공학석사 학위논문

Security Analysis of Secure Virtual Keyboards in Android Mobile Payment Apps

안드로이드 금융앱 보안키패드 분석

2016 년 2 월

서울대학교 대학원

전기컴퓨터공학부 컴퓨터공학전공

최 서 윤

Security Analysis of Secure Virtual Keyboards in Android Mobile Payment Apps

안드로이드 금융앱 보안키패드 분석

지도 교수 전 병 곤

이 논문을 공학석사 학위논문으로 제출함

2016 년 2 월

서울대학교 대학원

컴퓨터공학부

최 서 윤

최서윤의 공학석사 학위논문을 인준함

2016 년 2 월

위 원 장 염 헌 영 **(인)**

부위원장 전 병 곤 **(인)**

위 원 엄 현 상 **(인)**

Abstract

Security Analysis of Secure Virtual Keyboards in Android Mobile Payment Apps

Seoyoon Choi

Department of Computer Science and Engineering

College of Engineering

The Graduate School

Seoul National University

Mobile payment applications typically employ extra security measures due to the sensitivity of information that they handle. This paper investigates the security of secure virtual keyboards which are frequently used in South Korea. Unlike numerous studies on Android apps in the past, analyzing payment apps is particularly challenging as they use obfuscation. To overcome these difficulties, we extend TaintDroid to leverage the user interfaces that keyboards use to interact with others. With the tool, we examine how securely these apps handle encrypted user input through secure virtual keyboards. We find that although these apps encrypt user data through a third-party secure virtual keyboard library to protect against memory dumping attack, all the target apps decrypt all the sensitive information using the decryption APIs of secure virtual keyboard libraries, increasing a vulnerability time window. We conclude the paper with a discussion of possible countermeasures.

Keywords: Security Virtual Keyboard, Encryption, Input Taint Tracking, Reverse Engineering

Student Number: 2014-21795

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Mobile payment applications (referred to as apps henceforth) running on smartphones are gaining ubiquity thanks to their convenience. People can purchase products online and in retail stores at their fingertips with these apps: examples include retailers' apps (e.g., CurrentC [17]), bank and credit card apps, third-party payment apps (e.g., a payment module added on messenger apps), and device vendors' apps (e.g., Apple Pay and Samsung Pay). It is estimated that the annual global mobile payment market will reach 2.8 trillion US dollars in 2020 [16]. With the growth of the mobile payment market, the concerns of the apps' security will increase.

Attacks on mobile payment apps can cause direct financial losses and sensitive user data leakages, because the apps perform monetary transactions using sensitive information such as credit card numbers and passwords. Therefore, mobile payment apps typically employ many security techniques. For user verification, they often use personal identification numbers (PINs), one time password (OTP) [7], and twofactor authentication [21]. In addition, payment apps typically encrypt the sensitive data [22] and communicate with server via secure connections, such as SSL/TLS [26], [27], [14]. However, they are vulnerable to various attacks such as keyboard hooking attack [19], [15], and memory dumping attack [12].

Because it is critical to secure mobile payment apps, in South Korea the government recommends that financial companies adopt some extra security measures, and most of Android apps implement them. In particular, to protect inputs on touch screen displays, the apps use a secure virtual keyboard (referred to as SVK henceforth), which is specially designed for financial apps. SVKs randomize the layout of letters to defend against key-logging attacks and encrypt user input to avoid memory dumping attacks.

1.1 Motivation

To follow the government recommendations, most of mobile payment apps use SVKs that security companies develop. However, there has been no prior security analysis showing that these added security measures are really effective. In this paper, we evaluate the effectiveness of SVK in Android payments apps in South Korea. Our in-depth analysis of four popular apps reveals weaknesses of the security features. We hope that our research sheds lights on understanding these security measures and will facilitate future research in the security of mobile payment apps.

Specifically, we focus on the following research questions:

- Do mobile payment apps handle private user credentials entered from SVKs safely without decrypting? If not, is the decryption absolutely necessary?

1.2 Approach

Analyzing the mobile payment apps is challenging because these apps use various measures against reverse engineering to prevent malicious attacks. Payment apps often heavily use the code obfuscation, which makes it difficult to understand actual control flows statically. To overcome these difficulties, we leverage the user interfaces that keyboards use to interact with users. keyboards use Android APIs related to user interface (UI), such as TextField and TouchEvents to handle the private data. This implies that certain UI-accessing APIs are the start points to trace the sensitive data. With these UI-based hints, we successfully scope down the amount of app code to analyze.

To investigate the internals of mobile payment apps, we implemented an automated tool based on the UI-based approach and combined it with existing security tools that we improved to our purposes. We run the apps in an enhanced TaintDroid [23], a taint tracking system for Android, to track how the sensitive user input flows from a SVK within an app. The enhanced TaintDroid shows the methods that use the sensitive data, we thereby find out how and why the methods handle the data.

1.3 Contribution

We analyzed four popular Android mobile payment apps published in South Korea. From our analysis, we show that the apps expose private information as plaintext in the memory of the apps for unnecessary reasons, which makes these apps susceptible to data leaks. These weaknesses we found demonstrate that the extra security measures are not effective. It provides users only a feeling of improved security but does not actually achieve it.

In this paper, we make the following contributions:

- This study is the first to analyze the (in)effectiveness of SVKs, the security measures widely used in Android mobile payment apps.
- We have shown how app developers misuse SVKs to keep sensitive information longer than needed, thus increasing the vulnerability window.

1.4 Outline of the paper

The rest of the paper is structured as follows. Chapter 2 presents backgrounds on mobile payment apps and their extra security measures. Chapter 3 details our methodologies. Chapters 4 present the analysis of SVKs, respectively. Chapter 5 discusses the implications of our findings, and Chapter 6 concludes.

Chapter 2

Background

We start by describing mobile payment apps and their security threats. We then describe secure virtual keyboards, the extra security measures we focus on.

2.1 Android Mobile Payment Apps

Mobile payment apps enable users to make payments conveniently. They serve requests from other apps that require billing services. In addition, these apps operate in a standalone mode for users to register and manage their own payment information.

Mobile payment apps typically operate in the following steps: Initially users register their user credentials such as credit card number, password, and card validation code (CVC). After registering, users can conveniently make payments using the app with a simple authentication process.

In South Korea, most e-commerce web sites and mobile shopping apps widely adopt the mobile payment apps for billing. The payment apps in South Korea can be classified into two categories:

- **Payment service providers (PSPs)** support various cards issued by different card companies. These apps are deployed by companies that are not banking or credit card companies. The users can register multiple cards and choose one of them during the payment process.

- **Appcards** are provided by banking or card companies. They work with only the cards issued by the company. Users can register their existing plastic cards or receive a new mobile card.

2.2 Security Threats in Payment Apps

Mobile payment apps are also vulnerable to common Android security issues, such as app repackaging attacks [20], [24], root exploits [3], [4], [28], and malware [25], [13], [6]. In addition, device rooting, which users may do [18], makes the problems worse.

Attackers have big incentives to target mobile payment apps, because these apps are capable of performing monetary transactions and dealing with sensitive user information such as financial information and personal credentials. Attackers can initiate or manipulate financial transactions by impersonating a user or tampering an app. By making false payments, they can profit from users' accounts. They can also steal sensitive user information that is related to financial transactions and sell or use the information for their profits.

2.3 Secure Virtual Keyboards

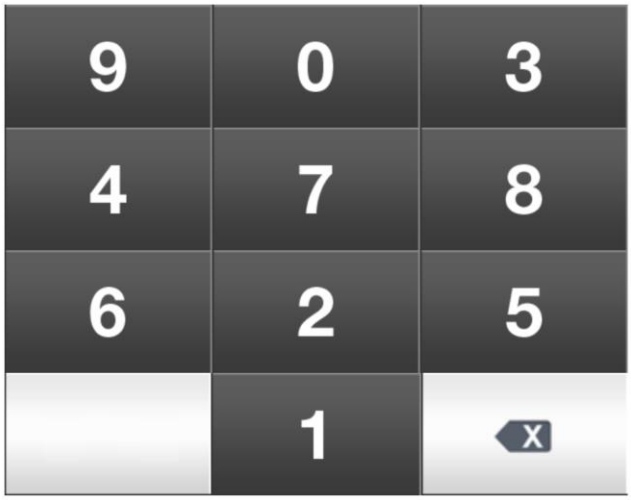

Figure 2.1: Screenshots of mobile payment apps' SVK.

In this paper, we evaluate the effectiveness of secure virtual keyboards that are important and common in payment apps used in South Korea. A secure virtual keyboard is a specialized input method that prevents the exposure of user input by randomizing layout and

encrypting input data. Most mobile payment apps employ this technique, which third-party security companies provide, to protect sensitive financial user input.

The Android default system keyboard processes a user input as plaintext in apps' memory space. This allows the attackers to loot the information through memory dumping [12] or keyboard hooking [19], [15]. To protect sensitive user information from these attacks, mobile payment apps use SVKs. While users registering the apps, they provide SVKs for entering the private information such as credit card number and password. User interface of SVKs are similar with Figure 2.1, which provide randomization of the layout. The layout is randomized whenever the keyboard is brought up; this prevents keyboard hooking attacks as the attackers cannot guess the key values from the touch coordinates. Furthermore, mobile payment apps encrypt the user input using SVKs, such that it can be decrypted only on their private servers, preventing the attackers from comprehending user information obtained through memory dumping.

Chapter 3

Methods

We introduce the target Android payment apps. Then, we present our analysis methodology that combines dynamic and static analysis. Finally, we describe three program points where the target apps decrypt the SVK-encrypt data, which is the focus of our analysis.

App Name	Apay	Bpay	Cpay	Dpay
Downloads	1M - 5M	10M - 50M	100k - 500k	5M - 10M
Category	Payment Service Provider		Appcard	

Table 3.1: A list of our target apps. The names of the apps are anonymized.

3.1 App Selection

In Google Play's Finance-Free category in South Korea, we found 19 mobile payment apps that run on Android 4.3 and 4.4 (in the last week of June 2015). We selected four mobile payment apps to analyze as shown in Table 3.1. These four apps were selected to reflect the diversity across categories (payment service providers or appcards) and popularity (download counts).

3.2 Analysis Methodology

We use dynamic and static analysis to track sensitive entered into the SVKs. For dynamic analysis, we extend TaintDroid [23], a widely used information flow tracking system

on Android, to detect methods which use the sensitive data. Then, for the detected methods, we use reverse engineering techniques to examine how they handle the sensitive data.

Step 1) Tracing Sensitive Data with TaintDroid: To identify methods which utilize the sensitive data, we modify TaintDroid. With TaintDroid, we can track and trace the flow of the tainted data. The following steps show how we extended TaintDroid to track sensitive data from the SVKs.

1.1) To trace sensitive data, we first add taint markings on the sensitive data while an *Activity* for registering user credentials is displayed. In particular, we utilize two user interface classes related to input event:

- **EditText.** Mobile payment apps commonly use EditText to retrieve text input from the user. App developers can easily access the input text from EditText through the *getText()* API. We instrument *getText()* to add taint markings to the returned text.

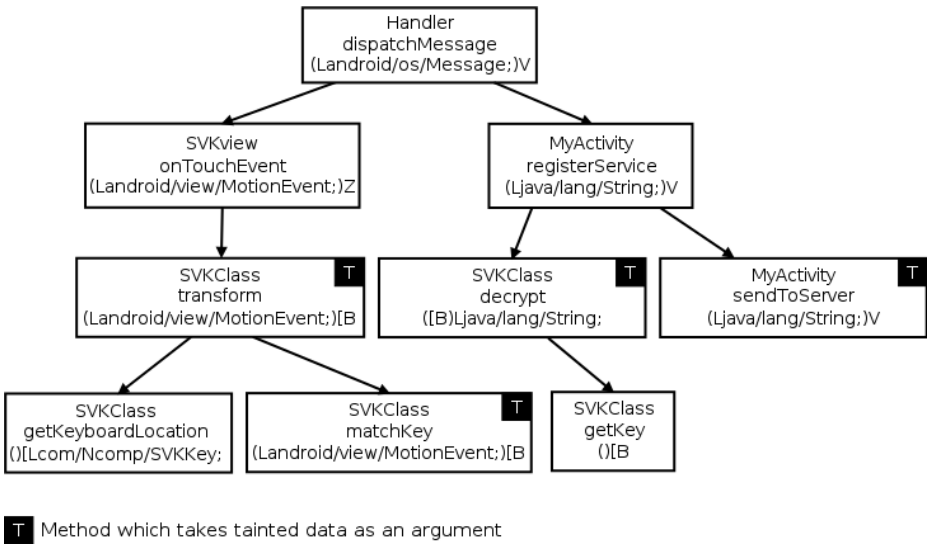

Figure 3.1: Tracing sensitive data with TaintDroid

- **MotionEvent.** Some mobile payment apps use customized components to handle user input instead of using a regular text view. In this case, we note that the apps use MotionEvent to handle touch events. When a user touches the keyboard, Android delivers a touch

event to the appropriate View with the touch coordinate through this class. We add taint markings on the touch coordinate values by modifying the MotionEvent APIs.

1.2) To get a method call graph which marks the methods of processing tainted user input, we instrument Dalvik interpreter. We record method calls of the entire lifecycle of the SVK-embedded Activity, from *onCreate()* to *onDestroy()*, by applying the methodology from Compac [1]. Then, we mark the methods which process the tainted data as an argument by modifying *invoke* opcodes, which is related to method invocation, in Dalvik interpreter. As a result we obtain a method call graph as shown in Figure 3.1. This graph identifies the methods related to user input, which are our main interests.

Step 2) Analyzing Identified App Methods: In order to examine how the app handles user input, we reverse engineered the app based on the method call graph created by Step 1. To analyze the app code statically, we decompile the apps using dex2jar [5] and jd-gui [10]. These tools are helpful by showing the Java source code, which is easier to read than Dalvik bytecode. However, the mobile payment apps are heavily obfuscated, so many methods are not decompiled. Thus, instead of decompilation, we need to disassemble them using apktool [2]. To understand how the methods work in more details, we debug the apps using a commercial debugger, IDA [8].

3.3 Analysis Targets

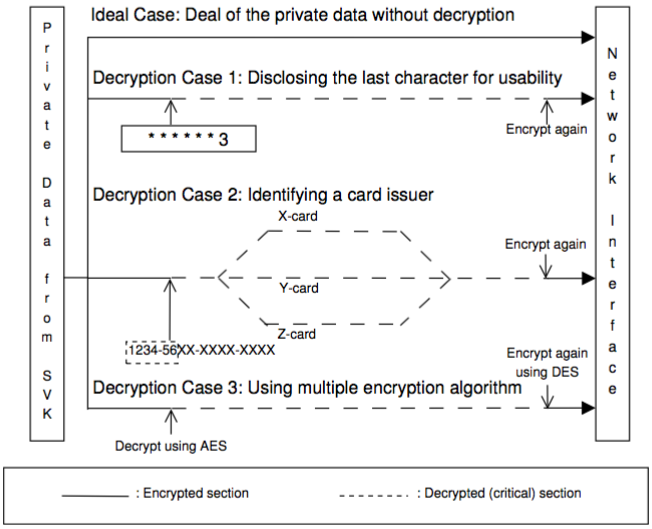

Figure 3.2: An ideal case and decryption cases of handling private data from a SVK inside an app.

The purpose of SVKs is preventing attackers from reading users' private information from the apps' memory. To prevent the sensitive information from being exposed in memory as plaintext, the SVK maps the user input to randomly created values digit by digit, and then encrypt the data with its cipher module. The app is then able to handle the information as ciphertext and send it to the app server without ever decrypting it in the device end. Since the server is more secure than the mobile device, the app with SVK is able to decrypt and manipulate user data more safely. To check the effectiveness of using SVK libraries in terms of security, we analyze whether the target apps maintain the sensitive data given from the SVKs as encrypted. By using the above tool, we find that the target apps commonly decrypt the data at the following three program points as illustrated in Figure 3.2.

Case 1. Disclosing the last character for usability. The mobile payment app usually hides user input characters (e.g., credit card numbers). But in some cases, the app displays the last character entered in a clear text format. This feature provides improved usability by reducing

typing errors.

Case 2. Identifying a card issuer. The PSP app supports multiple cards from different companies. Since each card company has its own authentication mechanism, the app needs to identify a card issuer. Therefore, the apps want to get the first six digits of a card number, which represents the card issuer [9], to identify a card company.

Case 3. Using multiple encrypting algorithms. The mobile payment app encrypts data with its encryption algorithm before sending user information to the server. Different card companies use different encryption algorithms or keys in the authentication processes. The app may decrypt the private data from the SVK to re-encrypt it with an appropriate algorithm.

Chapter 4

App Analysis

Private information		Apay	Bpay	Cpay	Dpay
Card Information	Card Number	○	○	○	○
	Expiration Date	○	○	○‡	○‡
	CVC	○	○	○	○
	Card Password	○∗	○∗	○	○
Social Security Number		—	—	○	○
PIN Number		○	○	○	○

○: Sensitive user input exposed as plaintext
—: Not required data
‡: User input from a regular virtual keyboard
∗: An app gets only part of the data

Table 4.1: List of private information revealed as plaintext by each app.

Using the methodology discussed above, we identify the methods that handle the sensitive data as plaintext. To our surprise, as Table 4.1 shows, all the target apps decrypt all the sensitive user data using the decryption APIs of SVK libraries. This makes the user data vulnerable to memory dumping attacks. We analyze why the apps decrypt the private information and whether the reason is acceptable at each of the three decryption points.

```
1  public class Apay_CardRegister {
2
3     EditText editCardNumber;
4     private String G_cardNumber;
5     private String plainText;
6
7     public void onSVKClick(SvkKeyboard svkKeyboard) {
8        if (svkKeyboard.getData() != null) {
9           byte[] plainDataByte =
                 Xlibrary.decrypt(svkKeyboard.
                 getFocus());
10          plainText = new String(plainDataByte);
11       }
12       if (edtCardNumber.hasFocus()) {
13          editCardNumber.setText(
                 ApayUtils.starChange(plainText));
14          G_cardNumber = plainText;
15       }
16    }
17
18    private void goNext() {
19       Intent intent = new Intent();
20       intent.putExtra("card_number",
              G_cardNumber);
21    }
22
23 }
```

Listing 4.1: A part of Apay's implementation which shows the last entered character. We omit, simplify, and rename the code.

1) Disclosing the last character for usability: To display the last character of user input, Bpay and Cpay uses the feature provided by the SVK libraries. However, Apay implements its own version of this feature without paying attention to security. Since security libraries such as SVK are more aware of security issues, it is a safer decision to use the features provided by SVK libraries.

Cpay (Xlibrary) Cpay utilizes the Xlibrary to show the last character. It first decrypts the ciphertext to byte array and displays the last character. Then it clears the memory by writing zeros to the byte array, which shortens the exposure time of the plaintext.

Apay (Xlibrary) Even though it also uses the Xlibrary as Cpay, the app uses self-implemented version of this feature as shown in Listing 4.1. It also decrypts the entire ciphertext using the decrypt() API to show the last entered character (line 9). The method starChange() in line 13 hides user input behind asterick characters and only shows the last entered character. The implementation of Apay is insecure because it stores sensitive input as plaintext in a member variable, and after that continuously handles the decrypted data (Lines 4, 20). In other words,

the app does not take much advantage of SVK over any regular virtual keyboard.

Bpay (Ylibrary) The app uses a different SVK library from the two previous apps. The Ylibrary provides its SVK in the form of an *Activity* with a custom text field for displaying the last character. This custom text field displays its characters as images as opposed to plaintext. To display the appropriate image for a given character, the library decrypts user input using DEC_WORD().

```
1   public class Bpay_CardRegister {
2
3     public void registerCard(UserInfo user) {
4       String plain_cardnumber =
            Ylibrary.getDecryptCipherData(
5                         encryptCardNum);
6       boolean b1 = plain_cardnumber.startsWith(
7                         cardcompany1_code);
8       if (b1) {
9         sendData(user.getDataType1());
10      } else {
11        sendData(user.getDataType2());
12      }
13    }
14
15  }
```

Listing 4.2: A part of Bpay's implementation which decrypts the SVK encrypted data to identify a card company. We omit, simplify, and rename the code.

2) Identifying a card issuer: The first six digits of a card number identify a card issuing company while the other digits identify a card holder. Therefore, decrypting the entire card number just to identify a card issuer exposes more private information than necessary.

Bpay (Ylibrary) Listing 4.2 describe how the app identifies a card issuer. Because a type of Bpay is a payment service provider, it needs to identify a card issuer to submit corresponding authentication information (Lines 7 - 12). Bpay gets the whole card number through SVK in a text field, thus it decrypts the card number entirely using getDecryptCipherData() in order to know the front digits identifying card issuer (Line 5). Bpay can get this information without decryption, as Apay does in the following.

Apay (Xlibrary) Apay is also a PSP app, but it does not decrypt user information for this

purpose. When users register their card with Apay, they first need to choose their card company and then enter the card information. To identify the card company, Apay uses this selected data instead of decrypting the whole card number. As this case shows, app developer can minimize decryption cases by utilizing app design.

	Apay	**Bpay**	**Cpay**	**Dpay**
SVK	SEED	SEED	SEED	SEED
App Custom	Public-key	SEED	DES	AES
Strategy	○	●	○	●

○: Decryption and encryption
●: Double encryption

Table 4.2: Using multiple encryption algorithms

3) Using multiple encryption algorithms: All the four target apps apply their own encryption algorithm before sending the data to a private server. There were two different ways in doing this. Dpay and Bpay encrypt once again the already encrypted SVK input with their own encryption algorithms, which do not expose the sensitive data as plaintext. In contrast, Apay and Cpay first decrypt the SVK encrypted input, then re-encrypt the plaintext with their own encryption. This method of transmission dangerously exposes sensitive data as plaintext in the mobile device. The results of this analysis are summarized in Table 4.2 and further discussed below.

```
1  public class Cpay_CardRegister {
2
3    EditText editCardNumber;
4    EditText editCardPassword;
5
6    public void registerCard() {
7      String cardNum =
           editCardNumber.getText().toString();
8
9      String cardPassword =
           editCardPassword.getText.toString();
10     localHashMap.put("cardNumber",
           Xlibrary.decrypt(cardNum));
11
12     localHashMap.put("cardPassword",
           Xlibrary.decrypt(cardPassword));
13     verityCardForRegister(localHashMap);
14   }
15
16   public static void verifyCardForRegister
17        (HashMap<Stirng, Object> paramHashMap) {
18     String userdata = encryptDES(paramHashMap);
19     sendHttps(userdata);
20   }
21
22 }
```

Listing 4.3: A part of Cpay's implementation which decrypts the SVK encrypted data and re-encrypt the plaintexts with its own encryption algorithm. We omit, simplify, and rename the code.

Apay, Cpay (Xlibrary) These apps decrypt the SVK-encrypted data using the SVK's decryption API decrypt(), and apply their own encryption before sending it to their servers, which exposes the sensitive data in memory as plaintext. We will show how Cpay works in detail in Listing 4.3. Cpay decrypts the SVK input encrypted with SEED algorithm [11], and saves the plaintext in a HashMap (Lines 7 - 12). The app calls encryptDES() to concatenate all user information in a string and encrypts it using DES (Line 18). The app then sends user data with HTTPS protocol using sendHttps() (Line 19). Though transmitting the data safely using HTTPS, it could send the SVK-encrypted data and re-encrypt it as below two apps do, but it doesn't. Apay also has the same problem. Apay use the public-key cryptography to encrypt the decrypted user data with the public key of the corresponding card company. For doing this, the app uses the already decrypted user input when showing the last entered character. The re-encrypted card information is then sent to the Apay's server. In this design, Apay can send the

SVK-encrypted data and handle it safely at the server without decrypting it at the device end, but it doesn't.

Bpay (Ylibrary), Dpay (Xlibrary) Compared to the above two apps, these apps are more secure because they doubly encrypt the sensitive data instead of decrypting and encrypting again. Dpay encrypts the SVK-encrypted data with the AES algorithm to send it to a private server. As a result, the data is doubly encrypted with SEED by the SVK and then with AES by the app. Bpay also uses double encryption in a similar fashion to Dpay, with the difference that Bpay encrypts the SVK-encrypted data one more time with the same SEED method.

Summary. We examined the cases in which the target apps decrypt sensitive user data obtained through SVK. Even though the four apps use SVK libraries, the level of security achieved through such libraries varies according to how the app developers handle the SVK-encrypted data. The more comprehensive APIs of SVK libraries, such as secure implementations of frequently used features, can improve the security of apps which use the libraries. The app developers then can focus on the development of application logic and leave more complex security details up to the library, which reduces the chances of leakage of sensitive data. We discuss suggestions of improved SVK APIs in Chapter 5.

Chapter 5

Discussion

```
1  public class CardRegisterActivity extends Activity {
2
3     private SVKEditText cardNumEdit, cardPassEdit;
4
5     public void onCreate() {
6        cardNumEdit = findViewById(R.id.editCard);
7        cardPassEdit = findViewById(R.id.editPass);
8        cardNumEdit.setShowLastCharacter(true);
9     }
10
11    private void sendData() {
12       SVKData cardNum = cardNumEdit.getData();
13       SVKData cardPass = cardPassEdit.getData();
14       int cardType = cardNum.partialDecrypt(0,6);
15
16       SVKDataGroup dataGroup = SVKDataGroup();
17       if (cardType == 101010) {
18          dataGroup.append(cardNum);
19          dataGroup.append(cardPass);
20          JsonObject json =dataGroup.getJson(
                 SVKLib.EncryptType.RSA,
                 public_key_bank1 );
21       } else {
22          dataGroup.add(cardNum);
23          JsonObject json =dataGroup.getJson(
                 SVKLib.EncryptType.RSA,
                 public_key_bank2 );
24       }
25    }
26
27 }
```

Listing 5.1: An example that uses the proposed APIs.

Our findings show that mobile payment apps do not fully take advantage of SVK libraries. In some cases, the app developers do not understand well how to use the SVK libraries and they trade off security for convenience. We think that there is an interesting research direction on how to improve the usability of security libraries such as SVK libraries.

One way to improve the usability of SVK-encrypted data is to provide functions commonly used by application developers as part of SVK APIs. App developers can use them to handle encrypted user input more safely. Here we sketch a few APIs that cover the three

program points where the payment apps decrypt sensitive data (Chapter 3.3). The first API is SVKEditText, a custom EditText View class, that provides setShowLastCharacter(boolean). When the option is true, SVKEditText displays the last entered character while minimizing the time to keep it in plaintext. The second API is SVKData. SVKEditText returns encrypted user data as an SVKData object. SVKData provides partialDecrypt(int start, int end), which returns decrypted values from start index to end index. This API can be used to identify a card company by accessing the first 6 digits. Lastly, since the payment apps apply their own encryption algorithms before transmitting data to servers, we provide an API to retrieve SVK-encrypted-data encrypted by requested algorithms. To encapsulate a set of encrypted data, we provide SVKDataGroup. This class provides append(), which appends multiple SVKData objects. The class provides a method to return the data in SVKDataGroup as a JSON object. The data is re-encrypted with the algorithm given to the method. Listing 5.1 shows an example of the suggested APIs. Line 8 is a setShowLastCharacter(true) call to display the last charactertyped.Line14isapartialDecrypt(0, 6)call for obtaining a card issuer. In lines 18-19, we show how to use SVKDataGroup to return a JSON object with data encrypted again with the provided algorithm.

Chapter 6

Related Work

Financial security. Much research has been done to diagnose the security-level of the financial services deployed in various forms such as mobile app and web service. Recent studies investigated security in branchless banking apps, an emerging financial service [29], [30]. Joel et al. showed that security images, which prevent phishing attacks on Internet banking, are ineffective due to user's ignorance [31]. In contrast, our work focuses on analyzing how effective third-party security libraries are to improve the security of mobile payment apps.

User data privacy. Taint tracking is one of the famous methodologies for Android privacy research, which adds taint markings to the important information and tracks it. TaintDroid [23] is a dynamic taint tracking system which monitors the flow of sensitive information and detects leakages of the information. DroidScope [32] is a platform for virtualization-based malware analysis by reconstructing both the OS-level and Java-level semantics. It also uses the dynamic taint tracking methodology as a part of the analysis. On the other hand, FlowDroid [33] is a static taint analysis tool for detecting privacy leaks in Android apps. Klieber et al. also introduced a static taint flow analysis tool to track inter-component and intra-component data flows between multiple Android apps [34]. Our tool analyzes the detailed step of data propagation by modifying TaintDroid and combining it with our method call tracking tool.

Securing user input. Like SVKs which we analyzed in this paper, much research focuses on securing the user input, a primary source of privacy sensitive user data. Chen et al. performs a systematical study to understand the threat caused by the leakage of private sensitive user input

through input method editor (IME) apps [19]. The research proposes sandboxing to defend the security threats posed by IME Apps. In contrast, our research focuses on analyzing the flow of user input in the apps. Thus, we make an efficient analysis tool to track user input and identify methods which utilize the user input.

UIPicker [35] and SUPOR [36] are static analysis tools that automatically identify sensitive information among input data entered through UI. They deal with all types of input entered through text input fields in addition to the predefined resources such as built-in keyboards. Similarly, our research analyzes text fields that receive data entered via SVKs. In our research, we can assure that all the data entered via SVKs are privacy-sensitive, because the mobile payment apps provide SVKs only for user credentials.

Android app security. Our research applies to security of general Android apps, which other research is also interested in. Enck et al. analyzed various Android apps and uncovered pervasive misuse of personal information, as well as instances of deep penetration of advertising networks [37]. Many researchers have examined Android interactions and have identified security risks in permission systems and communication systems [38], [39], [40]. Several other researches have investigated SSL/TLS security, or the lack of it, on Android apps [26], [30], [41]. Egele et al. studied the misuse of cryptographic APIs, which secure data such as passwords and personal information [22]. Our research analyzed app behaviors on using third-party libraries by using dynamic and static analysis techniques with UI-based hints. The purpose of our analysis was to determine whether apps succeed to ensure data privacy by using third-party security libraries.

Security assessment tool. Our research also contributes to the creation of automated tools to assess or enhance Android app security. Android has various means by which to measure app security. MalloDroid is a static tool which investigates apps with regard to the correct usage of SSL/TLS. Jeon et al. presented a tool to infer and enforce finer-grained permissions on Android apps [42]. Compac [1] is a system which provides component-level access control to confine the

permission of third-party libraries. By extending the findings of prior studies [23] and adding new findings, our tool reveals vulnerabilities related to app obfuscation and data leakages.

Chapter 7

Conclusion

In this paper, we analyze four popular Android mobile payment apps in South Korea to investigate SVKs that protect the apps and their sensitive data. By leveraging the UI events of the measures such as text fields, and touch events, we have semi-automated our analysis and have reduced the manual effort we need significantly. Our analysis shows that the apps misuse SVKs to keep users' private information longer than needed. We hope that this work leads to future research on improving the security of mobile payment apps.

Bibliography

[1] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce component-level access control in Android. In *ACM CODASPY*, pages 25–36, 2014.

[2] apktool - A tool for reverse engineering Android apk files. http: //ibotpeaches.github.io/Apktool/.

[3] C-skills: yummy yummy, GingerBreak! http://c- skills.blogspot.kr/2011/ 04/yummy-yummy- gingerbreak.html.

[4] C-skills: Zimperlich sources. http://c- skills.blogspot.kr/2011/02/ zimperlich- sources.html.

[5] dex2jar - Tools to work with android .dex and java .class files. https: //github.com/pxb1988/dex2jar.

[6] T. Vidas, J. Tan, J. Nahata, C. Tan, N. Christin, and P. Tague. A5: Automated Analysis of Adversarial Android Applications. In *SPSM*, 2014.

[7] How secure the mobile payments are? https://storify.com/ williamjohn005/how- secure-the- mobile- payments- are.

[8] IDA Debugger. https://www.hex- rays.com/.

[9] Issuer Identification Number (IIN). http://publicaa.ansi.org/sites/apdl/ Documents/Other%20Services/Registration%20Programs/Important- Info.pdf.

[10] Java Decompiler. http://jd.benow.ca/.

[11] SEED algorithm specification. computer- tips- tricks/what- your- credit- card- numbers-mean/.

[12] P. Stirparo, I. N. Fovino, M. Taddeo, and I. Kounelis. In-memory credentials robbery on android phones. In *IEEE WorldCIS*, pages 88– 93, 2013.

[13] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *ISOC NDSS*, 2014.

[14] L. Onwuzurike and E. De Cristofaro. Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. *arXiv preprint arXiv:1505.00589*, 2015.

[15] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. Tapprints: your finger taps have fingerprints. In *ACM MobiSys*, pages 323–336, 2012.

[16] R. Boden. Mobile payment market worth $2.8tn by 2020. http://www.nfcworld.com/2015/05/27/335470/mobile-payment-market- worth- 2- 8bn- by-2020/.

[17] N. Bose. Retailer-backed mobile wallet to rival Apple Pay set for test. http://www.reuters.com/article/2015/08/12/us-currentc-mobile- payment-idUSKCN0QH1RY20150812.

[18] M. Cesaris. Number of Rooted Android Smartphones — Macro De Ce- saris — LinkedIn. https://www.linkedin.com/pulse/20140728070440- 13998576- number- of- rooted- android-smartphones.

[19] J. Chen, H. Chen, E. Bauman, Z. Lin, B. Zang, and H. Guan. You Shouldnt Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps. In *USENIX Security*, 2015.

[20] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE*, pages 175–186, 2014.

[21] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun. Smartphones as practical and secure location verification tokens for payments. In *ISOC NDSS*, 2014.

[22] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM CCS*, pages 73–84, 2013.

[23] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information- Flow Tracking System for Realtime Privacy Monitoring on Smart- phones. *ACM TOCS*, 32(2):5, 2014.

[24] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *ACM MobiSys*, pages 431–444, 2013.

[25] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith. Hey, NSA: Stay Away from my Market! Future Proofing App Markets against Powerful Attackers. In *ACM CCS*, pages 1143–1155, 2014.

[26] S.Fahl,M.Harbach,T.Muders,L.Baumga¨rtner,B.Freisleben,and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM CCS*, pages 50–61, 2012.

[27] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM CCS*, pages 38–49, 2012.

[28] J. Freeman. saurik/mempodroid. https://github.com/saurik/mempodroid.

[29] A. Harris, S. Goodman, and P. Traynor. Privacy and security concerns associated with mobile money applications in Africa. *Wash. JL Tech. & Arts*, 8:245, 2012.

[30] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. Butler. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applica- tions in the Developing World. In *USENIX Security*, 2015.

[31] J. Lee, L. Bauer, and M. L. Mazurek. The Effectiveness of Security Images in Internet Banking. *Internet Computing, IEEE*, 19(1):54–62, 2015.

[32] L.-K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security*, pages 569–584, 2012.

[33] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN*, pages 259–269, 2014.

[34] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *ACM SIGPLAN*, pages 1–6, 2014.

[35] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. UIPicker: User-Input Privacy

Identification in Mobile Applications. In *USENIX Security*, 2015.

[36] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *USENIX Security*, 2015.

[37] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security*, volume 2, page 2, 2011.

[38] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter- application communication in Android. In *ACM MobiSys*, pages 239– 252, 2011.

[39] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS*, pages 627–638, 2011.

[40] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security*, 2011.

[41] L. Onwuzurike and E. De Cristofaro. Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. *arXiv preprint arXiv:1505.00589*, 2015.

[42] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *SPSM*, pages 3–14, 2012.

요 약

모바일 결제 앱은 사용자의 민감한 개인 정보를 다루기 때문에 여러가지 보안 장치들을 사용한다. 우리는 한국 대부분의 모바일 결제 앱이 사용하는 보안 장치인 보안 키패드를 분석했다. 난독화가 심하게 되어있는 금융 앱을 분석하기 위해서 동적 리버싱 도구를 만들었다. 보안 키패드로 입력받은 사용자의 개인 정보를 추적하기 위해서 테인트드로이드(TaintDroid)를 확장했다. 스마트폰에서 사용자의 입력 값을 받을 때 사용되는 UI 적인 공통점 – 터치 이벤트, 텍스트 필드 – 을 테인트 소스로 제공하여 보안 키패드로 받은 사용자의 암호화 된 입력 값을 추적했다. 이 툴을 이용해서 분석 범위를 줄이고, 대상 앱들이 보안 키패드로 입력받은 암호화 된 데이터를 안전하게 다루는지를 분석했다. 그 결과 우리는 모든 조사 대상 앱들이 보안 키패드로 입력받은 사용자의 민감한 개인 정보를 복호화하는 것을 발견했다. 이는 보안 키패드 솔루션의 목적에 위배되는 것이기 때문에, 우리는 이 복호화가 꼭 필요한 것이었는지를 리버싱을 통해 자세히 분석했다. 우리는 분석 데이터를 기반으로 앱 개발자들이 보안 라이브러리를 더 안전하게 사용할 수 있도록 대안을 제시한다.

주요어: 보안키패드, 입력 데이터 추적, 암호화, 리버싱

학 번: 2014-21795