Instead, we showed that our approach, after being trained on recent ransomware samples, is able to efficiently detect new variants and families. Even with mixed datasets including benign, malicious, scareware, and ransomware apps, HelDroid correctly recognized 99 % never-seen-before samples (375 + 11 + 4 over 394, in a dataset containing also 49 corner-case apps). Interestingly, the remainder 4 were incorrectly

flagged by commercial AVs as ransomware. Thus, it is a first, significant step toward designing proactive detectors that provide an effective line of defense.

HelDroid could be integrated in mobile AVs, which would submit files to our JSON API, as recently proposed in [37]. Alternatively, HelDroid shall be deployed in one or more of the many checkpoints offered by modern app-distribution ecosystems. For instance, HelDroid could be part o the app-vetting processes performed by the online marketplaces, or upon installation (e.g., the Google App Verify service scans apps right before proceeding with installation).

Acknowledgments. We are thankful to the anonymous reviewers and our shepherd, Patrick Traynor, for the insightful comments, Steven Arzt, who helped us improving FlowDroid to track flows across threads, and Daniel Arp from the DREBIN project. This work has been supported by the MIUR FACE Project No. RBFR13AJFT.

References

1. Young, A., Yung, M.: Cryptovirology: extortion-based security threats and countermeasures. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 129–140, May 1996
2. McAfee Labs: Threats report, November 2014. McAfee Labs, November 2014
3. Ransomware on the rise, January 2015. http://www.fbi.gov/news/stories/2015/january/ransomware-on-the-rise
4. Perlroth, N.: Android phones hit by 'Ransomware', August 2014. http://bits.blogs.nytimes.com/2014/08/22/android-phones-hit-byransomware/
5. Lab. Koler - the police ransomware for android, June 2014. http://securelist.com/blog/research/65189/behind-the-android-oskoler-distribution-network/
6. SurfRight. HitmanPro.kickstart, March 2014. http://www.surfright.nl/en/kickstart
7. Avast Software. Avast ransomware removal, June 2014. https://play.google.com/store/apps/details?id=com.avast.android.malwareremoval
8. Arp, D., et al.: Drebin: effective and explainable detection of android malware in your pocket. In: Network and Distributed System Security (NDSS) Symposium, San Diego, California (2014)
9. Spagnuolo, M., Maggi, F., Zanero, S.: BitIodine: extracting intelligence from the bitcoin network. In: Financial Cryptography and Data Security, Barbados, 3 March 2014
10. Jarvis, K.: CryptoLocker ransomware, December 2013. http://www.secureworks.com/cyber-threat-intelligence/threats/cryptolockerransomware/
11. Chrysaidos, N.: Mobile crypto-ransomware simplocker now on steroids, February 2015. https://blog.avast.com/2015/02/10/mobile-cryptoransomware-simplocker-now-on-steroids/
12. Hamada, J.: Simplocker: first confirmed file-encrypting ransomware for android, June 2014. http://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android
13. Unuchek, R.: Latest version of svpeng targets users in US, June 2014. http://securelist.com/blog/incidents/63746/latest-version-ofsvpeng-targets-users-in-us/
14. Kelly, M.: US targeted by coercive mobile ransomware impersonating the FBI, July 2014. https://blog.lookout.com/blog/2014/07/16/scarepakage/

15. Gröbert, F., Willems, C., Holz, T.: Automated identification of cryptographic primitives in binary programs. In: Recent Advances in Intrusion Detection, pp. 41–60 (2011)
16. Lestringant, P., Guihéry, F., Fouque, P.-A.: Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, pp. 203–214, New York, NY, USA (2015)
17. Aggarwal, C.C., Zhai, C.: A survey of text classification algorithms. In: Aggarwal, C.C., Zhai, C. (eds.) Mining Text Data, pp. 163–222. Springer, US (2012)
18. The snowball language. http://snowball.tartarus.org/
19. Shuyo, N.: Language detection library for java (2010). http://code.google.com/p/language-detection/
20. van der Veen, V., Bos, H., Rossow, C.: Dynamic analysis of android malware. VU University Amsterdam, August 2013. http://tracedroid.few.vu.nl/
21. Hoffmann, J., et al.: Slicing droids: program slicing for smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1844–1851, New York, NY, USA (2013)
22. Arzt, S., et al.: 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, pp. 259–269, New York, NY, USA (2014)
23. 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, Heidelberg (2014)
24. Maggi, F., Valdi, A., Zanero, S.: AndroTotal: a flexible, scalable toolbox and service for testing mobile malware detectors. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 49–54, New York, NY, USA (2013)
25. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy, San Francisco, CA, May 2012. http://www.malgenomeproject.org/
26. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: a new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
27. Schwartz, E.J., et al.: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: USENIX security (2013)
28. Slowinska, A., Stancescu, T., Bos, H.: Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA (2011)
29. Manning, C.D., et al.: The stanford Core NLP natural language processing toolkit. In: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55–60 (2014). http://www.aclweb.org/anthology/P/P14/P14-5010
30. Poeplau, S., et al.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), pp. 23–26 (2014)
31. Zhou, W., et al.: Fast, scalable detection of "piggybacked" mobile applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, pp. 185–196, New York, NY, USA (2013)

32. Bursztein, E., Martin, M., Mitchell, J.: Text-based CAPTCHA strengths and weaknesses. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 125–138, New York, NY, USA (2011)
33. Chakradeo, S., et al.: 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, New York, NY, USA (2013)
34. Shabtai, A., et al.: Andromaly: a behavioral malware detection framework for android devices. J. Intell. Inf. Syst. **38**(1), 161–190 (2012)
35. Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., Kirda, E.: Cutting the gordian knot: a look under the hood of ransomware attacks. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) DIMVA 2015. LNCS, vol. 9148, pp. 3–24. Springer, Heidelberg (2015)
36. Young, A.: Cryptoviral extortion using microsoft's crypto API. Int. J. Inf. Secur. **5**(2), 67–76 (2006)
37. Jarabek, C., Barrera, D., Aycock, J.: ThinAV: truly lightweight mobile cloud-based anti-malware. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 209–218, New York, NY, USA (2012)