Heldroid: Fast and Efficient Linguistic-Based Ransomware Detection

BY

NICOLO' ANDRONIO
B.S., Politecnico di Milano, Milan, Italy, 2012

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2015

Chicago, Illinois

To my **parents**, who always believe in me even when I don't

and

To my **friends** and the Cho'Gath Group, who kept me away from silly things like studying

ACKNOWLEDGMENTS

Particular thanks to *Steven Arzt* for helping us fix some FlowDroid (1) issues with cross-thread flows and to *Daniel Arp* for letting DREBIN (2) scan one of our datasets.

Heartfelt thanks to my advisor *Federico Maggi* for constantly pushing me forward toward my goal.

NA

TABLE OF CONTENTS

CHAPTER			PAGE
1	**INTRODUCTION**		1
	1.1	What is ransomware?	1
	1.2	A ferocious uprising	3
	1.3	The problem	4
	1.4	Our proposal	5
	1.5	Our contributions	6
2	**BACKGROUND AND MOTIVATION**		8
	2.1	Ransomware features	8
	2.1.1	Device locking	9
	2.1.2	Data encryption	9
	2.1.3	Threatening text	10
	2.2	Goals and challenges	11
3	**GENERAL APPROACH**		13
	3.1	Filtering	13
	3.2	Ransomware detection	15
	3.2.1	Threatening text detection	15
	3.2.2	Encryption detection	16
	3.2.3	Locking detection	17
4	**APPROACH DETAILS**		18
	4.1	Filtering	18
	4.1.1	Feature set	18
	4.1.2	Feature selection	23
	4.1.3	Classifier algorithm and training	23
	4.2	Threatening text detection	24
	4.3	Encryption detection	29
	4.4	Locking detection	33
5	**IMPLEMENTATION DETAILS**		36
	5.1	Static code analysis	36
	5.2	Natural language processing	37
	5.3	Sandboxing	38
6	**EXPERIMENTAL VALIDATION**		40
	6.1	Datasets and settings	40

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

6.2 Experiment 1: Filter precision 41
6.3 Experiment 2: Filter speed 43
6.4 Experiment 3: Ransomware detection 46
6.5 Experiment 4: Ransomware detection speed 48
6.6 Experiment 5: Impact of the filter 49

7 LIMITATIONS AND FUTURE WORK 51
7.1 Portability 51
7.2 Internationalization 52
7.3 Threatening text evasion 52
7.4 Classifier obsolescence 54
7.5 Encryption aware APIs 54

8 RELATED WORK 55

9 CONCLUSION 57

APPENDICES 58
Appendix A 59
Appendix B 60

CITED LITERATURE 67

VITA 75

LIST OF TABLES

TABLE **PAGE**

I RANSOMWARE FAMILIES TIMELINE 12

II SUMMARY OF DATASETS 42

III PERFORMANCES OF SEVERAL FILTERING CLASSIFIERS . 43

IV COMPLETE LIST OF FEATURES (Before selection). 60

LIST OF FIGURES

FIGURE **PAGE**

1 Screenshot of Reveton ransomware 3

2 PornDroid ransomware asking for administration privileges 10

3 Diagram of the overall system at a conceptual level 14

4 Filter execution time (raw data). 44

5 Filter execution time (boxplot). 45

6 Locking detection time (boxplot) 49

7 Text classification time (boxplot) 49

8 Impact of the filter on the overall system 50

LIST OF ABBREVIATIONS

APK	Android Application Package
API	Application Programming Interface
NLP	Natural Language Processing
DEX	Dalvik Executable
OCR	Optical Character Recognition
AV	Anti-Virus (software)

SUMMARY

In every era, there has always been a slice of society devoted to deception and subtlety. Sadly, but inexorably, progress in technology has changed the approach of criminals to thievery and blackmailing. In our time, information is the real merchandise and it is thus subject to theft as any other goods: credit card numbers, cryptocurrency and illegal digital material are on the top of the list.

Since information is so important, it can also be used to blackmail people. The most recent trend consists in denying victims access to their own files and requiring a relatively large amount of money to restore data back to normal. Applications exhibiting this kind of behaviour have been unsurpisingly named *ransomware* and their keen already produced millions of dollars worth of losses in the past two years. Since they exploit social engineering, there isn't any suitable form of prevention apart from backing up data frequently. Recovery tools may have worked in the past, but as attacks become more and more aggressive and sophisticated, they are not future proof.

Motivated by an in-depth analysis of existing ransomware, the fallacy of currently available detection methods and the absence of any consistent prevention technique, we devised HELDROID, an efficient, fully automated and learning-based approach aimed at recognizing ransomware proactively. We built it by first considering the building blocks of ransomware attacks and then composing them into a tool that actually percieves the intent behind an application behaviour. Our results show that HELDROID is able to correctly identify never-seen-before threats with high precision and speed.

CHAPTER 1

INTRODUCTION

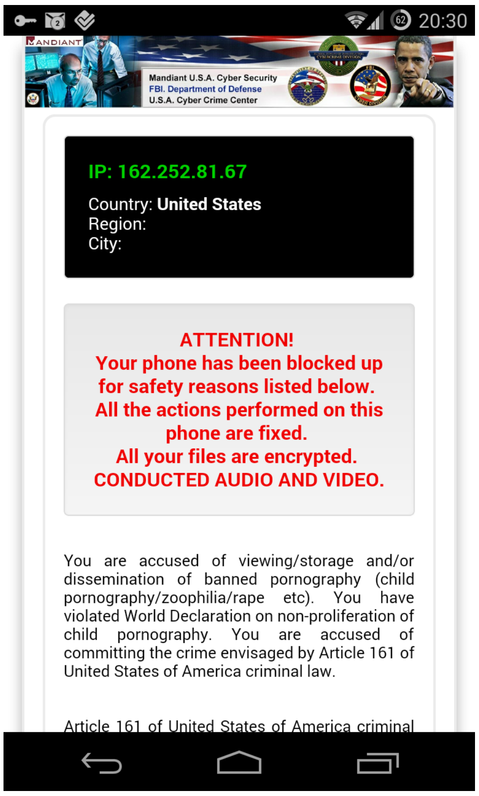
In the last few years, the world of technology witnessed an astonishingly fast growth of the mobile market. The fast-paced rythm of modern life requires tools to keep people updated and aware of the rest of the world in real-time. These tools are what we all know as *apps*, computer programs running on our smartphones that deliver every imaginable flavor of service: from weather forecast to gaming, from home-banking to office utilities, from traffic updates to online dating. However, since the market is so large and the community so diverse, users tend to trust what they see in order not to be overwhelmed by an enourmous flow of information. They choose to install an application solely based on its textual description and a few screenshots, unconditionally bestowing upon the installed program every kind of system permission.

There a is general fluorishing of applications in the mobile markets. Most are useful and harmless, but some is written or repackaged with the intent to steal data or money from users (and, more rarely, solely to harm). We refer to the first class of applications as *goodware* and to the second as *malware*. Because Android systems let users enforce security and those same users often do not fully understand what they are asked, the mobile environment quickly became crowded with malware.

1.1 What is ransomware?

Malware can be further classified depending on its purpose; in this work we focus on two particular categories: **ransomware** and **scareware**. The main goal of *ransomware* is extorting money from users under the threat of locking the users' device and/or permanently encrypting their files, thereby denying them access to their own data. The threat is usually followed by real actions: in this case, the application takes control of the device and will not allow victims to use it or access any file until the ransom is paid, usually through external services like MoneyPak and MoneyGram. Since these actions are extremely invasive, ransomware always produces a series of accusations against the device owners to justify such measures. The most common indictments include detention of pornographic content, pedophilia, zoophilia, sexual bestiality or violation of copyright laws, as shown in Figure 1. However, given the gravity of these accusations, people easily become scared and tend to believe in just about anything: crooks may exploit the fear of severe punishment to trick users into paying realtively high fees without even bothering to actually lock the device or encrypt any data. In this case, we speak of *scareware*.

From hereinafter we implicitly use the terms ransomware and scareware as synonyms, unless explicitly mentioned in the text.



Mandiant U.S.A. Cyber Security
FBI. Department of Defense
U.S.A. Cyber Crime Center

IP: 162.252.81.67

Country: **United States**
Region:
City:

ATTENTION!
Your phone has been blocked up for safety reasons listed below.
All the actions performed on this phone are fixed.
All your files are encrypted.
CONDUCTED AUDIO AND VIDEO.

You are accused of viewing/storage and/or dissemination of banned pornography (child pornography/zoophilia/rape etc). You have violated World Declaration on non-proliferation of child pornography. You are accused of committing the crime envisaged by Article 161 of United States of America criminal law.

Article 161 of United States of America criminal

Figure 1: Screenshot of Reveton ransomware

1.2 A ferocious uprising

Malware is a longstanding problem and *mobile* malware is quickly evolving qualitatively and quantitatively. As reported (3, p.28) by *McAfee Labs*, in Q32014 the total amount of mobile malware grew by more than 100% with respect to Q32013, which is honestly quite impressive despite not really surprising. Along with these dangerously high numbers, *VirusTotal*, one of the most widespread online scanning tool, receives hundreds of thousand Android samples to analyze *every week* (4), with Android applications being the fourth most common file types.

Ransomware is riding the tides of this uprising and is constantly expanding, finding new ways to blackmail users and force them to pay fees in order to retrieve their stolen data. The *New York Times* reported (5) that, only in August 2014, more than 900,000 users have been infected by a single ransomware family called *ScarePackage*. In their recent public security report, *McAfee Labs* state (3, p.30) that the total number of ransomware samples[1] collected so far reaches 2 million units and predict that ransomware will be among the top 5 deadliest meances to cybersecurity in the near future.

1.3 The problem

Given the impressive diffusion of ransomware, it is desirable to have tools able to detect it beforehand. As of now, to the best of our knowledge, there exists only one tool specifically tailored for mobile malware: the *Avast Ransomware Removal* (6) app. Despite its extremely narrow target, this tool employs standard anti-virus technologies, such as signature checking. This is one of the reasons it is also able to restore encrypted files: in fact, many ransomware families encrypt data using known algorithms and fixed hard-coded keys that can be easily found with modest reverse engineering efforts; once the sample family is recognized, it is trivial to recover all encrypted files by simply decrypting them with the recovered key.

[1]Both mobile and non-mobile variants are included in the total count.

Several similar solutions for the PC world exist. Indeed, many anti-virus products also support ransomware detection and removal as a side-feature, which is however still based on the same techniques and heuristics applied in standard AV checks. More tailored tools like SurfRight's Hitman-Pro.Kickstart (7) are able to detect ransomware infections on a PC and eventually remove the threat found by looking for artifacts of known ransomware.

The pitfall of all existing approaches based on signature checking is their inability to detect new families or variants of malware: they are easy to evade, need constant and timely updates, which may not be effective to avoid infection of quickly spreading malware, and therefore are not future proof.

We want to build a tool able to detect ransomware based on its distinguishing features, capable of adapting to new variants and preventing infections before they spread by analysing unknown samples to recognise signs of ransomware behaviours. Additionally, given the sheer number of scans that users require per day (4), our second goal is to make our tool scalable, by efficiently excluding applications that are most likely goodware from in-depth analysis.

1.4 Our proposal

After manually analyzing and reverse engineering a consistent number of known ransomware families and their variants, we concluded that the key-features that distinguish ransomware from other malware (and from goodware of course) are essentially three: they *threaten* users, *lock* their devices and *encrypt* their data. Notice that the last two features are embedded in the source code of the application, allowing us to look for signs of this suspicious behaviour using program analysis techniques. In particular, HELDROID employs static taint analysis and lightweight symbolic execution to find code paths that indicate device-locking activity or attempts to encrypt files on external media. Detecting "threatening" apps is trickier but constitute one of the main novel contributions of our approach. We observed that displaying false accusations against users is the most compelling aspect of any ransomware, an action that by its nature requires the app to display some kind of ominous *text* similarly to phishing or social-engineering attacks. HELDROID uses a tailored classifier, that we devised, based on natural language processing, in order to determine the presence of accusation, threatening, or similar sentiments in the text.

In order to make the system more scalable, we added a pre-filtering step that quickly discards applications exhibiting goodware characteristics (or do not exhibit well-known malware characteristics) with high confidence. This step is implemented through a classifier trained on a large and comprehensive set of static, lightweight attributes extracted from a labeled dataset containing more than 185,000 samples of both kinds. We designed and tuned this filter such that it favors precision and speed at the cost of a slight loss in accuracy: we do not want to misclassify malicious applications as goodware, whereas letting some goodware through does not cause any harm because, in the worst case, the subsequent analysis does not qualify them as ransomware nor scareware.

1.5 Our contributions

Experimental results obtained on hundreds of thousands of samples (containing both goodware and generic malware) showed that the filter exhibits a prediction accuracy of 92.7% with 99.6% precision and a running time of a couple of seconds on server-class hardware (scaling linearly). Among unfiltered samples, HELDROID was able to detect all true ransomware, setting it apart from corner-case samples (e.g., perfectly legitimate applications that exhibited some, but not all, the typical characteristics). Remarkably, HELDROID recognized ransomware that it has never seen during training, effectively showing the robustness of its future-proof detection logic. In summary, our contribution is twofold:

- We propose a novel – and, to the best of our knowledge, the first of its kind – approach for ransomware detection based on natural language processing, static source analysis and symbolic execution, implemented both through existing libraries and custom-written code, as described in Section 5.1.
- We couple our detection tool with a lightweight and efficient filter able to reduce the computational resources needed to perform our analysis, which can also be used in any Android malware scanning pipeline. The novelty in this contribution resides in the choice and design of novel features used as classification attributes, as described in Section 3.1.

Additionally, we disclose our tools and datasets to the public, providing a JSON-based API for ransomware analysis that anyone can access at `http://ransom.mobi`.

CHAPTER 2

BACKGROUND AND MOTIVATION

Despite the raging success of ransomware campaigns traces back only to 2013/2014, there have been several sporadic cases of ransomware infections dating back to 2000, first on Windows (8) (9) (10) (11) and later for Android (12) (13) (14) (15) (16) (17), as described in Table I. Interestingly, the idea of using cryptography as a mean of extorting money was first theorized by Young and Yung in 1996 (18), introducing the concept of a *cryptovirus* as a *"trojan horse 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."*

2.1 Ransomware features

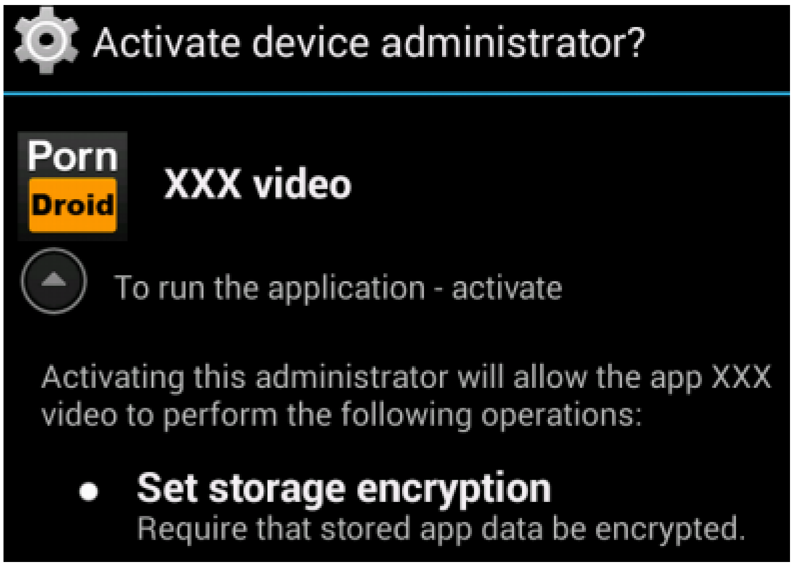
In order to understand precisely how to construct a valid and consistent approach for ransomware detection, we started by reverse engineering an handful of samples coming from the Contagio Minidump (22) public database. Starting from the original application package, we extracted all files contained therein using an Android specific unpacking tool (Apktool, more details in Section 5.1), then we disassembled the main executable file into plain java code through `dex2jar` (23) and `luyten` (24). We inspected the application manifest, its textual assets and the source code extensively. In this first step, we noticed that all analyzed families exhibit all or some of the following behaviours.

2.1.1 Device locking

"Device locking" consists in denying users the ability of accessing their device and apps, rendering the phone effectively non-functional. There are several ways in which malware authors may accomplish this task and we restricted these options to three main design choices, which can also be used on top of one another. The first approach is flamboyant, since it requires to directly ask users for their permission to acquire administration privileges, which is typically flanked by some poorly made-up excuse that users can still fall victim of. After obtaining such privileges, an app can shut the screen down whenever it pleases. This first option is the one chosen by famous ransomware PornDroid, as shown in Figure 2. The second technique consists in over-imposing an unkillable full-screen alert dialog that can be dismissed only after entering a proper payment code. Similarly, the last option traps key-press events on *Home* and *Back* buttons so that the victim can no longer switch away from the current activity (i.e. the ransomware activity).

2.1.2 Data encryption

There are few samples that take this aspect seriously. Most of them don't actually contain encryption routines even if they *threaten* to encrypt user's data; alternatively, they may contain encryption routines that are however never called (most likely copy-pasted from other malware samples). To the best of our knowledge, the only ransomware family that makes consistent use of encryption is SimpLocker (or SimpleLocker). In its first versions, the encryption key was fixed and hard-coded in the source code, whereas its most recent revision (17) generates


Activate device administrator?

XXX video

To run the application - activate

Activating this administrator will allow the app XXX video to perform the following operations:

- Set storage encryption
  Require that stored app data be encrypted.

Figure 2: PornDroid ransomware asking for administration privileges

per-device keys. Nevertheless, many samples of this family never actually call the decryption routine, leaving data permanently unavailable even after payment.

2.1.3 Threatening text

Regardless of the family it belongs to, every ransomware or scareware sample exhibits a common invariant: it displays some sort of text to the user, which is used first and foremost to accuse him and then to specify payment information. English-localized ransomware typically ask victims for MoneyPak (25) codes that are sent to a remote C&C server for validation. Samples localized in different languages (like Russian) also make use of cryptocurrency or credit card transactions.

2.2 Goals and challenges

Considering the relevant threat posed by ransomware and an enourmous attack surface comprising billions of connected devices, and given the limitations of currently available countermeaures, our final goal consists in implementing an adaptable and efficient ransomware detection tool. In particular, we want to overcome the limitations of signature-based approaches and identify a set of distinguishing features to recognize both known and novel ransomware. Additionally, we want our system to be scalable and usable in real-world settings.

TABLE I: TIMELINE OF KNOWN RANSOMWARE AND SCAREWARE FAMILIES

First Seen	Name	Platform	Extort	Encrypt	Lock	Threaten	Target and notes
1996	(18)	Macintosh	-	RSA	No	No	Research prototype
Late 2000	One_Half (8)	Windows	No	No	No	Yes	Portion of hard drive
Late 2004	GpCode (9)	Windows	100 USD	AES, RSA	No	No	Media and documents
Mid 2012	Troj/Ransom (10)	Windows	10.99 USD	No	Yes	Yes	Media and documents
Mid 2012	Reveton (19)	Windows	100 EUR	No	Yes	Yes	Police-themed screen lock
Late 2013	CryptoLocker (11)	Windows	400 USD	AES, RSA	No	No	Media and documents
Mid 2014	CryptoWall (20)	Windows	500–1,000 USD	RSA	Yes*	No	Media and documents
Dec 2014	Reveton (21)	Windows	100 EUR	No	Yes	Yes	Police-themed screen lock; steals cryptocurrency wallet passwords
May 2014	Koler (Reveton) (12)	Android	300 USD	No	Yes	Yes	Police-themed screen lock; localized in 30 countries; spreads via SMS
Jun 2014	Simplocker (13)	Android	260 UAH	Yes	Yes	Yes	All files on SD card; uses hard-coded, non-unique key
Jun 2014	Svpeng (14)	Android	200 USD	No	Yes	Yes	Police-themed screen lock
Aug 2014	ScarePackage (15)	Android	(hundreds) USD	No	Yes	Yes	Can take pictures and scan the device for banking apps or financial details
Early 2015	CBT-Locker (16)	Windows	(variable) BTC	EC	No	No	Media and documents; spreads via localized spam email.
Early 2015	New Simplocker (17)	Android	200 USD	Yes	Yes	Yes	Per-device keys; advanced

We exclude minor variants an aliases. Missing details, which are denoted with a '-', are unknown to the best of our knowledge. * = prevents system recovery

CHAPTER 3

GENERAL APPROACH

This section describes how Heldroid works, specifying the input, output and the logical decision flow of its modules. Figure 3 highlights the overall structure of the system and graphically illustrates the relations and I/O between components. Heldroid starts with a pre-filtering phase to decide whether an input sample is worth analyzing (i.e. whether it is not goodware). If so, the APK is passed through three different detectors corresponding to the three main ransomware features (described in Section 2.1) to compute the final verdict.

3.1 Filtering

In order to quickly decide whether an application is suspicious, we adopt a supervised machine-learning approach. Note that such an approach is not novel and has been widely used in practice by a consisten number of research projects (26) (2) (27) (28) (29) (30) (31). All these works differ in how they treat the problem of detection by choosing a diverse feature set or learning algorithm; however, all of them share the common goal of identifying malware, either in a general or specific way. Although we use features proposed in literature[1] alongside novel ones, our vision of the problem shifts in the opposite direction. Instead of aiming at detecting malware, we want to find those applications that are very likely to be goodware,

[1]Notice that there is a core set of features which is really hard to leave out when performing static analysis. For example, Android permissions explicitly summarize what an application can do and are thus a very clear indicator of its general behaviours. They are also extremely easy to retrieve from configuration files and constitute a very basic starting point of any analysis.

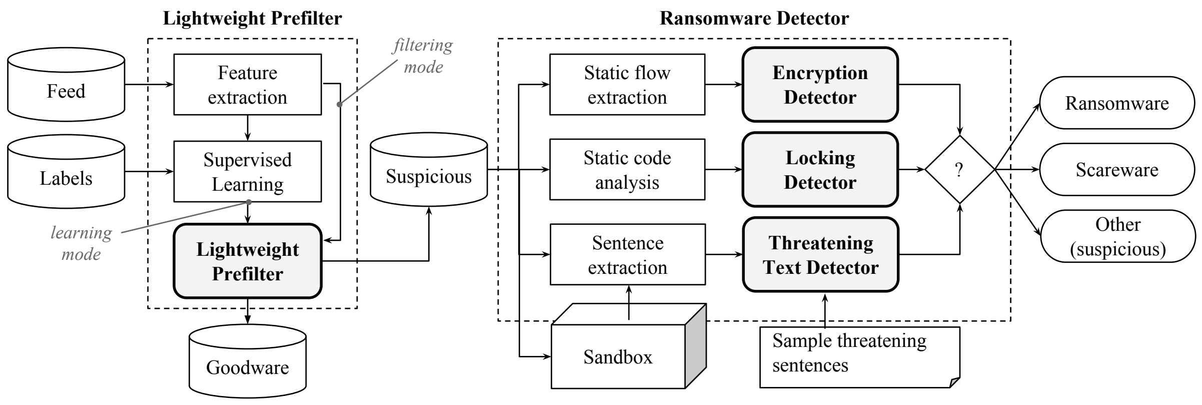


Figure 3: Diagram of the overall system at a conceptual level

i.e. completely benign. Translated in statistics terms, while traditional approaches maximize accuracy, we maximize precision on goodware. HELDROID accomplishes this task by assigning asymmetric weights to misclassification errors: for us, classifying a malicious sample as goodware is abysmally worse than classying a benign sample as malware; the latter error only incurs in resource consumption without harming anyone, while the former can generate security issues. This of course comes with a small price on accuracy.

Moreover, since we want this phase to be as fast as possible, we focus on features that can be statically extracted from an APK without resorting to heavy and time-consuming dynamic analysis. The new types of features that we propose include behavioural code, file heuristics, obfuscation and package formats (more details in Section 4.1.1).

The output of this step is thus binary: either *goodware* or *suspicious*.

3.2 Ransomware detection

Once an application has been labeled as suspicious by the filter, it passes through three other modules, which can be executed in parallel. Based on their output, HELDROID can formulate a final prediction on the sample. If the only triggered module is the **Threatening Text Detector**, the final label will simply be *scareware*: this means that the application displays a generic form of threatening or blackmailing to the user through some textual resources, trying to trick him into paying a sum of money by means of social engineering. If the **Encryption Detector** or **Locking Detector** are also triggered, the label will be either *ransomware* or *crypto-ransomware*: this means that the application not only threatens the victim but actively takes control of the device and/or data.

We reduce the problem of detecting "threatening text" to a text-classification problem, while we solve the other two with deterministic decision criteria based on static flow analysis and symbolic execution.

3.2.1 Threatening text detection

An application is able to display text by loading strings from a number of media such as APK resources or remote web pages. From a technical point of view, it could also resort to embedding text into images, however we focus exclusively on the difficult problem of deciding whether a text is threatening or not. Details on this approach's limitations and incremental improvements can be found in Chapter 7.

HELDROID first analyzes strings extracted from the application resources (assets and configuration files). If they contain no threats, it proceeds further and analyzes strings obtained with a dynamic analysis: in particular, it focuses on plaintext web traffic sniffed while executing the application in a emulated sandboxed environment.

To determinate whether a string resource contains threatening text, we employ a natural language processing supervised classifier trained on a dataset of sentences manually extracted from known ransomware samples. Sentences are labeled and divided into context categories (*threat*, *law*, *copyright*, *porn* and *money*) used both to direct analysis heuristics and to organize them coherently for better readability. We face the problem of localization transparently: HELDROID processes strings in any natural language it has been trained on, and can be easily be extended to support other languages in a matter of minutes, as discussed in Section 6.4.

Conceptually, the outcome of this phase is binary, either *found* or *not found*, but since the task is based on heuristics and probabilities, we provide several numerical indicators of how well the text actually matches our dataset (see Section 4.2 for more details).

3.2.2 Encryption detection

Unsolicited file encryption opterations are usually carried out by reading the external storage[1] media, sequentially looping over all available files, encrypting them and thereby deleting the original copy. We are thus interested in finding execution flows that originate from read operations in the external storage and terminate into encryption routines. To this end, we exploit a static taint analysis framework (described in Section 5.1) to track execution paths from

[1]There are two types of storage in Android: internal and external. The former is a per-application directory accessibile only and exclusively by the application code (not even system can read it). The latter is a shared space accessibile by every process, generally mounted from an SD card

`Environment.getExternalStorageDirectory()` to `Cipher.doFinal()` or to `CipherOutputStream` constructor.

The output of this module is a boolean decision telling whether there exists any execution flow from sources to sinks as explained above.

3.2.3 Locking detection

As described in Section 2.1.1, there are several different techniques to enact device locking. Without repeating them, we detect locking operations by static code analysis. More precisely, we check whether certain methods return specific values or whether the code contains certain API calls with specific parameters values.

Similarly to other modules, the output of this module is a binary decision representing the presence of any locking strategy.

CHAPTER 4

APPROACH DETAILS

This section describes the details of our approach at a conceptual level, without providing implementation-specific details, which are covered in Chapter 5.

4.1 Filtering

We implement the filter by means of supervised machine learning, training a classifier model on a large dataset of samples. Details on the dataset numerosity, distribution, source and collection mode are explained in Section 6.1. Here we present the fundamental concepts at the root of the approach: choice of features, feature selection algorithm and choice of the learning algorithm.

4.1.1 Feature set

The choice of a good feature set is the starting point of any data mining approach. In this case, we want our features to be representative of an application's behaviour and sufficient to learn a discriminant model able to discern goodware. Some of them were inspired by previous work (32) (2), but we added several features of our own: statically mined behavioural patterns, package-name heuristics, file types and counts, obfuscation, domain name well-formed-ness and availability and system utility executions through `Runtime.exec`. Notice that only a smaller subset is actually used in the final model since we applied a feature selection algorithm. In the following paragraphs, we describe the chosen features by category, while the complete list of all features (before selection) is reported in Appendix B.

Permissions (binary). Android applications are executed in isolation: they are sandboxed in distinct Linux processes with distinct UIDs and an additional layer of security that regulates inter-process communication. Permissions are bestowed by the user upon the application at installation time (33) (34). Since users often grant such permissions without really understanding what they imply (35), malware can easily exploit them to escape the sandbox. As shown by previous work (30), permissions are in general distributed differently in goodware and malware and they are thus useful in discriminating samples.

We extract permission features from the `AndroidManifest.xml` configuration file and encode them as boolean attributes, depending on whether they are present (declared by the application) or not. In order to make results more readable, we further divided them into four categories (*dangerous*, *harmless*, *data* and *communication*).

Intent filters (binary). *Intents* are message objects used for intra-application communication in Android. They usually contain an action and some data: the action denotes which task is being invoked, while data specify further parameters details. *Intent filters* are declared in the application manifest and used to hook up system and application-specific events. Whenever an event is triggered (i.e. an intent is launched), Android looks for every application that declares the corresponding intent filter and notifies it. Notification happens sequentially according to a priority level that can be specified by the programmer: this mechanism can be exploited to intercept and manipulate any relevant event, without the rest of the system noticing. By declaring proper intent filters, a program may know, for example, when a new app is being installed, when a media is mounted or when the phone is rebooted. The latter is used very often to make malware persist across reboots.

Similarly to permissions, intent filters can be easily extracted from the `AndroidManifest.xml` configuration file and encoded as boolean attributes. We limit their number by focusing on about twenty intent filters that we deemed potentially dangerous.

API calls (binary). If permissions and intent filters are useful in determining what the high-level behaviour of an application may be, API calls are even more informative on its actual low-level inner workings. By disassembling a sample and inspecting its code, we know precisely which API methods it invokes. However, Android exposes a large number of classes, each one with myriads of methods: if we were to take into account every possible method invocation, we could easily end up with more features than records. For this reason, we adopted a heuristic approach to skim the number of interesting and potentially useful API methods down to a few dozens. More precisely, we drew 8,000 samples at random from the **AndRadar** dataset[1]; then we statically extracted the list of all invoked API methods per sample, aggregating everything into a map that counts the number of occurrences of each method. Finally, we computed a score s as follows:

$$s(f) = \begin{cases} -\frac{p(f|M)}{p(f|G)} & \text{if } p(f|M) > p(f|G) \\ \frac{p(f|G)}{p(f|M)} & \text{otherwise} \end{cases}$$

[1]Dataset composed by real-world applications download from several different independent markets, totalling more than 170,000 samples. More details in Section 6.1.

where f represents a feature (i.e. an API call), while $p(f|M)$ and $p(f|G)$ are the conditioned probabilities[1] of finding f in malware and in goodware, respectively. This score is similar to the *probability factor* described in (29, p.6)[2], except that s can be negative and is always greater than or equal to 1 in absolute value.

By ranking all invoked API calls based on the score s and selecting those that exhibit an abnormally large absolute value (empirically, in the range $10^1 - 10^2$), we obtained a list of several dozens features, like `TelephonyManager. getSubscriberId`, `System.loadLibrary`, `Parcel.marshall`, etc.

Lightweight behavioural patterns (binary). One of the flaws of static features extracted so far is that they are "standalone", i.e. they are oblivious of the external environment. For example, if the `System.loadLibrary` is invoked, its associated attribute is *true*, however we do not know where or in response to what event it was invoked. We wanted to overcome such limitation by adding some other more complex features that are representative of concrete behaviours that frequently appear in malware. To this end, we perform a lightweight (finite-depth) reachability analysis on the application source code to determine whether it

- sends SMS as startup;
- reads phone data at startup;
- sends data when receiving an SMS;

[1] Actually, frequentist probability estimations, since we have a finite amount of data.

[2] With partitions R and M representing goodware and malware respectively.

- sends SMS to short numbers (used in premium services);
- calls any built-in system utility (i.e. `su`, `root`, `chmod`, etc.).

Even if these features alone are of course insufficient to indicate suspicious behaviours, when used together with the others they provide interesting feedbacks. Again, they are encoded as boolean attributes.

Heuristics (binary). This subset contains miscellaneus and heuristic features, for example: whether the package name is composed by only one part, whether the domain of the main package is valid[1], presence of hard-coded URLs, presence of URLs whose domain does not match the main package, whether Proguard has been used to obfuscate the source code[2], and so on. We designed this diverse but simple subset of features after manually inspecting several malicious and benign samples.

Assets and source code stats (numerical). Finally, in order to get a representation of how the application is packaged, we include information as the number of files in an APK, number of permissions, activities and services, size of the APK, average class size, total number of packages and the number of classes in the main package.

[1]We consider a domain to be valid if it is listed among the ICANN top-level domains, as reported in (36).

[2]Proguard employs a specific naming convention that always assigns to each entity a progressive lowercase alphabet letter, starting from *a*, and using two or more letters when the naming options are terminated (e.g. *aa*, *ab*, etc.). Because of Java scope nesting, most classes and methods are simply renamed to *a*.

4.1.2 Feature selection

By considering all the above features, we ended up with more than 220 attributes, which are too many with respect to the size of the training dataset. We thus selected the first 120 most significant attributes throught gain-ratio ranking (37, p.301). Gain ratio is proportional to the information gain of an attribute, i.e. the difference between the dataset entropy and the total entropy of partitioned dataset according to the attribute values. It is also inversely proportional to the intrinsic value of the attribute, a measure of its "sparseness" (i.e. how many values does it allow?). The choice of gain ratio as information measure was driven by the usage of decision trees and random forests as suitable classifier models.

4.1.3 Classifier algorithm 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 tradeoff between learning time, accuracy and precision is a compound classifier that performs majority voting (38) 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 i.e. non-goodware misclassified as goodware weigh more than false negatives (39). This is crucial for maximizing prediction precision on goodware. By empirical tests, we found that the cost to assign to misclassifications of such type in order to obtain reasonably high accuracy and very high precision ranges between 16 and 20 (1 being the base misclassification cost).

4.2 Threatening text detection

This module is the core of the entire system and essentially describes how HELDROID maps conventional plain text strings to a mathematical representation suitable for classification. The transformation process can be seen as a function that maps each string to a set of boolean vectors, as follows:

- **Language detection.** In order to determine which components to use in the following steps, we need to know the natural language used in the given string. A simple frequency analysis is more than enough to discern with very high accuracy any supported language. More details on the specific implementation are given in Section 5.2.
- **Segmentation.** The string is split into *sentences* according to a language-specific sgmentation model.
- **Stemming.** Each sentence is passed through a filter that removes all *stop-words*. Stop-words are common "noise" words used to make a sentence syntactically correct but without any relevant meaning for the sentence semantic. For English, some examples are "the", "more", "he", "are", etc. HELDROID then processes all remaining words to extract their *stems*, i.e. "the part of the word that is common to all its inflected variants" (40, p.248). Repeated stems are ignored.
- **Vector space.** Even if from a computational point of view the process of transformation stops at the last step, from a mathematical perspective, the sets of stems can be seen

as vectors of an highly-dimensional boolean space: the vector's components are 1 iif the associated stem is present in the sentence that the vector represents, 0 otherwise. We call these vectors **stem vectors**[1]. The boolean space dimensionality should be as large as the total number of different stems that we can possibly find in any existing document.

To make the process clearer, here is an example with input string "This device has been locked for safety reasons. All actions performed are fixed."

1. **Language detection.** English
2. **Segmentation.** {"This device has been locked for safety reasons", "All actions performed are fixed"}
3. **Stop-words removal.** {"device locked safety reasons", "action performed fixed"}
4. **Stemming.** {{"device", "lock", "safe", "reason"}, {"action", "perform", "fix"}}
5. **Vector space.** $\{(1, 1, 1, 1, 0, 0, 0), (0, 0, 0, 0, 1, 1, 1)\}$[2]

During training, HELDROID transforms all manually collected sentences from known ransomware into stem vectors and stores them in a training set T (we use different training sets for each supported language). At run-time, the system processes strings extracted from the application data (see Section 3.2.1) in the same way and compares them to those in the training set, yielding a final *score*; this score is then analyzed in order to make the final binary *decision*.

[1] In the following pagraphs, *sentence* and *stem vector* will be considered synonims

[2] This representation assumes the universe of stems to be composed only be those listed in step 4.

Scoring. Given a stem vector x representing a sentence extracted from application resources or web traffic, its score is defined as:

$$score(x) = \max_{t \in T} s(x, t)$$

where $s(x, t) \in [0, 1]$ is the cosine similarity between the two vectors x and t:

$$s(x, t) = \frac{\sum_{i=1}^{N} x_i \cdot t_i}{\sqrt{\sum_{i=1}^{N} {x_i}^2} \cdot \sqrt{\sum_{i=1}^{N} {t_i}^2}}$$

This means that the score of a sentence is a measure of how similar that sentence is with respect to those contained in the training set. This scoring algorithm acually works in a way similar to a k-Nearest Neighbors classifier (with k=1), using cosine similarity as distance measure (41, p.88). Differently from k-NN, however, we are interested in the best-match distance measure rather than in its label.

From a computational viewpoint, since x and t are *boolean* vector and they are actually stored as *set of strings* (i.e. a `Set<String>` in Java), we can reduce s to:

$$s(x, t) = \frac{|x \cap t|}{\sqrt{|x|} \cdot \sqrt{|t|}}$$

which takes $O(min(|x|, |t|))$ time to compute.

Now, recall that each sentence in the training dataset is assigned to one specific category among *threat*, *law*, *porn*, *copyright* and *money*. The first four categories represent various forms of *accusations*, i.e. different types of indictment that ransomware can adduce against a victim; the last category, *money*, contains sentences related to payment methods. When HELDROID evaluates the score of a sentence x, it also assigns x a category, which is the same as the category of its best-match in training set T. At this point, we have a list containing all sentences extracted from plaintext strings, each one associated with a category and a numeric score. From this list, HELDROID considers the best-matching sentence $\hat{c}_a$ from *accusation* categories and the best-matching sentence $\hat{c}_m$ from *money* category. The simplest way to make a decision with this information is to check whether $score(\hat{c}_a)$ and $score(\hat{c}_m)$ both exceed a certain threshold value. However, adopting this approach would account for only a pair of sentences in the whole application text, which may be way larger. For this reason, we decided to apply an *increment function* to such results.

Score increment. Let's assume that $m_a = score(\hat{c}_a)$; then the augmented value of m_a will be[1]:

$$\hat{m_a} = m_a + (1 - m_a) \cdot \left(1 - e^{-\alpha \sum_{i=1}^{n} \max\{score(c_i) - t(c_i), 0\}} \right)$$

where n is the number of sentences in all the accusation categories, c_i is the i-th sentence and t is an *adaptative threshold function.* For the moment, ignore the formulation of the exponential and consider only the general shape of the formula. We wanted to take into account *all* relevant sentences in the mined text rather than only the best-match. From a mathematical point of

[1]The same can be applied to $m_m = score(\hat{c}_m)$

view, since the score is bounded between 0 and 1, the maximum value that we can add to m_a without exceeding the score bounds is $(1 - m_a)$. The exponential function $1 - e^{-x}$ converges to 1 as $x \to +\infty$, thus we used it as a tool to express how "much" of $(1 - m_a)$ we could exploit. Ideally, if all sentences are relevant, $\hat{m}_a$ will be very close to 1 (or exactly 1 since floating point precision is finite). The following paragraph describes when a sentence is considered "relevant" and how t is computed.

Consider the argument of the summation in the exponential. The function t accepts a sentence and returns a value in $[0, 1]$ representing its specific *threshold*, i.e. the minimum score value that must be exceeded in order for the sentence to be considered *relevant*. The idea behind t 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. More formally:

$$t(c) = \tau_{max} - \gamma(c) \cdot (\tau_{max} - \tau_{min})$$

$$\gamma(c) = max(0, min(1, \frac{\sum_{c_i \in c} c_i - \sigma_{min}}{\sigma_{max} - \sigma_{min}}))$$

The summation $\sum c_i$ is just a formal way to express the number of stems in sentence c, which would be the number of 1s in the boolean stem vector associated to 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}.

Finally, the parameter α controls the steepness of the exponential function, which indirectly influences how much a single relevant sentence (i.e., a sentence whose score is greater than the threshold) affects the overall category score. It can be tuned according to different needs, but in all our tests its value was simply 1.

During our training, we set these parameters such that the classifier distinguishes known ransomware from malware or goodware. More precisely, we obtained our results with: $\tau_{min} = 0.35$, $\tau_{max} = 0.63$, $\sigma_{min} = 3$, and $\sigma_{max} = 6$.

Decision. In order to translate the obtained values into a binary decision, HELDROID simply checks $\hat{m}_a$ and $\hat{m}_m$ against a fixed threshold: if *both* exceed the threshold, then the sample triggers the threatening text detection.

4.3 Encryption detection

As already mentioned in Section 3.2.2, encryption is detected if a flow between `Environment.getExternalStorageDirectory()` and `Cipher.doFinal()` or `CipherOutputStream()` exists. No further explaination is needed since flow detection is a known technique and we use a third-party library for this task. However, for the sake of completeness, we'll illustrate an example of detected flow sourced from a real-world application[1] in Listing 4.1 and Listing 4.2. Comments should be informative enough to understand what the code is doing and how the execution flow

[1]MD5: `c83242bfd0e098d9d03c381aee1b4788`

leads from one method to another: notice that, since the code is taken from a real sample, it is obfuscated and thus not very readable by human beings.

```
.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 and then
    # passed to another method
    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
    # [...]

    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"
    # [...]

    # This code is inside a loop. It invokes the encryption routine "a"
    # (from Listing 4.2) on the file v0, which is then deleted after being
    # encrypted with a different name
    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
```

Listing 4.1: Source of the encryption flow

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

.method public final a(Ljava/lang/String;Ljava/lang/String;)V
    .locals 6
    # Initializes a FileInputStream to read file contents. The 2nd
    # argument of this method is the string path of such file (p1)
    new-instance v0, Ljava/io/FileInputStream;
    invoke-direct {v0, p1}, Ljava/io/FileInputStream;-><init>(Ljava/lang/String;)V
    new-instance v1, Ljava/io/FileOutputStream;

    # Creates a new FileOutputStream for writing
    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;

    # Uses the first argument (a string encryption key) to create
    # a SecretKeySpec object used for encryption
    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;

    # Then creates a Cipher object with algorithm and key information.
    # This is used to actually perform encryption
    invoke-virtual {v2, v3, v4, v5},
        Ljavax/crypto/Cipher;-><init>
        (ILjava/security/Key;Ljava/security/spec/AlgorithmParameterSpec;)V

    # A CipherOutputStream is initialized and used to encrypt the file passed
    # as argument by the last method in Listing 4.1, using the Cipher object
    # as encryption blackbox and the FileOutputStream as write support
    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
```

Listing 4.2: Sink of the encryption flow

4.4 Locking detection

The three supported locking strategies described in Section 3.2.3 can be enacted in the following ways:

Administration lock. Require administration privileges and then call `DevicePolicyManager.lockNow()`, which forces the device to act as if the lock screen timeout expired. In order to require administration privileges, an application must expose the `BIND_DEVICE_ADMIN` permission and subsequently launch a new intent with `android.app.action.ADD_DEVICE_ADMIN` as action and a java `Class` object as content: such object will implement the new administration policy. Additionally, the `AndroidManifest.xml` must contain a `meta-data` tag specifying an `android:resource` attribute pointing to a configuration file containing a list of actions available to the new administrator. Actions are declared as xml tags inside such configuration file: in order to be able to call `lockNow`, the `force-lock` tag must be present.

All these conditions can be verified by simply inspecting configuration files and scanning the source code sequentially.

Navigation inhibition. Fill the screen with an activity that disables navigation by inhibiting the *Back* and *Home* buttons. This can be achieved by overriding the `onKeyUp` and `onKeyDown` methods of any `Activity` child class and returning `true` when the key pressed correspond to *Back* or *Home*. Returning `true` in such methods means that the key event was properly handled and that Android should not forward that event any further to other components in the lower parts of the view stack. An alternative solution is to draw over software-implemented navigation buttons, but this approach only works for recent device models and require the `SYSTEM_ALERT_WINDOW` permission.

Detecting this behaviour is the trickiest task among the three. We accomplish it by parsing the Smali source code, finding declarations of `onKeyUp` or `onKeyDown` and then emulating every single statement in their body to verify the existence of an execution path in which the return value is `true`. We take particular care in analyzing `if` statements that compare the value of the key with a consant integer value, since we know the integer codes associated to *Back* (`0x04`) and *Home* (`0x03`). A real-world [1] example of this behaviour is shown in Listing 4.3.

Immortal dialog. Open an immortal (uncloseable) dialog window. When writing a class derived from `AlertDialog`, the programmer can call `setCancelable(false)` to avoid it to be canceled by the user. Additionally, he can call `Window.setFlags` passing as a paramenter any flag including `FLAG_SHOW_WHEN_LOCKED` (`0x00080000`) to allow the dialog to be visible also when the screen is locked (thus showing it on top of the lock screen).

Finding such invocations can be achieved by program slicing; this is made easier by the fact that parameters must be constant values or "literals", which are declared in bytecode by a proper *const* directive right before the method invocation. Anyway, our code is smart enough to also consider values declared in other parts of the class (even if this is not strictly necessary).

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

    # p1 contains an integer with the key code associated to the pressed key.
    # The constant 4 indicates the Back button. In this case, if the back
    # button is pressed, it is used to go back in a WebView instead of the
    # Activity stack
```

[1]MD5: `b31ce7e8e63fb9eb78b8ac934ad5a2ec`

```
    const/4 v0, 0x4
    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;

    invoke-virtual {v0}, Lcom/android/x5a807058/ae;->c()Z
    :cond_0

    # Otherwise, the method returns true, which means the event was handled:
    # this event is not forwarded any further to other components. This
    # actually inhibits navigation.
    const/4 v0, 0x1 #true
    return v0
.end method
```

Listing 4.3: Inhibiting navigation through onKeyDown overriding

CHAPTER 5

IMPLEMENTATION DETAILS

This chapter illustrates the technical implementation details of HELDROID, used libraries, data formats and algorithm specifications. All machine-learning tasks (preprocessing included) are carried out through the Weka (42) data mining framework, version 3.7.

5.1 Static code analysis

APK unpacking. Android applications are packed into a standard compressed format, with extension `.apk`. The most famous and widespread tool that allows decompression and decoding of APKs is Apktool (43), which is also our preferred choice. In particular, we use version 2.0.0 RC3. This tool isn't perfect and may be unable to correctly process certain packages: in fact, crooks often try to build APKs in such a way as to cause extraction and disassembling tools to crash, despite being well-formed for the Dalvik virtual machine. Applications that cannot be unpacked are discarded in our experiments, even if they are included in the total number of samples when reporting datasets numerosity. We aknowledge this limitation, but correctly unpacking every single APK is not the focus of our work.

Disassembling. Android systems run applications on top of the Dalvik virtual machine: binaries are encoded in a platform-specific bytecode and wrapped into a single `.dex` file (Dalvik EXecutable). In order to perform source code analysis, it is often convenient to transform bytecode in an equivalent representation using a plaintext language with the same capabilities. To this end, the Android community of researchers and experts developed several solutions, among which we recall *Jasmin*, *Jimple* and *Smali*. A comparison of these languages is given in (44). We chose to work on the **Smali** intermediate language because Apktool directly supports bytecode disassembling in Smali. To our knowledge, there exists only one tool able to perform static code analysis on Smali code: SAAF (45) (46). However, we found the latter to be unstable while working in a multi-threaded environment and thus we wrote our simple slicing and flow-analysis routines, specifically synchronized and tailored to our needs. To keep the approach fast, we only supported the minimum number of statement types required for detection purposes. HELDROID accomplishes locking detection using this custom written code.

Encryption detection. For this task, we leverage FlowDroid (1), a context-, flow-, field-, object- and lifecycle-aware static taint analysis framework with high recall and precision. Despite being a very "young" project, being presented less than a year ago, FlowDroid showed its robustness in more than 50 research works. The only information we need to plug into the framework are sources and sinks specifications, which we provide in Appendix A.

5.2 Natural language processing

Threatening text detection is a core part of our analysis and it should be able to support multiple natural languages. The language identification task is performed by the Cybozu open-source library *langdetect* (47). For each supported language HELDROID needs three different components:

- **Segmenter.** This component processes an input text and extracts meaningful units of information, such as words, sentences or topics. In our case, we want to segment the text into sentences. For this task, we employ *OpenNLP* (48), a generic extensible multilanguage NLP library. OpenNLP is capable of creating language-specific *sentence models* by training on a user-provided dataset composed by manually pre-separated sentences in the target language. The model can later be used by a run-time Java object to perform actual segmentation. A sentence model for the English language is directly available from the OpenNLP website. In addition, we train a model for Russian manually: for this task, we used as reference texts a transcript of *XXVI Congress of the CPSU and Challenges of Social Psychology* (49) and a Wikipedia article about Law (50).

- **Stemmer.** This component procsses an input word and returns its stem. The OpenNLP trunk contains a Java implementation of Snowball (51), a domain-specific language for coding stemmers. In particular, the `SnowballStemmer` abstract class is base to every language-specific stemmer: the main repository contains about a dozen already implemented snowball stemmers.

- **Stop-word list.** It is very easy to find stop-word lists for nearly every language. Ours come from the *Stop-words Project* (52).

5.3 Sandboxing

If static analysis does not find any threatening text, HELDROID resorts to dynamic analysis. In this scenario, the target sample is loaded into an emulator and run for a variable amount of time, usually in the range of 4 to 6 minutes. Thanks to a network sniffer, all traffic sent and received by the application during emulation is saved into a dump file, which can later be scanned to look for malicious text, as mentioned in Section 3.2.1. We follow an approach similar to *TraceDroid* (53) to simulate user interaction, rebooting, incoming/outgoing SMS or calls, and so on. Notice that a network-dump is sufficient to start an analysis and thus other developers may resort to their emulation tool of choice, like Andrubis (31) (54). Additionally, our text analysis technique can be applied to any string extracted from the sample, especially those coming from a memory dump (see CopperDroid (55)). Even if we do not deal with this approach, it would be useful to cope with encryption limitations, as explained in Chapter 7.

CHAPTER 6

EXPERIMENTAL VALIDATION

This chapter describes the results of all experiments we performed to test HELDROID. Each experiment aims at showing a specific feature of the system, like precision, accuracy, running time and quality of predictions. We performed running time tests on server-grade hardware (Xeon X3440 2.53GHz and 16GB RAM).

6.1 Datasets and settings

We employed six different datasets; size, content and purpose are reported in Table II.

- The **AndRadar** (56) dataset contains apps automatically downloaded from various independent markets between February 2011 and October 2013. Crawled markets include Blackmart (57), Opera (58), Camangi (59), PandaApp (60), SlideMe (61) and GetJar (62). Samples from this dataset are labeled as *Benign* or *Malicious*, depending on whether VirusTotal analysis triggered at least *10%* of anti-virus softwares. Out of the total, 147,145 samples are labeled as Benign.

- The **AndroTotal** (63) dataset contains apps submitted to the omonimous analysis platform[1] between June 2014 and December 2014. It is labeled in the same way as **AndRadar**. Out of the total, 6,837 samples are labeled as Malicious.

[1]This service is also the source that we query when our system receives a scan request for a specific hash.

- The **MalGenome** (30) dataset is an academic-only repository of malware collected between August 2010 and October 2011.
- The **ContagioMinidump** (22) dataset comes from a community-driven project, constantly updated with new samples of malware (June 2011 – December 2014).
- The **Ransomware1** dataset contains only and exclusively verified ransomware. We assembled it by using the VirusTotal intelligence API looking for samples with at least **5** detections and containing one of these keywords: *ransomware*, *koler*, *locker*, *fbilocker*, *scarepackage*, *simplelocker* or *simplocker*. Additionally, we ensured that, for each sample, at least **5** detection labels agreed on the same ransomware/scareware family. All samples have been submitted to VirusTotal no later than December 2014.
- The **Ransomware2** dataset contains potential ransomware. We assembled it in the same way as **Ransomware1**, but restricting the research between December 2014 and January 2015. Moreover, we didn't make sure that labels agreed on detection of ransomware, therefore it may contain false positives (and it does, as commented in Section 6.4). We purposedly omitted the last check because this dataset will serve as a disjoint test set for the system.

6.2 Experiment 1: Filter precision

As mentioned in Section 4.1, filtering is accomplished through a machine-learning classifier. We trained it on the union of **AndRadar** and **AndroTotal** datasets, performing a standard 10-fold cross-validation to get rid of biased results. Table III shows that classification capabilities

TABLE II: SUMMARY OF DATASETS

Name	**Size**	**Labelling**	**Content**	**Use**
AndRadar	172,174	VT 10%+	M, G, "R"	T, E
AndroTotal	12,842	VT 10%+	M, G, "R"	T, E
MalGenome	1,260	Implicit	M	E
ContagioMinidump	242	Implicit	M, R, S	T, E
Ransomware1	207	VT 5+, Manual	R, S	T, E
Ransomware2	443	VT 5+	G, R, S	E

M = Generic Malware, G = Goodware, R = Ransomware
"R" = Possibly Ransomware, S = Scareware, T = training, E = evaluation.

of the pre-filtering phase are very encouraging, especially considering that the training dataset is not homogeneous, containing samples from different sources and timeframes. To the best of our knowledge, this is also the largest dataset used in mobile security literature to train a classifier for detection/ranking purposes.

Bear in mind that the filter aims at detecting goodware and thus to maximize detection precision on goodware: it should **not** be used to classify malware! Also notice that false-negatives (benign samples misclassified as malicious) are a negligible cost: in fact, in a normal scenario they would have to be analyzed anyway; our filter instead reduces their quantity vastly, and quickly, as shown in the next experiment.

TABLE III: PERFORMANCES OF SEVERAL FILTERING CLASSIFIERS

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

6.3 Experiment 2: Filter speed

Since training is performed offline, we are interesting in measuring[1] the speed of the actual filtering phase in the final system. This phase can be divided into three sequential steps: *unpacking*, *feature extraction* and *prediction*.

Undoubtely, feature extraction is the most crucial step and its contribution is of paramount importance when determining the overall performances of the filter. Rather than giving just an average time, since the training dataset is very heterogeneous, we decided to plot execution times against several different measures: total size of Smali classes, total count of Smali classes, total files count and APK size. For sake of completeness, we provide a plot of raw data in

[1]Graphs and data for this experiment were gathered by scanning samples from the **AndRadar** dataset.

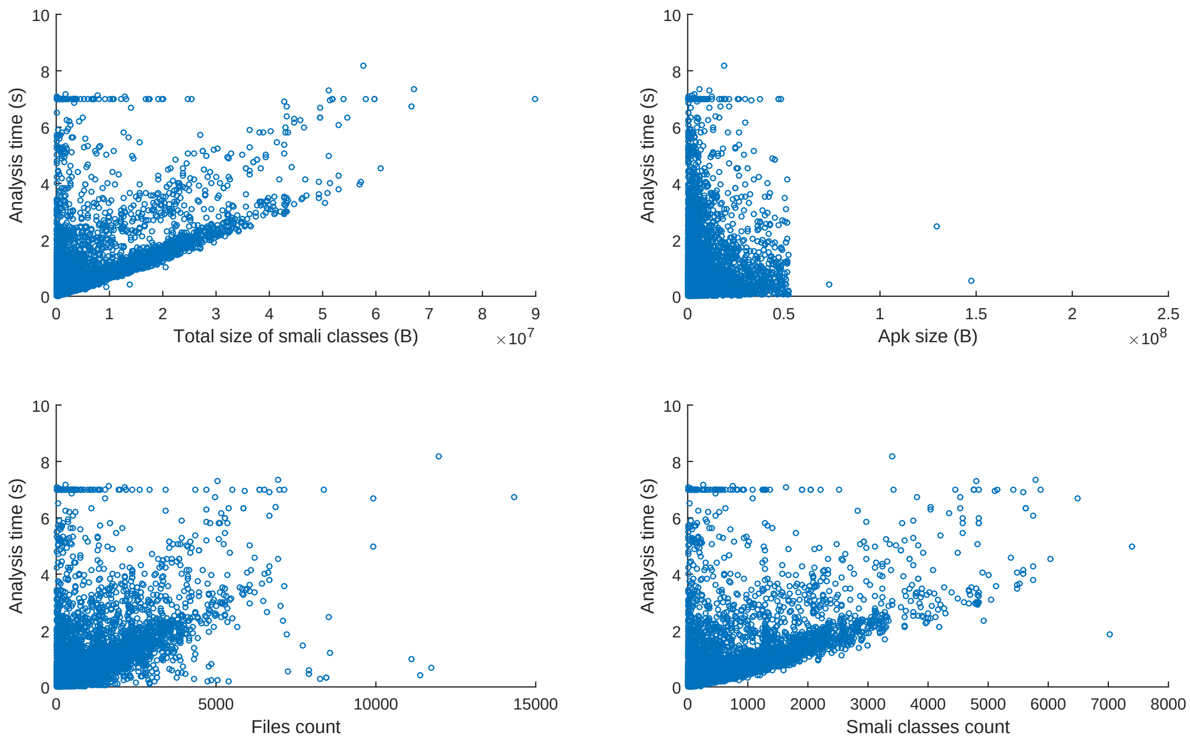


Figure 4: Filter execution time (raw data).

Figure 4[1]. As you can appreciate from the graph, the most suitable measure for further data processing is *total size of smali classes*, since the heaviest features are computed through static code analysis. A more refined visualization of the dependency between execution time and code size is shown in Figure 5. The worst-case average running time is less than 1.5 seconds.

[1]This graph is plotted using only a 10k random subsample of the **AndRadar** dataset, because the image format is vectorial and representing over 800,000 points would blow up the size of this document.

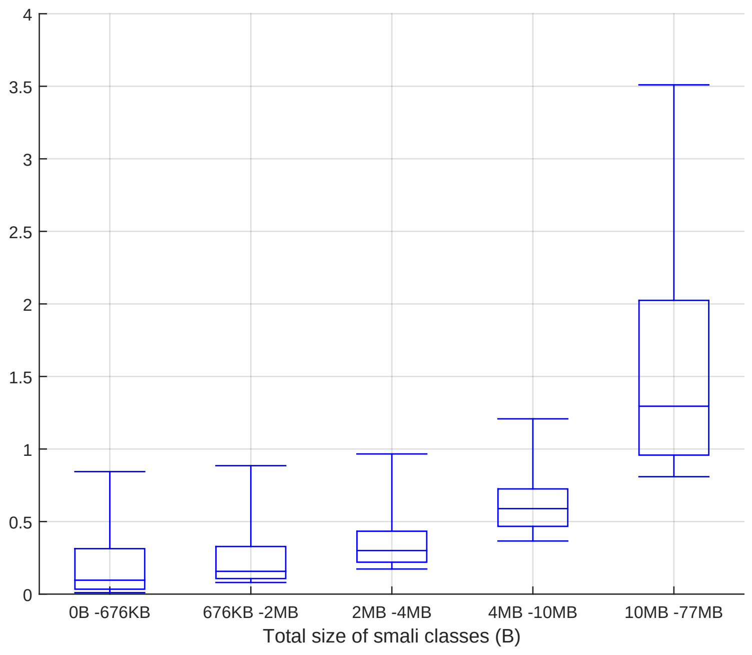


Figure 5: Filter execution time (boxplot).

As far as unpacking is concerned, notice that it isn't a task strictly related to our system. It is just a necessary overhead in order to prepare data for processing. Our tests showed that unpacking takes 2.484s on average, with a 1.922s median and 2.814s 3rd quartile.

Finally, the prediction step is so fast that its impact is negligible (in the order of milliseconds).

6.4 Experiment 3: Ransomware detection

We trained the *threatening text classifier* on a set of sentences[1] manually extracted from several ransomware samples in the **Ransomware1** dataset. As expected, after the training phase, HELDROID was able to correctly idenfity all relevant ransomware/scareware applications in the same training dataset: in particular, 194 were detected simply through static analysis and 13 needed further dynamic analysis. As you may notice, the total number of detected samples (207) is less than the size of the dataset (232): the rest accounts for corrupted packages, applications that couldn't be unpacked by Apktool or true ransomware whose C&C server was however offline, rendering the app unable to perform any action beside locking, which despite being correctly detected does not constitute a sufficient discriminant criterium.

After training, we ran HELDROID against the **Ransomware2** dataset, which is more recent and disjoint with respect to **Ransomware1**, in order to ascertain the system prediction quality and power. Recall that **Ransomware2** contains samples that trigger detection on at least *five* VirusTotal anti-virus softwares, among which only **one** needs to produce a ransomware-related label: thus, it may contain samples that are not ransomware. In fact, out of the 443 total apps, 375 were true positives (actual ransomware or scareware detected by our system) and 49 true negatives (strictly non-ransomware and non-scareware not detected by our system). Here is a breakdown of the last 49 samples:

- 14 BaDoink + 15 PornDroid clones (see below);

[1] 50 for English, 30 for Russian.

- 6 lock-screen applications to modify its look&feel;
- 14 benign, adware or other threats not directly related to ransomware (such as spyware).

PornDroid and *BaDoink* are notorious applications used as infection vectors for ransomware. Since they were not detected, we decided to install them on a real Nexus S device to observe their behaviour. However, the specific samples we analyzed were unable to perform any malicious action apart from locking the device screen. Additionally, a deeper analysis of captured network traffic revealed that remote endpoints of requests issued during execution were unreachable, thus preventing the sample from downloading and showing any webpage. As a matter of fact, an application that does not threaten the user and does not ask for any payment is, by definition, not a ransomware.

Out of the remaining 19 undetected samples:

- 11 were localized in a language that HELDROID wasn't trained on (French, German, Spanish and Chinese);
- 4 didn't contain any static or dynamically generated text, thus we concluded that they were disarmed, bogus or simply incorrectly flagged by 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. HELDROID was able to correctly detect other samples of this unknown family with fully functional C&Cs in **Ransomware1**.

As a proof of concept, we also implemented support for an additional language (Spanish) in order to check how it behaved on those first 11 samples. As explained in Section 5.2, to support a new language, HELDROID needs a segmenter model, a stemmer, a list of stop-words and naturally a list of threatening sentences. The whole process of looking for suitable texts, inspecting Spanish-localized ransomware samples, assembling the model and importing the Java stemmer took less than 30 minutes. In the end, we were able to correctly detect Spanish ransomware. A shortcoming of this strategy is that it cannot be applied to languages such as Chinese, where stemming and segmentation are extremely different from occidental cultures. This aspect is discussed more in depth in Chapter 7.

As a final note, consider that this experiment was challenging since an extensive and reliable ground-truth for ransomware does not exist. As we pointed out in previous paragraphs, analysis platforms like VirusTotal tend to exhibit a considerable number of false positives (apps labeled as ransomware even if they are not) due to commercial AVs incorrectly assigned labels. It is thus extremely difficult to automatically extract a large enough set of samples that contains exclusively ransomware: manual inspection is still necessary, but it ensures good quality datasets.

6.5 Experiment 4: Ransomware detection speed

Following the same approach used in Section 6.3, we measured execution times of each detector component: *threatening text detector*, *locking detector* and *encryption detector*. Results were produced by aggregating data from 5 runs of the system on a 1,000 items random subsample of the **AndRadar** dataset. As shown in Figure 6 and Figure 7, text classification is very fast

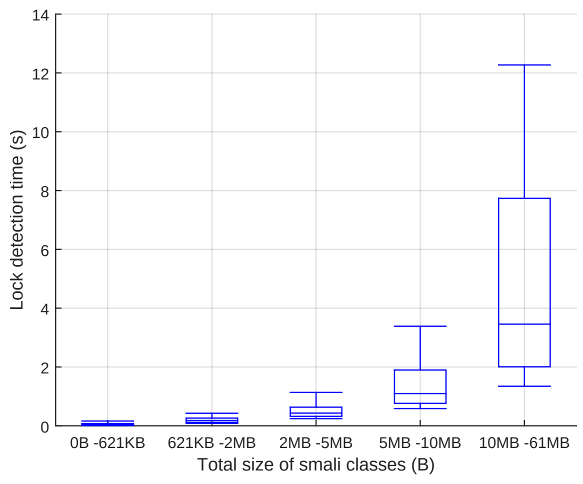


Figure 6: Locking detection time (boxplot)

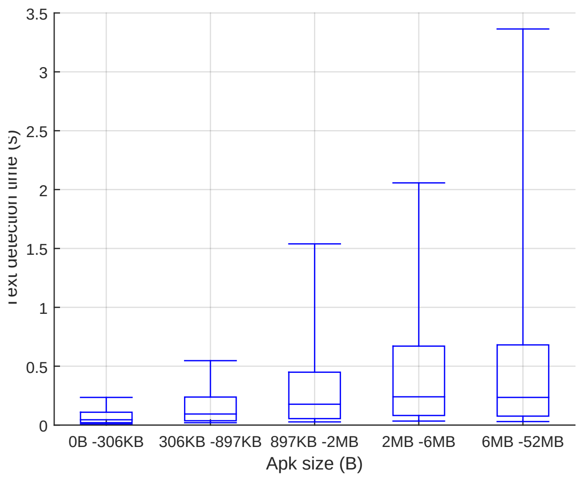


Figure 7: Text classification time (boxplot)

in all cases, while locking detection is the main bottleneck, still taking less than 4s on average for very large applications. We do not provide a plot for the encryption detection running time, because its impact is negligible (in the order of 10^{-2} seconds).

6.6 Experiment 5: Impact of the filter

Our last experiment aims at showing that the filter impact is significant on the overall system. To accomplish this, we prepared a test dataset composed by a 1,000 items random subsample of **AndRadar**, **MalGenome**, **ContagioMinidump** and the first 1,000 APKs submitted to VirusTotal between December 2014 and January 2015. Then we measured the *total execution time* of HELDROID with (D+F) and without (D) filtering on 50 distinct random splits composed by 1,000 samples each. The whole process was repeated 3 times, each time using a different choice for the filter classifier among those that we produced in Section 6.2. Figure 8

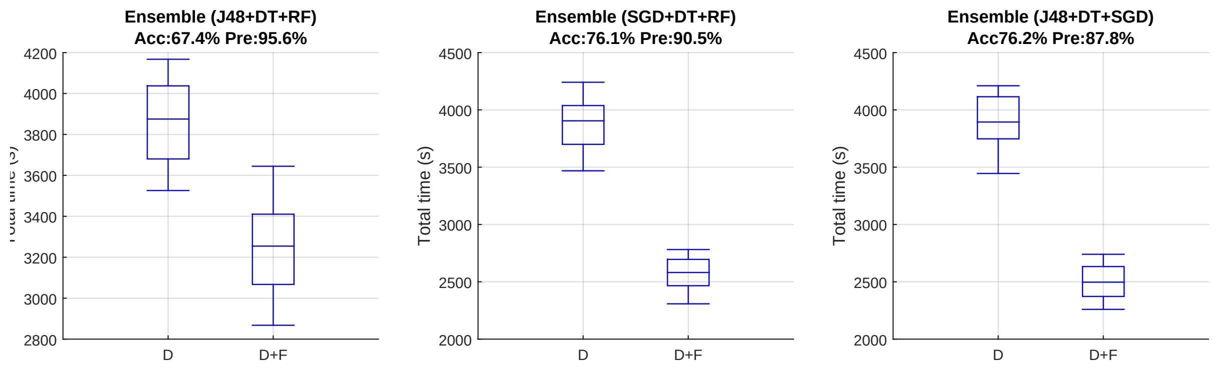


Figure 8: Impact of the filter on the overall system

shows that D+F scenarios always result in a better execution time than D scenarios. Accuracy and precision values reported above each graph represent classification performances when dataset labels are known apriori. We didn't plot detection ratios in the two cases because the filter never misclassifies ransomware, thus resuling in no impact on detection quality. Moreover, consider that this test dataset contains on average way more malware than it is to be expected by an ideal real-world scenario, because we wanted to measure the impact of whitelisting in worst-case conditions. Results show that even in this case we can expect a **50% to 100% speedup**.

CHAPTER 7

LIMITATIONS AND FUTURE WORK

With HELDROID we wanted to provide a proof of concept for our approach, focusing on ransomware detection through natural language processing and static code analysis. The system that we built served its purpose but it is not perfect. In this chapter we observe its flaws and limitations, while paving the road for other possible improvements.

7.1 Portability

Despite our focus being toward the mobile world, ransomware is a general widespread problem. While not trivial, it is feasible to port HELDROID approach onto other operative systems and platforms. Translating the filter and the threatening text detector for a non-mobile setting would be the most straightforward task: thanks to many years of research on malware detection, known heuristics and methods can be refocused to implement a pre-filtering phase; instead, handling text classification would be even simpler, since our approach is transparent with respect to text origin, and mining strings from a binary executable or any configuration file is a known problem. Indeed, the toughest part is related to locking and encryption, which may as well assume a totally new meaning in desktop environments. Our solution takes advantage of the rigid and completely open structure of the Android operative systems, thus making code analysis an almost trivial task. Accomplishing the same objective on different, more complex, systems would be a major challenge. Nevertheless, advancements on static program analysis (64) (14) and reverse engineering (65) on x86 produced an array of tools that would render the task of porting HELDROID to the non-mobile world way easier.

7.2 Internationalization

As already mentioned in Section 6.4, adding support for other languages is a very simple task. Problems arise when facing locales like Chinese, Japanese or Korean, whose grammar is extremely different from other latin or greek based languages, rendering our stemming and segementation rules totally unfit. Since our knowledge about the linguistic topic is extremely limited, it is impossible for us to speculate on how to tackle the problem, but we are confident that similar NLP solutions could be adopted.

7.3 Threatening text evasion

There are several different techniques that could be employed to evade or trick HELDROID text detection module:

- **Encrypt strings.** String literals could be encrypted with a fixed or even variable key, both in the code or in any ancillary asset file packaged within the application archive. They would be decrypted on-the-fly just before use. In our opinion, this constitutes the most straightforward and efficient approach to evade detection. However, there is a rather simple way to deal with this problem: writing an Xposed (66) module to hook every call to `TextView.setText` (and other similar methods) would allow us to hijack every dynamically displayed string in plain text. After capturing decoded strings, we could simply feed them to the threatening text detector to get a normal outcome. Notice that this approach does not even need to deal with decryption. Its downside, of course, consists in having to install such module on every target Android system. Another possible countermeasure could invoke online dynamic analysis platforms like Andrubis (54) or CopperDroid (55) to get a memory dump of string literals.

- **Embed text into images.** Using state-of-the-art OCR software would be enough to demolish any evasion attempt. While additional evasion techniques can be mounted against OCR, recall that the goal of attackers is to make the text as clear and readable as possible for the victims. Employing scrambling methods to avoid optical recognition would be actually counter-productive for the attackers; besides, previous research (67) proved the fallacy of even the most extreme distortion techniques.

- **Display text out-of-band.** Delivering threatening text out-of-band (for example, in a email) would obviously evade our detection. However, as already pointed out in the previous paragraph, this approach would render the ransom process more cumbersome for the victims and in the end act against the attackers interest. Displaying messages interactively and synchronously within the application is way simpler and more effective, making out-of-band threats ill-suited for the task.

- **Mess with grammar.** Our solution is based on the assumption that threatening text is essentially correct with respect to syntax and grammar. Therefore, purposedly introducing spelling mistakes, punctation errors and large numbers of irrelevant words would mess up our classifier. Even if lexical mitigations could be adopted to tamper with this attacks via suitable preprocessing, the vast majority of all analyzed samples tend to contain perfectly written messages, possibly composed by native language speakers. This is actually a social-engineering technique to make attackers' claims sound legitimate. In fact, careful crafting of sentences and precise choice of words are essential steps to any social-engineering attack.

7.4 Classifier obsolescence

Any learning-based technique is subject to obsolescence. Therefore, the classifier should be periodically retrained to adapt to new trends. The availability of vast amount of data sources, like VirusTotal, makes this task feasible.

7.5 Encryption aware APIs

Encryption is one of the strongest point of pressure exploited by ransomware, since it actually gives credibility to textual menaces. A possible way of dealing with ransomware at the system level consists in adding an additional layer of security between applications and encryption APIs. Ideally, encryption of files should be explicitly authorized by users beforehand. This task is not trivial from an usability viewpoint since long sequences of calls and usage of encryption by benign applications could translate into a really annoying series of popups for users to mind to. Moreover, the Android security community already knows very well that handing decision power to users does not really solve the issue. However, this is one of our best choices at the moment.

CHAPTER 8

RELATED WORK

Malware detection. Several different approaches have been proposed in literature, including static (27) (2), dynamic (28) and hybrid (26) techniques. In particular, the first two cited works, MAST and DREBIN, implement very similar solutions to our filter, even if they tackle the problem from the standard perspective of malware detection. DREBIN exhibits 94% accuracy and as low as 1% false positive rate by extracting static features such as permissions, intent filters, contacted hosts and networks, called APIs etc. and embedding them into a vectorial space for training a Support Vector Machine classifier. MAST, on the other hand, employs Multiple Correspondence Analysis to rank applications by suspiciousness: thanks to this ranking, it spots 95% of malware at the cost of analyzing 13% of goodware.

We noted that generic malware detection approaches seem to be inadequate for ransomware detection. In fact, we submitted **Ransomware1** dataset to DREBIN and verified that its detection quality is lower than HELDROID's. Even if in that case the training set of DREBIN was a bit outdated, the authors, which we contacted personally, stated that their system is vulnerable to mimicry attacks. Interestingly, ransomware can be considered to carry mimicry attacks against standard detection systems, since it composes generally benign actions (displaying text, encrypting a file or handling key press) toward a malicious goal (money extortion).

Ransomware detection. To the best of our knowledge, this is the first work on this specific topic. Previous research focused on malicious use of cryptography for implementing ransomware[1] attacks (68) (18), but no explicit detection approach was studied for this class of malware. There exist, however, several commercial solutions that deal specifically with ransomware, but they generally exploit knowledge embedded in signature databases to detect and recover from infections of known samples, exhibiting zero prevention power for new families.

[1]This is an anachronism, since at the time of writing the term didn't exist yet in the form we know today.

CHAPTER 9

CONCLUSION

There is no doubt that ransomware is on the rise and attackers have been working consistently to produce new and more resilient samples (17). Recent events (5) have shown that one ransomware family is enough to single-handedly affect nearly a million users in just one month. Before HELDROID, no specific detection tool existed, and the only available mitigation technique consisted in resorting to generic or signature-based malware detection. Our work showed that HELDROID, once trained on a very small set of ransomware, is able to correctly identify new families and variants, laying the foundations for the design of new proactive detectors implementing effective defensive measures against new threats.

HELDROID provides a publicly accessible JSON API, but can also be integrated within any other scanning or validation platform, rendering it a flexible and usable system for ransomware prevention.

APPENDICES

Appendix A

LIST OF SOURCES AND SINKS

```
# FlowDroid configuration file

<android.os.Environment: java.io.File getExternalStorageDirectory()> -> _SOURCE_

<javax.crypto.CipherOutputStream: void <init>(java.io.OutputStream,javax.crypto.Cipher)>
     -> _SINK_
<javax.crypto.Cipher: byte[] doFinal()> -> _SINK_
<javax.crypto.Cipher: byte[] doFinal(byte[])> -> _SINK_
<javax.crypto.Cipher: byte[] doFinal(byte[],int,int)> -> _SINK_
<javax.crypto.Cipher: int doFinal(byte[],int)> -> _SINK_
<javax.crypto.Cipher: int doFinal(byte[],int,int,byte[])> -> _SINK_
<javax.crypto.Cipher: int doFinal(byte[],int,int,byte[],int)> -> _SINK_
<javax.crypto.Cipher: int doFinal(java.nio.ByteBuffer,java.nio.ByteBuffer)> -> _SINK_
```

Appendix B

COMPLETE LIST OF FEATURES

TABLE IV: COMPLETE LIST OF FEATURES (Before selection).

Group	Name	Data type	Selected
C2M Intent	`c2dm.intent.RECEIVE`	Boolean	✓
	`c2dm.intent.REGISTRATION`	Boolean	✓
	`c2dm.intent.REGISTER`	Boolean	✗
Suspicious Intent Filter	`AIRPLANE_MODE_CHANGED`	Boolean	✗
	`BOOT_COMPLETED`	Boolean	✓
	`CONFIGURATION_CHANGED`	Boolean	✗
	`BATTERY_CHANGED`	Boolean	✗
	`DEVICE_STORAGE_LOW`	Boolean	✗
	`DOCK_EVENT`	Boolean	✗
	`EXTERNAL_APPLICATIONS_AVAILABLE`	Boolean	✗
	`MANAGE_PACKAGE_STORAGE`	Boolean	✗
	`MEDIA_MOUNTED`	Boolean	✗
	`MY_PACKAGE_REPLACED`	Boolean	✗
	`NEW_OUTGOING_CALL`	Boolean	✗
	`PACKAGE_ADDED`	Boolean	✓
	`PACKAGE_CHANGED`	Boolean	✗
	`PACKAGE_DATA_CLEARED`	Boolean	✗
	`PACKAGE_FIRST_LAUNCH`	Boolean	✗
	`PACKAGE_INSTALL`	Boolean	✗
	`PACKAGE_REMOVED`	Boolean	✓
	`PACKAGE_REPLACED`	Boolean	✓
	`PACKAGE_RESTARTED`	Boolean	✗
	`POWER_CONNECTED`	Boolean	✗
	`PROVIDER_CHANGED`	Boolean	✗
	`REBOOT`	Boolean	✗
	`SHUTDOWN`	Boolean	✗
	`USER_PRESENT`	Boolean	✗

(continued on next page)

Appendix B (continued)

Table IV – continued from previous page

Group	Name	Data type	Selected
Api Call	`Landroid/content/Context;->startService`	Boolean	✓
	`Landroid/telephony/TelephonyManager;->getSubscriberId`	Boolean	✓
	`Landroid/telephony/TelephonyManager;->getDeviceId`	Boolean	✓
	`Landroid/telephony/TelephonyManager;->getLine1Number`	Boolean	✓
	`Landroid/telephony/TelephonyManager;->getSimSerialNumber`	Boolean	✓
	`Landroid/telephony/TelephonyManager;->getSimOperatorName`	Boolean	✓
	`Landroid/telephony/TelephonyManager;->getCellLocation`	Boolean	✓
	`Landroid/telephony/cdma/CdmaCellLocation;->getSystemId`	Boolean	✓
	`Landroid/telephony/SmsManager;->sendTextMessage`	Boolean	✓
	`Landroid/content/Intent;->setDataAndType`	Boolean	✓
	`Landroid/content/Intent;->setType`	Boolean	✓
	`Landroid/app/ActivityManager;->getRunningServices`	Boolean	✓
	`Landroid/app/ActivityManager;->getMemoryInfo`	Boolean	✗
	`Landroid/app/ActivityManager;->restartPackage`	Boolean	✗
	`Landroid/content/pm/PackageManager;->getInstalledPackages`	Boolean	✓
	`Ljava/lang/System;->loadLibrary`	Boolean	✓
	`Ljavax/crypto/Cipher;->getInstance`	Boolean	✓
	`Landroid/provider/Browser;->getAllBookmarks`	Boolean	✓
	`Landroid/content/pm/PackageManager;->queryContentProviders`	Boolean	✓
	`Landroid/content/Intent;->describeContents`	Boolean	✓
	`Landroid/content/pm/PackageManager;->getPreferredActivities`	Boolean	✓
	`Landroid/app/Service;->onLowMemory`	Boolean	✓
	`Landroid/os/Parcel;->marshall`	Boolean	✓
	`Ldalvik/system/DexClassLoader;-><init>`	Boolean	✗
	`Ljava/lang/ClassLoader;->loadClass`	Boolean	✓
	`Landroid/accounts/AccountManager;->getAccounts`	Boolean	✓
	`Landroid/content/BroadcastReceiver;->abortBroadcast`	Boolean	✗
Communication	`INTERNET`	Boolean	✓
	`ACCESS_NETWORK_STATE`	Boolean	✓
	`CHANGE_NETWORK_STATE`	Boolean	✓
	`BLUETOOTH_ADMIN`	Boolean	✗
	`BLUETOOTH`	Boolean	✗
	`WRITE_APN_SETTINGS`	Boolean	✓

(continued on next page)

Appendix B (continued)

Table IV – continued from previous page

Group	Name	Data type	Selected
	`NETWORK`	Boolean	✓
	`SUBSCRIBED_FEEDS_WRITE`	Boolean	✓
	`NFC`	Boolean	✗
	`NETWORK_PROVIDER`	Boolean	✓
	`WRITE_SOCIAL_STREAM`	Boolean	✓
	`SEND_SMS`	Boolean	✓
	`USE_SIP`	Boolean	✓
Misc	`Apk Name`	string	✗
	`Checks adb_enabled`	Boolean	✗
	`Tries to modify adb_enabled`	Boolean	✗
	`Airpush Included`	Boolean	✓
	`Classes with Ad prefix`	integer	✓
	`Contains Hardcoded URLs`	Boolean	✓
	`URLs Domains differ from Package`	Boolean	✓
	`Contains URL known to be suspicious`	Boolean	✓
	`Single Package Name`	Boolean	✓
	`Valid Domain in Package Name`	Boolean	✗
	`Package Name contains Tetragrams`	Boolean	✗
	`Total number of packages`	real	✓
	`Total number of classes`	real	✓
	`Number of classes in main package`	real	✓
	`Average class size`	real	✓
	`Obsfuscation present`	Boolean	✓
	`Is main package obfuscated?`	Boolean	✗
	`Size of apk`	real	✓
	`Number of images`	real	✓
	`Number of files`	real	✓
	`Number of permissions`	real	✓
	`Number of activities`	real	✓
	`Number of services`	real	✓
	`Number of receivers`	real	✓
	`Hidden Apk`	Boolean	✗
	`Sends SMS to Suspicious Number(s)`	Boolean	✗

(continued on next page)

Appendix B (continued)

Table IV – continued from previous page

Group	Name	Data type	Selected
	`Package Domain Exists`	Boolean	✗
	`Reads phone data at startup`	Boolean	✓
	`Sends SMS at startup`	Boolean	✗
	`Starts service at startup`	Boolean	✗
	`Sends SMS when receiving SMS`	Boolean	✗
	`Sends data to a remote page when receiving SMS`	Boolean	✗
	`Accesses a database when receiving SMS`	Boolean	✗
Harmless Permission	`ACCESS_SURFACE_FLINGER`	Boolean	✗
	`ACCOUNT_MANAGER`	Boolean	✗
	`ADD_VOICEMAIL`	Boolean	✓
	`CONTROL_LOCATION_UPDATES`	Boolean	✗
	`DEVICE_POWER`	Boolean	✗
	`EXPAND_STATUS_BAR`	Boolean	✓
	`FLASHLIGHT`	Boolean	✓
	`FORCE_BACK`	Boolean	✓
	`GET_PACKAGE_SIZE`	Boolean	✗
	`GET_TOP_ACTIVITY_INFO`	Boolean	✗
	`GLOBAL_SEARCH`	Boolean	✓
	`INSTALL_SHORTCUT`	Boolean	✓
	`MANAGE_DOCUMENTS`	Boolean	✗
	`MODIFY_AUDIO_SETTINGS`	Boolean	✗
	`READ_USER_DICTIONARY`	Boolean	✓
	`REORDER_TASKS`	Boolean	✗
	`SEND_RESPOND_VIA_MESSAGE`	Boolean	✗
	`SET_ALARM`	Boolean	✓
	`SET_ANIMATION_SCALE`	Boolean	✓
	`SET_ORIENTATION`	Boolean	✓
	`SET_POINTER_SPEED`	Boolean	✗
	`SET_TIME`	Boolean	✓
	`SET_TIME_ZONE`	Boolean	✓
	`SET_WALLPAPER`	Boolean	✓
	`UNINSTALL_SHORTCUT`	Boolean	✓
	`VIBRATE`	Boolean	✓

(continued on next page)

Appendix B (continued)

Table IV – continued from previous page

Group	Name	Data type	Selected
	`WAKE_LOCK`	Boolean	✓
	`WRITE_CALENDAR`	Boolean	✓
	`WRITE_CALL_LOG`	Boolean	✗
	`WRITE_CONTACTS`	Boolean	✓
	`WRITE_USER_DICTIONARY`	Boolean	✓
Notification Api Call	`Landroid/support/v4/app/NotificationCompat.Builder;->build`	Boolean	✗
	`Landroid/app/Notification;-><init>`	Boolean	✓
	`Landroid/app/NotificationManager;->notify`	Boolean	✓
	`Landroid/app/Notification$Builder;-><init>`	Boolean	✓
	`Landroid/app/Notification$Builder;->setLargeIcon`	Boolean	✓
	`Landroid/app/Notification$Builder;->setSound`	Boolean	✓
C2M Permission	`c2dm.permission.RECEIVE`	Boolean	✓
	`C2D_MESSAGE`	Boolean	✓
	`c2dm.permission.SEND`	Boolean	✗
Content	`content://com.android.calendar`	Boolean	✓
	`content://calendar`	Boolean	✓
	`content://mms`	Boolean	✗
	`content://sms`	Boolean	✓
	`content://com.facebook.katana.provider.AttributionIdProvider`	Boolean	✓
	`content://telephony/carriers/preferapn`	Boolean	✓
	`content://media`	Boolean	✓
Can Steal Data	`ACCESS_FINE_LOCATION`	Boolean	✓
	`ACCESS_COARSE_LOCATION`	Boolean	✓
	`ACCESS_LOCATION_EXTRA_COMMANDS`	Boolean	✓
	`ACCESS_MOCK_LOCATION`	Boolean	✗
	`ACCESS_WIFI_STATE`	Boolean	✓
	`CAMERA`	Boolean	✗
	`CAPTURE_AUDIO_OUTPUT`	Boolean	✗
	`CAPTURE_SECURE_VIDEO_OUTPUT`	Boolean	✗
	`CAPTURE_VIDEO_OUTPUT`	Boolean	✗
	`DIAGNOSTIC`	Boolean	✓
	`DUMP`	Boolean	✓

(continued on next page)

Table IV – continued from previous page

Group	Name	Data type	Selected
	GET_ACCOUNTS	Boolean	✓
	GET_TASKS	Boolean	✗
	LOCATION_HARDWARE	Boolean	✗
	READ_CALENDAR	Boolean	✓
	READ_CALL_LOG	Boolean	✗
	READ_CONTACTS	Boolean	✓
	READ_EXTERNAL_STORAGE	Boolean	✗
	READ_HISTORY_BOOKMARKS	Boolean	✓
	READ_PHONE_STATE	Boolean	✓
	READ_PROFILE	Boolean	✗
	READ_SMS	Boolean	✓
	READ_SOCIAL_STREAM	Boolean	✓
	READ_SYNC_SETTINGS	Boolean	✗
	RECEIVE_MMS	Boolean	✓
	RECEIVE_SMS	Boolean	✓
	RECORD_AUDIO	Boolean	✗
Dangerous Permission	ACCESS_SUPERUSER	Boolean	✗
	BLUETOOTH_PRIVILEGED	Boolean	✗
	BRICK	Boolean	✗
	CHANGE_COMPONENT_ENABLED_STATE	Boolean	✓
	CLEAR_APP_USER_DATA	Boolean	✓
	DELETE_CACHE_FILES	Boolean	✓
	DELETE_PACKAGES	Boolean	✓
	DISABLE_KEYGUARD	Boolean	✗
	FACTORY_TEST	Boolean	✗
	INSTALL_PACKAGES	Boolean	✓
	INJECT_EVENTS	Boolean	✓
	INTERNAL_SYSTEM_WINDOW	Boolean	✓
	KILL_BACKGROUND_PROCESSES	Boolean	✗
	MASTER_CLEAR	Boolean	✓
	MODIFY_PHONE_STATE	Boolean	✓
	MOUNT_FORMAT_FILESYSTEM	Boolean	✓
	MOUNT_UNMOUNT_FILESYSTEM	Boolean	✗

(continued on next page)

Appendix B (continued)

Table IV – continued from previous page

Group	Name	Data type	Selected
	PROCESS_OUTGOING_CALLS	Boolean	✗
	READ_LOGS	Boolean	✗
	REBOOT	Boolean	✗
	RECEIVE_BOOT_COMPLETED	Boolean	✓
	STATUS_BAR	Boolean	✓
	WRITE_EXTERNAL_STORAGE	Boolean	✓
	WRITE_HISTORY_BOOKMARKS	Boolean	✓
	WRITE_PROFILE	Boolean	✓
	WRITE_SECURE_SETTINGS	Boolean	✗
Calls a System Routine	su	Boolean	✗
	ls	Boolean	✗
	loadjar	Boolean	✗
	grep	Boolean	✗
	/sh	Boolean	✗
	/bin	Boolean	✗
	pm install	Boolean	✗
	/dev/net	Boolean	✗
	insmod	Boolean	✗
	rm	Boolean	✗
	mount	Boolean	✗
	root	Boolean	✗
	/system	Boolean	✗
	stdout	Boolean	✗
	reboot	Boolean	✗
	killall	Boolean	✗
	chmod	Boolean	✗
	stderr	Boolean	✗
	ratc	Boolean	✗

CITED LITERATURE

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and 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 '14, pages 259–269, New York, NY, USA, 2014. ACM.

2. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In Network and Distributed System Security (NDSS) Symposium, San Diego, California, 2014.

3. McAfee Labs: Threats report, november 2014. Technical report, McAfee Labs, November 2014.

4. Virustotal usage statistics. https://www.virustotal.com/en/statistics/. Retrieved Feb 24, 2015.

5. Perlroth, N.: Android phones hit by 'ransomware'. http://bits.blogs.nytimes.com/2014/08/22/android-phones-hit-by-ransomware/, August 2014. Retrieved Feb 9, 2015.

6. Avast Software: Avast ransomware removal. https://play.google.com/store/apps/details?id=com.avast.android.malwareremoval, June 2014. Retrieved Dec 3, 2014.

7. SurfRight: Hitmanpro.kickstart. http://www.surfright.nl/en/kickstart, Mar 2014. Retrieved Feb 19, 2015.

8. ESET: One_half. http://www.eset.com/us/threat-center/encyclopedia/threats/onehalf/, 2004. Retrieved February 15, 2015.

9. Tromer, E.: Cryptanalysis of the gpcode.ak ransomware virus. http://rump2008.cr.yp.to/6b53f0dad2c752ac2fd7cb80e8714a90.pdf, 2008. Retrieved February 15, 2015.

CITED LITERATURE (continued)

10. Micro, T.: Troj_ransom. http://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/troj_ransom, 2012. Retrieved February 15, 2015.

11. Jarvis, K.: Cryptolocker ransomware. http://www.secureworks.com/cyber-threat-intelligence/threats/cryptolocker-ransomware/, Dec 2013. Retrieved February 15, 2015.

12. Lab, K.: Koler - the police ransomware for android. http://securelist.com/blog/research/65189/behind-the-android-os-koler-distribution-network/, June 2014. Retrieved February 15, 2015.

13. Hamada, J.: Simplocker: First confirmed file-encrypting ransomware for android. http://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android, Jun 2014. Retrieved February 15, 2015.

14. Unuchek, R.: Latest version of svpeng targets users in us. http://securelist.com/blog/incidents/63746/latest-version-of-svpeng-targets-users-in-us/, Jun 2014. Retrieved February 15, 2015.

15. Kelly, M.: U.s. targeted by coercive mobile ransomware impersonating the fbi. https://blog.lookout.com/blog/2014/07/16/scarepakage/, July 2014. Retrieved February 15, 2015.

16. Labs, M.: Mcafee labs threat advisory - ctb-locker. https://kc.mcafee.com/corporate/index?page=content&id=PD25696, February 2015. Retrieved February 15, 2015.

17. Chrysaidos, N.: Mobile crypto-ransomware simplocker now on steroids. https://blog.avast.com/2015/02/10/mobile-crypto-ransomware-simplocker-now-on-steroids/, February 2015. Retrieved February 13, 2015.

18. Young, A. and Yung, M.: Cryptovirology: extortion-based security threats and countermeasures. In Proceedings of the IEEE Symposium on Security and Privacy, pages 129–140.

19. Krebs, B.: Inside a "reveton" ransomware operation. https://krebsonsecurity.com/2012/08/inside-a-reveton-ransomware-operation/, Aug 2012.

20. Intelligence, D. S. C. T. U. T.: Cryptowall ransomware. http://www.secureworks.com/cyber-threat-intelligence/threats/cryptowall-ransomware/, Aug 2014. Retrieved February 15, 2015.

21. Bacani, A.: Reveton ransomware spreads with old tactics, new infection method. http://blog.trendmicro.com/trendlabs-security-intelligence/reveton-ransomware-spreads-with-old-tactics-new-infection-method/, Dec 2014.

22. Parkour, M.: Contagio mini-dump. http://contagiominidump.blogspot.it/. Retrieved March 2, 2015.

23. Dex2jar. https://code.google.com/p/dex2jar/. Retrieved March 26, 2015.

24. Luyten java decompiler. https://github.com/deathmarine/Luyten. Retrieved March 26, 2015.

25. Moneypak. https://www.moneypak.com/. Retrieved February 14, 2015.

26. Zhou, Y., Wang, Z., Zhou, W., and Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS).

27. Chakradeo, S., Reaves, B., Traynor, P., and 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, ed. ACM, pages 13–24, New York, NY, USA, 2013.

28. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y.: “andromaly”: a behavioral malware detection framework for android devices. Journal of Intelligent Information Systems, 38:161–190, February 2012. Print ISSN: 0925-9902, Online ISSN: 1573-7675.

29. Apvrille, L. and Apvrille, A.: Pre-filtering mobile malware with heuristic techniques. In The 2nd International Symposium on Research in Grey-Hat Hacking, Grenoble, France, November 2013.

30. Zhou, Y. and Jiang, X.: Dissecting android malware: Characterization and evolution. In Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012), San Francisco, CA, May 2012.

CITED LITERATURE (continued)

31. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., van der Veen, V., and Platzer, C.: Andrubis-1,000,000 apps later: A view on current android malware behaviors. In Proceedings of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014.

32. Aafer, Y., Du, W., and Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In Security and Privacy in Communication Networks, eds. T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, volume 127 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 86–103. Springer International Publishing, 2013.

33. Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D.: Android permissions demystified. In Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, pages 627–638. ACM.

34. Felt, A. P., Ha, E., Egelman, S., Haney, A., and Chin, E.: Android permissions: User attention, comprehension, and behavior. In Proceedings of the Eighth Symposium on Usable Privacy and Security.

35. Felt, A. P., Egelman, S., and Wagner, D.: Ive got 99 problems, but vibration aint one: A survey of smartphone users concerns. Technical Report UCB/EECS-2012-70, EECS Department, University of California, Berkeley, May 2012.

36. List of top-level domains. https://www.icann.org/resources/pages/tlds-2012-02-25-en. Retrieved March 30, 2015.

37. Han, J. and Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann publishers, 2nd edition edition, 2006. University of Illinois at Urbana-Champaign.

38. Kittler, J., Hatef, M., Duin, R. P. W., and Matas, J.: On combining classifiers. IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 20:226–239, 1998.

39. Domingos, P.: Metacost: A general method for making classifiers cost-sensitive. In Fifth International Conference on Knowledge Discovery and Data Mining, pages 155–164, 1999.

40. Kroeger, P.: Analyzing grammar. Cambridge University Press, 2005. Retrieved 2009-07-21.

CITED LITERATURE (continued)

41. Aggarwal, C. C. and Zhai, C.: A survey of text classification algorithms. In Mining text data, pages 163–222. Springer, 2012.

42. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H.: The weka data mining software: An update. 11, 2009. http://www.cs.waikato.ac.nz/ml/weka/.

43. android-apktool. https://code.google.com/p/android-apktool/. Retrieved February 14, 2015.

44. Arnatovich, Y. L., Beng, H., Tan, K., Ding, S., Liu, K., and Shar, L. K.: Empirical comparison of intermediate representations for android applications.

45. Static android analysis framework. https://code.google.com/p/saaf/. Retrieved February 16, 2015.

46. Hoffmann, J., Ussath, M., Holz, T., and Spreitzenbarth, M.: Slicing droids: Program slicing for smali code. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, pages 1844–1851, New York, NY, USA, 2013. ACM.

47. Shuyo, N.: Language detection library for java, 2010.

48. Apache Foundation: Opennlp. https://opennlp.apache.org/. Retrieved February 24, 2015.

49. Xxvi congress of the cpsu and challenges of social psychology. http://www.voppsy.ru/issues/1981/816/816005.htm. Retrieved February 21, 2015.

50. Wikipedia: Law. https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B0%D0%B2%D0%BE. Retrieved February 27, 2015.

51. The snowball language. http://snowball.tartarus.org/. Retrieved February 28, 2015.

52. The stop-words project. https://code.google.com/p/stop-words/. Retrieved February 28, 2015.

53. van der Veen, V., Bos, H., and Rossow, C.: Dynamic Analysis of Android Malware. Master's thesis, VU University Amsterdam, August 2013.

54. Andrubis Website, 2015.

55. Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L.: CopperDroid: Automatic reconstruction of android malware behaviors. In Proceedings of the Network and Distributed System Security Symposium (NDSS). Internet Society.

56. Lindorfer, M., Volanis, S., Sisto, A., Neugschwandtner, M., Athanasopoulos, E., Maggi, F., Platzer, C., Zanero, S., and Ioannidis, S.: AndRadar: Fast discovery of android applications in alternative markets. In Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), volume 8550, pages 51–71. Springer-Verlag.

57. Blackmart alpha. http://www.blackmart.us/. Retrieved February 17, 2015.

58. Opera mobile store. http://apps.opera.com/. Retrieved February 17, 2015.

59. Camangi market. http://www.camangimarket.com/index.html. Retrieved February 17, 2015.

60. Pandaapp. http://www.camangimarket.com/index.html. Retrieved February 17, 2015.

61. Slideme. http://slideme.org/. Retrieved February 17, 2015.

62. Getjar. http://www.getjar.mobi/. Retrieved February 17, 2015.

63. Maggi, F., Valdi, A., and 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 & mobile devices, ed. ACM, pages 49–54, New York, NY, USA, 2013.

64. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In Proceedings of the 4th International Conference on Information Systems Security, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

65. Slowinska, A., Stancescu, T., and Bos, H.: Howard: a dynamic excavator for reverse engineering data structures. In Proceedings of NDSS 2011, San Diego, CA, 2011.

66. Xposed framework. http://repo.xposed.info/. Retrieved March 11, 2015.

CITED LITERATURE (continued)

67. Bursztein, E., Martin, M., and Mitchell, J.: Text-based captcha strengths and weaknesses. In Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pages 125–138, New York, NY, USA, 2011. ACM.

68. Young, A.: Cryptoviral extortion using microsoft's crypto api. International Journal of Information Security, 5(2):67–76, 2006.

69. Alessandro, S. and Zanero, S.: AndroCrawl : studying alternative Android marketplaces. Master's thesis, Politecnico di Milano, December 2013.

70. Zhou, Y., Wang, Z., Zhou, W., and Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternativeandroid markets. In Network and Distributed System Security (NDSS) Symposium, February 2012.

71. Smali dissasembler. `https://code.google.com/p/smali/`. Retrieved February 28, 2015.

72. Google play services. `https://developer.android.com/google/play-services/`. Retrieved February 9, 2015.

73. Spagnuolo, M., Maggi, F., and Zanero, S.: BitIodine: Extracting intelligence from the bitcoin network. In Financial Cryptography and Data Security, Lecture Notes in Computer Science (LNCS). Springer-Verlag.

74. Ransomware on the rise. `http://www.fbi.gov/news/stories/2015/january/ransomware-on-the-rise`, January 2015. Retrieved Feb 18, 2015.

75. Nielsen: How smartphones are changing consumers' daily routines around the globe, Feb 2014.

76. Schwartz, E. J., Lee, J., Woo, M., and Brumley, D.: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In USENIX Security, 2013.

77. Jarabek, C., Barrera, D., and Aycock, J.: Thinav: Truly lightweight mobile cloud-based anti-malware. In Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12, pages 209–218, New York, NY, USA, 2012. ACM.

CITED LITERATURE (continued)

78. Truong, H. T. T., Lagerspetz, E., Nurmi, P., Oliner, A. J., Tarkoma, S., Asokan, N., and Bhattacharya, S.: The company you keep: Mobile malware infection rates and inexpensive risk indicators. In Proceedings of the 23rd international conference on World Wide Web (WWW).

79. Lever, C., Antonakakis, M., Reaves, B., Patrick, T., and Lee, W.: The core of the matter: Analyzing malicious traffic in cellular carriers. In Proceedings of the ISOC Network & Distributed Systems Security Symposium.

VITA

NAME:	Nicolo' Andronio
EDUCATION:	B.S., Computer Engineering, Politecnico di Milano, Italy, 2012
	M.S., Computer Engineering, Politecnico di Milano, Italy, 2015