Journal of Information Security and Applications 48 (2019) 102358

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

Security analysis of third-party in-app payment in mobile applications

Wenbo Yang, Juanru Li, Yuanyuan Zhang, Dawu Gu*

Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

ARTICLE INFO

Article history:
Available online 23 July 2019

Keywords:
Mobile security
In-app payment
App vulnerability
Program analysis

ABSTRACT

The massive growth of smart mobile devices has attracted numerous apps to embed third-party in-app payment, which involves more sophisticated interactions between multiple participants compared to traditional payments. Therefore, such payment is error prone and could be exploited easily, leading to serious financial deceptions. To investigate current third-party mobile payment ecosystem and find potential security threats, we conduct an in-depth analysis against China–world's largest mobile payment market. We study four mainstream third-party mobile payment cashiers, and conclude unified process models. We also summarize the security rules that must be regulated by cashiers and merchants and illustrate four types of attacks if violating these rules. Besides, we also detect seven cases of security rule violation on both Android and iOS platform. Our detection result shows that hundreds of popular apps violate at least one security rule, and hence face with various security risks, allowing attackers to consume commodities or services without purchasing them or deceiving others to pay for them. Our further investigation reveals that cashiers as well as merchants should be responsible for those vulnerable cases. We also performed proof-of-concept attacks in real world, reported these issues to all involved parties and helped them fix the vulnerabilities.

1. Introduction

The past few years has witnessed the extraordinary development in mobile payment. The significant growth of smartphone promotes the usage of third-party mobile payment services in mobile apps. Compared to transaction processes with traditional payment channels (e.g., via credit card), transactions with third-party in-app payment are settled within mobile app conveniently. Users can pay their bills directly without switching to another app or web browser. To help apps use their mobile payment services, third-party cashiers are willing to provide such functionality for popular Android and iOS apps to fulfill in-app payment. The cashiers provide third-party payment SDKs (TP-SDKs) to mobile app developers, leading a straightforward integration of in-app mobile payment. As a result, more and more apps are using third-party in-app payment as their major payment channel.

Nonetheless, implementing a secure in-app payment is not easy. In-app payment is still in its incipient stage and is especially error-prone due to many issues such as the misunderstanding of app developers, the improperly designed services, and the ambiguous documents or code samples released by cashiers. In addition, a process of third-party in-app payment involves more participants (user, cashier, and merchant), and the interaction steps compared to traditional payment processes is more sophisticated. Therefore, the potential attack surfaces for third-party in-app payment are much wider.

Previous studies on payment security [1–5] mainly focus on the security of e-commerce web applications other than mobile apps. Although numerous security flaws of e-commerce web applications integrating services of third-party cashiers had been revealed, on mobile platform the trust boundaries are redefined and the problems should be re-studied. The workflow of an entire in-app payment transaction is much more complex and the security analysis of e-commerce in web applications is unable to cover some new steps. Particularly, since the introduced mobile client plays an important role in this multi-party model, it is inadequate to directly employ traditional flaw detection of web applications on mobile apps. Client apps are generally considered untrustful since all the data handled by apps can be manipulated by an attacker with a rooted Android phone or a jailbroken iOS device, Hence, a comprehensive security analysis against third-party mobile payment must be re-designed.

To the best of our knowledge, there exists neither unified specification to regulate in-app payment process nor assessment approach to validate the security of them so far. Documents and sam-

* Corresponding author.
E-mail addresses: wbyang@securitygossip.com (W. Yang), jarod@sjtu.edu.cn (J. Li), yyjess@sjtu.edu.cn (Y. Zhang), dwgu@sjtu.edu.cn (D. Gu).

https://doi.org/10.1016/j.jisa.2019.102358
2214-2126/

ples of transaction provided by most in-app payment cashiers are insufficient or even incorrect. Analysts must reverse-engineer the binary code of an app and its integrated in-app payment SDKs to recover and assess the process. Therefore, it is expected to first summarize the status quo of current third-party in-app payment before searching potential security flaws. After that, the security analysis requires to conclude the basic security rules which should be obeyed by both cashiers and merchants throughout the transaction process as well as proposing corresponding violation detection methodology.

To investigate how widespread is insecure in-app payment in both Android and iOS apps, in this paper, we make a systematic study of the world's largest smartphone and mobile payment market–China. A distinguishing feature of Chinese mobile payment market is that most apps in China *only* support third-party in-app payments. User cannot use other payment channels such as credit card or online bank to pay the bills. On the other hand, users in China also prefer to choose these third-party cashiers to manage their payments because these cashiers provide more services than traditional banks. With the help of those cashiers, users not only purchase goods or make investment conveniently, but also transfer to each other through their accounts for free. Nonetheless, the prevalence of third-party in-app payment also brings significant risks. Once these third-party payments are vulnerable, every in-app transaction is inevitably suffering severe security threats.

Our study tries to answer the following questions for our research targets: (a) What exactly should be done to implement a secure in-app payment? (b) Which kind of attack can be conducted and who suffers financial loss? (c) What is the quantity of app with insecure in-app payment and how to detect them? (d) What factors lead to an insecure implementation? To answer the above questions, we first conduct an in-depth analysis on services of four mainstream third-party cashiers. The results of our analysis include: (1) we unveil the details of their payment processes through reverse engineering of relevant SDKs and apps; (2) we conclude a series of security rules which satisfy the security requirements of in-app payment; (3) we illustrate severe consequences of violating these rules including four attacks against both online and offline transactions.

After the investigation into the services of mobile payment cashiers, we then develop a methodology to detect seven representative violations of our proposed security rules in Android and iOS apps, their integrated third-party payment SDKs, as well as servers of merchants and cashiers. We prove the prevalence of in-app payment through scanning 7145/10,000 Android/iOS apps and pinpointing at least one in-app payment SDK in above one-third of the analyzed apps. We then detect 2679/3972 Android/iOS apps with payment and find hundreds of them vulnerable. In addition, none of the four cashier's SDKs reach the requirement regulated by our proposed security rules. Even though the improperly designed SDKs do not directly lead to vulnerabilities, they would expand the effect of user deception attack. Through combining these flaws, various attacks can be constructed targeting both merchants and users, including shopping for free or with another user's account. As a result, these flaws affect almost all aspects of daily life.

Finally, we investigated the root cause of the flawed implementations of in-app payments. To our surprise, we found the documents and sample codes provided by cashiers are often not carefully examined and confuse developers. Some of them are incorrect and even vulnerable, which directly lead merchants to these flaws. Different from previous works [1–4] that focus on the errors of merchants, our findings reveal that the cashiers also contribute to the flawed third-party payment on mobile platform. We have reported all the problems to the affected cashiers and obtained their credits.

2. Third-party in-app payment demystified

It is prevalent for modern apps to build their in-app payment functionality with the help of a third-party payment service provider. On the one hand, The costs would be too high for merchant to construct its own payment system. On the other hand, it is also unrealistic for users to create and manage their capital accounts in each app respectively.

Although in-app payment is pervasive on Android and iOS, the process of how an app fulfills a transaction via third-party payment service is often obscure due to several reasons. First, implementation variation of in-app payment is significant. Different third-party payment service providers (cashiers) regulate different in-app payment processes and release their own SDKs for app to integrate. Implementation aspects of third-party payment services such as used web APIs, integration style of SDKs, and the parameters required differ greatly from each other. Second, cashiers often release documents and samples to app developers. However, our review illustrates that most of these documents are ambiguous and may confuse the app developers. Some code samples even conflict with the process regulated by the documents. Only by reading cashiers' documents and studying their sample code is not adequate to conclude the exact payment process, and we will give our result in Section 5.5. Third, testing the in-app payment not only involves actual payment with money expending, but also requires some franchises and relevant documents only granted to verified identity such as registered companies. Many analysis efforts lack such qualifications and therefore are impeded.

To demystify the details of in-app payment process, we first give a brief description of participants involved in payment process. Then, we choose four popular cashiers: WexPay (in-app payment service provided by Wechat Wallet) [6], AliPay (Alipay Wallet) [7], UniPay (Unionpay Wallet) [8] and BadPay (Baidu Wallet) [9] and analyze their documents and code samples. We also reverse-engineer popular apps with in-app payment to understand the details of payment implementation. After this reverse engineering work, we gain a panoramic view of in-app payment process: two representative payment process models that cover necessary transaction steps for four cashiers are concluded.

2.1. Definitions

In a typical third-party in-app payment process, user browses, selects and buys commodities in a merchant app (*MA*). Implemented by the merchant, the *MA* and the merchant server (*MS*) interact with each other. Information such as users information, commodities provided, and order information are stored in databases on the *MS*.

To support payment in app, an *MA* integrates one or more third-party payment SDK (TP-SDK) released by the third-party cashier. In a checkout process, user chooses a third-party cashier in the *MA* and makes a payment to the cashier. The cashier server (*CS*) records the payment information and status, and informs the merchant the completion of the payment. The complete payment information is then stored to the *CS*.

Since cashier is the third party between user and merchant, the detail of the merchant order (e.g., unit price about the commodities) is not the necessary information for cashiers due to the privacy issue. Therefore, Apple's IAP (In App Purchase) is not a typical third-party payment. Apple, as a cashier, requires merchants to register all the content and its price on its website. Merchants need to offer their merchandises through the cashier rather than by themselves throughout the transaction process. Thus, only the cashier is actually involved in the Apple's IAP payment process. In addition, the strict restriction on the goods type regulated by Ap-

ple's IAP further limits its application scenario. As a result, Apple's IAP is not included in our work.

2.2. Unveiling payment process

To unveil the payment process, we conducted a systematic study against mobile payments in mainland Chinese market as our research target for the following reasons: First, it is the world's largest smartphone and mobile payment market: about 890 million users in this market use mobile devices to purchase goods and services (by the end of 2018, according to research from the China Internet Network Information Center). In the year of 2018, China's third-party mobile payment tools handled transactions worth more than 200 trillion yuan (29.4 trillion us dollar), much more than any other countries in the world. Second, unlike the mobile payment market in the U.S., where most mobile transaction is settled via credit card through web or Apple Pay, a large portion of apps use third-party in-app payment services in China. Third, instead of a single payment standard, most apps adopted a variety of payment schemes provided by different third-party cashiers simultaneously. Due to its sophisticated characteristic, mobile payments in mainland Chinese market is a very representative target and is worth being studied.

Another important fact for the mobile ecosystem of mainland China is that the official app store of Google (i.e., Google Play Store) is unavailable. As a result, many Chinese companies or developers chose to publish their apps on domestic app markets. Therefore, we also collected our app samples from these markets. We collected Android apps from *Myapp* [10], the largest Android app store in China [11]. This market also provides strict 'official' certification service, which requires publishers to submit a series of materials to prove their copyrights [12]. In addition, to collect iOS apps we have to address the encryption issue of the Apple's official App Store (all apps downloaded from the App Store are encrypted and the analysis of such an app needs a decryption first). Fortunately, we found *25PP*, a third-party iOS app market[1] have collected decrypted version of most popular iOS apps. We therefore crawled this market to obtain the samples needed.

Note that in our study we did not download all apps in both Android and iOS app markets. The main reason is that only apps with larger number of users and transactions would integrate mobile payment services. We thus only chose those apps with either large download numbers or a relatively high impact. In particular, we download 7145 most popular Android apps with at least 100,000 users for each from *Myapp*, and 10,000 most downloaded iOS apps from *25PP*. We argue that these apps are prevalent and representative samples for our study.

We choose four popular cashiers as our research targets: WexPay, AliPay, UniPay and BadPay. Each of them has at least 100 million users. Merchant can register to all four cashiers on two mobile platforms (Android and iOS) as long as it owns a legitimated company registered to the Chinese Commerce and Industry Bureau. For every cashier, we get the TP-SDK and auxiliary materials of Android and iOS including code samples and relevant documents. The documents describe not only interfaces of TP-SDK but also the suggested payment process and Web APIs of cashier server. Code samples illustrate simplified implementation for client app and server. Through studying the documents of four cashiers, reverse-engineering TP-SDKs and downloaded apps with static and dynamic analysis, monitoring the network traffic of the transaction process, and implementing sample code to real app and server, we have two observations about the in-app payment: (1) how prevalent is the third-party in-app payment on mobile platform; (2) which payment process model does merchant need to comply with when integrating third-party in-app payment function. We detail these observations in the following sections.

2.3. TP-SDK identification

In order to find out which app uses third-party in-app payment, we adopt feature based identification strategy to detect apps with TP-SDK. We reverse-engineer TP-SDKs of four cashiers on two platforms and extract their unique features. We observe that if an *MA* uses a TP-SDK, it needs to invoke a specific interface and passes parameters, hence we make use of these interfaces as the feature of TP-SDKs. For instance, if an Android *MA* uses TP-SDK of AliPay, it must pass the payment order information to AliPay SDK through a certain interface.[2] And for iOS apps, Alipay payment SDK also provide such interface.[3] For other three TP-SDKs, there are also similar features. Note that the name of interface is not always an available feature. Developers may use code obfuscation tools such as *ProGuard* [14]/ *ios-class-guard* [15] to obfuscate the function names in Android/iOS apps. Therefore, we manually pick a combination of special strings in every TP-SDK that are seldom used elsewhere as extra features. For instance, we find that WexPay uses specific strings,[4] to label transaction data of payment request in its SDK, which are unique and can hardly be found in other SDKs. Similarly, UniPay always has such strings as[5] in its SDK. Utilizing those features, we build static analysis tools to scan apps of Android and iOS respectively. Our tool is based on *AndroGuard* [16] for Android apps and *Radare2* [17] for iOS apps. The result is listed in Section 5.

2.4. Process analysis

After studying the documents and sample code of four cashiers, we registered as a merchant to cashiers and developed proof-of-concept apps and corresponding servers to better understand the payment processes adopted by the four cashiers. Besides, we even downloaded and analyzed several popular merchant apps and their servers statically and dynamically. We reverse-engineered the client apps (with IDA [18] for iOS apps and Android native code, with JEB [19] for Android apps), monitored the network traffic between apps and servers (with Burp Suite [20] for HTTP/HTTPS message, and Wireshark [21] for TCP/UDP-based protocol), hooked specific methods or functions to retrieve/modify runtime data of merchant apps (we have developed Xposed [22] and Frida [23] plugins for Android and iOS apps, respectively), and constructed customized network requests to cashiers and merchant servers.

We find that although the whole payment process of four cashiers are somewhat different, they can be concluded as two in-app payment process models (in Figs. 1 and 2). Both platforms (Android and iOS) of one cashier adopt the same model. Among the four cashiers, WexPay and UniPay follow the process model I (in Fig. 1), while AliPay and BadPay follow the other. We first choose process model I as an example to illustrate a complete third-party in-app payment process in detail. It is a simplified model only including essential steps and parameters of a transaction. The whole process of the model contains nine steps in general.

1. The *MS* receives a merchant order ($order_m$) and the type of cashier after a user selects the commodities and chooses a third-party cashier in the *MA*. $order_m$ contains order information only related to merchant (e.g., the type and the amount of

[1] (a.k.a. PP Assistant) [13], the largest third-party iOS app store in China.

[2] *com.alipay.sdk.app.PayTask- > pay()*

[3] *AlipaySDK payOrder:fromScheme:callback:*

[4] *_wxapi_payreq_appid, _wxapi_payreq_partnerid*, etc.

[5] *uppayuri, com.unionpay.uppay*, etc.

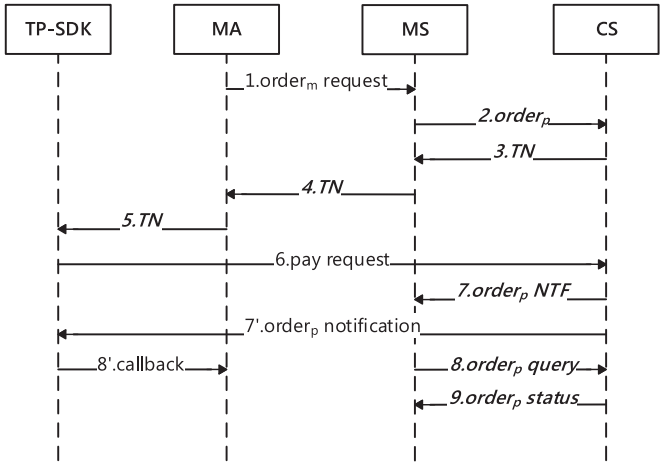


Fig. 1. In-app payment process model I adopted by WexPay and UniPay.

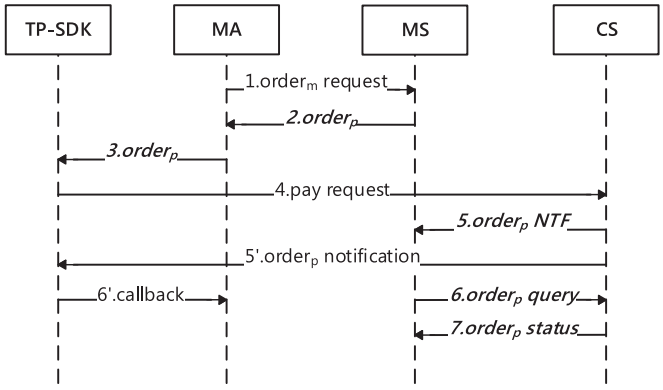


Fig. 2. In-app payment process model II adopted by AliPay and BadPay.

commodities user wants to buy), since cashier has not involved yet by now.

2. The *MS* generates a payment order ($order_p$) according to $order_m$, and sends it to the cashier by invoking the Web API defined. The $order_p$ should contain information about this payment, (e.g., *order ID, merchant ID, total amount, notify URL address*).
3. After receiving and verifying the $order_p$, the *CS* will store payment information into its database, and returns a signed message (*TN*) that contains a *transaction number*. Note that the *transaction number* identifies the payment order and is generated by the cashier.
4. After the *MS* receives and verifies *TN, MS* should sign *TN* and send it to *MA*. Note that *TN* now contains the merchant's signature.
5. *MA* deals with received *TN*, and passes it as parameters to the interface defined in TP-SDK.
6. TP-SDK prompts its payment UI to accept user's confirmation. The payment UI in TP-SDK shows the detailed information of the payment order acquired from the *CS* through its own network channel (omitted in the Figure). After user confirms the payment order and enters the account password, TP-SDK sends the pay request to the *CS*. The *CS* checks the request, and then pays for the order with money in user's account.
7. The *CS* sends a notification of payment to both TP-SDK (the step with *apostrophe*) and the *notify URL* of *MS* (the step without *apostrophe*).
8. The *MA* shows payment result to user according to the notification received by TP-SDK.
9. The *MS* validates the signature of the notification, and makes an extra query of the notified payment order to the *CS* to confirm details of the order including *order ID, merchant ID, total amount, etc.*

After all the above steps, the transaction is settled and the merchant can ship commodities or provide services to user.

When adopting process model I, WexPay and UniPay implement similar process with nuance differences. Both cashiers require different extra parameters for $order_p$ and $order_m$. Also, UniPay does not require the *TN* message to be signed and does not include Step 8 (in Fig. 1) as a necessary step in its suggested process.

When adopting process model II, however, AliPay and BadPay have relatively larger differences to WexPay and UniPay. The main difference occurs in Step 2. The *MS* just sends the generated signed payment order ($order_p$) back to the *MA* other than to the *CS* after receiving the merchant order ($order_m$) request from the *MA*. Compared with model I, in which the *MA* can only receive *TN*, the *MA* in Fig. 2 receives the complete payment order information including order ID, total amount of the payment, the notify URL address of the *MS*, etc. And it transfers all the information to the integrated TP-SDK, which is responsible for dealing with all the detailed parameters of the payment order in this process.

In Figs. 1 and 2, messages with bold and italics text need to be *signed* by the sender to prevent being tampered. So another important factor in the transaction process is the signing method of messages adopted by cashiers. AliPay and UniPay regulate the *SHA1-RSA* as their signing method. Merchant generates its RSA key pair, and sends the public key to the cashier. Also, cashier informs every merchant its public key. The *MS* verifies the received signed message with the cashier's public key, and sends message signed with its private key to cashier or to the *MA*. However, WexPay and BadPay adopt hash function (e.g., MD5) with a secret key (as the *salt* of the hash function) to generate the signature. The *secret key* is shared between the merchant and the cashier. In the later part of this paper, we denote both the secret key of hash function and the merchant's RSA private key as *KEY*.

3. Security analysis

In this section, we describe the conducted security analysis against the process models we concluded above. The security of third-party payment has been studied before in previous works [1,2,4,5]. However, all of them focus on web service. In the prevailing mobile platform, the *in-app* payment introduces new multi-party models and thus, faces new security challenge. The merchant client application and the embedded TP-SDK play more significant roles which do not exist in traditional web model. So it's necessary to re-consider the security threats of the in-app payment on mobile platform.

Although the payment process models that regulated by cashiers have been vetted before releasing and are supposed to be secure, such multi-party models still struggle against various unexpected security threats due to the information asymmetry in the transaction process. Moreover, the whole transaction process involves multiple parties including not only cashiers but also merchants and users. Due to the ambiguous documents and confusing sample code released by cashiers, developers of merchants often disobey the process model regulated by cashiers and implement diversified payment processes, which may lead to potential security flaws. Any mistake committed by any party in the multi-party model may lead to a vulnerable process. Therefore, it's necessary to conclude security rules to regulate all parties in the model.

In the following, we first describe the adversary model, and then define the security rules that a secure in-app payment must comply. In addition, we clarify what the cashier and the merchant

should pay extra attention to throughout the entire transaction process of in-app payment. Finally, we describe four concrete attacks in detail under our reasonable adversary model if the cashier or merchant violates the security rules, which may lead to the loss of multiple participants in the model.

3.1. Adversary model

Before discussing security rules must be followed by third-party payments and corresponding attacks, we first define the adversary model as follows:

We assume that an attacker can always reverse-engineer an *MA* and the embedded TP-SDKs, since the app can be easily acquired from both Android and iOS app markets (even if the app is protected, techniques have already developed to circumvent it [24]). Even though the attacker is not able to sniff or tamper the network traffic between the *MS* and the *CS* under any circumstances, he can forge a request or a message to either the *MS* or the *CS* in our model, because such destination URL is not difficult to be obtained (by reverse-engineering apps or reading cashiers' documents).

When the attack targets a cashier or a merchant, the attacker plays as a malicious user and tries to get profit (e.g., acquiring secret information from merchant/cashier or even purchasing things without paying) from either the merchant or the cashier. In this case, the attacker is assumed to use his own smartphone to do the shopping, which indicates that he can arbitrarily modify the mobile system (e.g., rooting the Android phone, jailbreaking the iOS device, debugging apps, etc) and thus manipulate the execution and data of both the merchant app and the embedded TP-SDKs.

When the attack involves other users of the *MA*, the attacker aims to fraud other users (e.g., deceiving other users to pay). In this case, the attacker is not able to control other users' devices, i.e., not able to install malicious apps or repackaged *MA* on victim's phone by subterfuge. However, the attacker is assumed to control the data transmission between them (e.g., conducting an MITM attack with the ARP spoofing or deceiving users to attacker's malicious Wi-Fi). Signature-based or anomaly-based IDS for smartphones [25] may reduce the possibility of users being phished in this case. However, it is not common to deploy an IDS on a real world smartphone, and an IDS cannot protect merchants or cashiers.

3.2. Security rules

According to the two types of process model adopted by four cashiers and the adversary model, we conclude the following security rules that must be obeyed throughout the whole process involving both cashiers and merchants, no matter how cashier regulates the process model or which cashiers *MA* chooses to use. Otherwise, the process will be breached.

1. Payment orders must be generated (Fig. 1) or signed (Fig. 2) by the *MS* only.
2. *Never* place any secret (e.g., private key for signing) in the *MA*.
3. TP-SDK must inform user *detailed* information of the payment order.
4. TP-SDK must verify the transaction belonging to the *MA*.
5. *Always* use secure network communication between client and server.
6. *MS* should make an *extra* query to confirm notified payment's details.
7. *Always* verify the signature of received messages.

We conclude the above security rules by considering both the security requirements of a secure mobile payment mentioned in official documents of SDKs, and the best practices of building a robust online payment system [1]. Particularly, our security rules are comprehensive and consider not only multiple parties such as *MA*, TP-SDK, *MS*, but also the connection between them. To the best of our knowledge, our proposed security rules are the very first ones for third-party in-app payments.

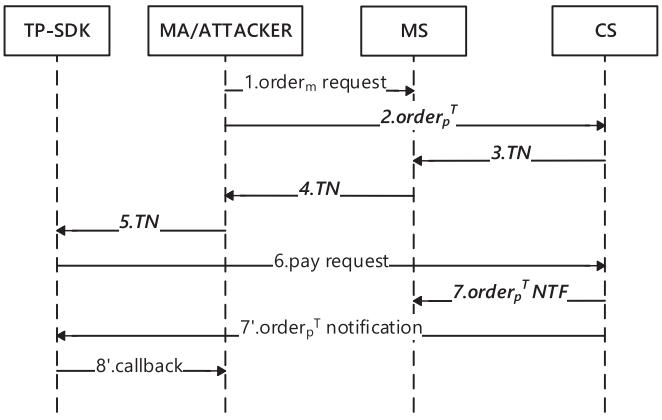


Fig. 3. Order Tampering Attack to Process Model I.

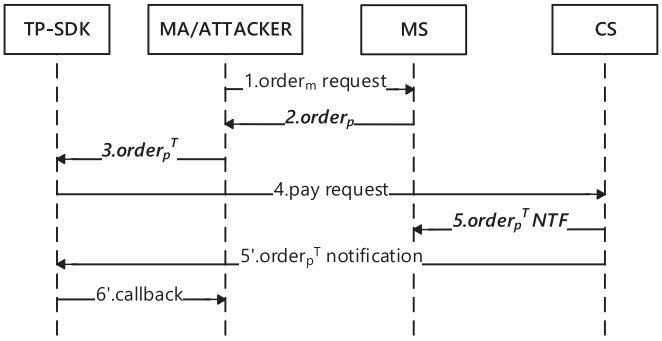


Fig. 4. Order Tampering Attack to Process Model II.

There are four types of attack that the payment process may suffer if one or more violation of security rules occur, and the victims involve normal users of *MA* and the merchant. Then we will describe them in details.

3.3. Order Tampering

In this type of attack, the attacker acts as a malicious user. If the merchant fails to obey the *Security Rule 1* and *Rule 6*, then attacker can cheat the merchant by sending a payment order ($order_p$) to the cashier different from the actual merchant order ($order_m$). In this situation, the attacker could tamper the content in the payment order such as the total amount and thus pay less money for the ordered commodities without the merchant's awareness.

The attack for process model I is shown in Fig. 3. In model I (Fig. 1), the signed payment order is generated and sent by the *MS* to the *CS*. A local attacker can only obtain the (*TN*) message which does not include any detailed information (e.g., the total amount of the payment) of the payment order. Thus, it's impossible to tamper the payment order information. However, if the *MA* incorrectly implement the payment order generation step in the app rather than its server, the attacker can succeed in tampering the payment information. Since the attacker can take full control of local app and system, we merge the attacker with the *MA* in the Fig. 3. The $order_p^T$ in the figure indicates that the payment order has been tampered already and so does Fig. 4.

In model II (Fig. 2), though the complete payment order information can be achieved by the attacker ($order_p$ need to be returned to *MA*), he can not tamper it since the payment order is signed by

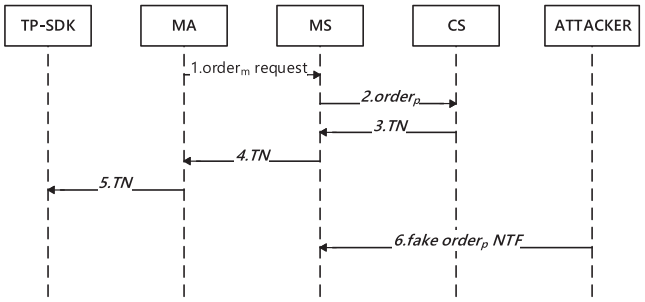


Fig. 5. Notification Forging Attack to process model I.

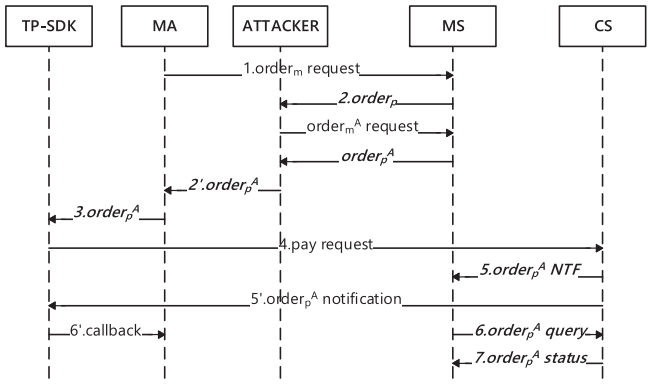


Fig. 6. Order Substituting Attack to process model II.

MS. A payment order with wrong signature will be rejected by the cashier. However, if the merchant signs the payment order or leaks the *KEY* in the *MA*, then the attacker could easily intercept and tamper the received order information (e.g., modifying the price) in the *MA*, and re-signing it as a legit one to TP-SDK (as shown in Fig. 4).

To fulfill this attack, another implementation flaw is required: the merchant fails to confirm the notified payment order information to ***CS*** (*Security Rule 6*). Otherwise, *MS* can get every details of the notified payment order including total amount, merchant ID, etc after make the query to *CS*. It can refuse the tampered order after verifying every details of it and does not ship commodities.

3.4. Notification forging

If the merchant fails to obey the *Security Rule 2* (or *Rule 7*), and *Security Rule 6*, then it suffers notification forging attack, allowing attackers to purchase commodity without paying it. In the attack, a normal payment process is performed until the TP-SDK requires user to confirm the order and enter password to pay for it. At that time, an attacker does not pay for it, but instead sends a fake payment result notification to notify the *MS* that the order is paid successfully. The attack to process model I (Fig. 1), for example, is shown in Fig. 5. If the order is not paid (Step 6 in Fig. 1), it still remains '*pending*' status, and the *MS* will not receive the notification from the *CS* (Step 7 in Fig. 1). However, afterwards attackers can forge the notification and send it to the *MS* (Step 6 in Fig. 5). If the merchant trusts this fake notification and does not confirm the order's details to the *CS* (Step 8 in Fig. 1), the payment is successfully forged. The attack can also be performed to model II (Fig. 2) with the same way, which we omit it here.

Attackers need to exploit several mistakes committed by the merchant to make a forged notification available. First, the notify URL address of *MS* that receives the payment notification from *CS* should be known beforehand. So the attack requires *MA* to contain the notify URL address, which would be placed by the developers accidently. Actually, as we illustrated before, *MA* who adopts process model II (AliPay and BadPay) certainly contains the notify URL, since all order information including notify URL is used as input of the TP-SDK (Step 3 in Fig. 2). Second, the attack needs to construct a forged payment order notification of the cashier and cheats *MS* to accept it. Attackers can obtain the data format of the notification message from documents released by the cashier, and then forge it with a signature which proves the identity of the sender. The *KEY* used here is often extracted from the *MA*, in which the merchant's developers place this shared secret key by mistake (*Security Rule 2*). Note that among four cashiers, only the notification of those who adopt hash-function as their signing method (WexPay and BadPay) can be forged because the cashier and the merchant share the same *KEY* as their signing *KEY*. For those using *SHA1-RSA* to sign the messages, the RSA private key of cashiers can be hardly leaked, thus, forging the cashier's message with legal signature is quite impossible. Moreover, we observe that some *MS*s even ignore validating the signature of the received messages (*Security Rule 7*). Thus, the fake notification even with wrong signature is unconditionally accepted. Finally, similar to *Order Tampering* Attack, notification forging also needs the *MS* to ignore the order re-confirmation step. Otherwise the merchant can find out that the notified payment order is still remain 'pending' status in *CS*.

3.5. Order substituting

Different from the two attacks above, the victim of order substituting attack becomes the normal user of *MA* rather than the merchant. The cause of the attack involves multi-parties' violation of security rules including both cashier (*Security Rule 3* and *4*) and merchant (*Security Rule 5*). In this attack, the attacker substitutes an order of one transaction to another, and misleads a victim user to pay for the attacker's order unconsciously.

Fig. 6 shows the order substituting attack to process model II (Fig. 2). The attack is available when the message returned from *MS* is transferred with an insecure network communication channel. Thus, the attacker can act as a man-in-the-middle between *MA* and *MS*. Attackers can intercept the message and substitute signed payment order ($order_p$) with another one ($order_p^A$) of a legal transaction, and send it to the *MA* on victim's device. The victim will then pay for the attacker's order rather than his own order. Note that the attacker uses a legal payment order to replace the original one. This message usually belongs to a normal trade performed by the attacker beforehand (steps between Step 2 and Step2′ in Fig. 6), so it is reasonable to cheat the victim's TP-SDK and finish the transaction with this message successfully. The attack to process model I (Fig. 1) is similar. The only difference is that the attacker needs to substitute the *TN* message (Step 4 in Fig. 1) rather than the $order_p$ (Step 2 in Fig. 2) returned from *MS*.

The root cause of this attack includes the lack of secure communication channel as well as the inadequate prompt information showed by TP-SDK. We discover that the payment UI (asking for user's confirmation before user paying) of TP-SDK generally does not show enough information about current payment order, thus the victim will confirm and pay for another order without being aware of it. For example, if the payment UI only shows the total amount of the order, then the attacker could make an order with the same price of the victim's order. Even if some TP-SDKs show the commodities and the merchant name of the order, the attacker could make an order with same commodities while modifying the consignee since it is not difficult for attacker to know what victim is going to buy through eavesdropping the merchant order request (Step 1 in Fig. 6) via insecure network connection between the *MA* and the *MS*. What's worse, if the TP-SDK accepts the $order_p$ (or *TN*), whatever it is generated by the host *MA* or not, this attack can be

expanded that even a transaction from another *MA* can be substituted to that from one *MA*. In other words, if attackers substitute the original $order_p$ (or *TN*) with anther $order_p$ (or *TN*) of malicious merchant registered to cashier by attackers themselves, the money paid for the transaction will be transferred to attackers directly. Nevertheless, our investigation indicates that some TP-SDKs do not verify *TN* carefully, allowing attackers to substitute the original one easily.

3.6. Unauthorized querying

If the merchant violates the *Security Rule 2*, leaking its *KEY* to attackers, it will also suffer the unauthorized querying attack. An unauthorized querying attack allows attacker to query the details of every transaction recorded in *CS*, acquiring secret business information which should only be shared by cashier and merchant. The root cause of this attack is due to the leaking of merchant's authentication credential. Cashiers provide several Web APIs for merchant to query various information, such as every payment order's status and details, the merchant's history bill of everyday, etc. Furthermore, cashiers make use of the signing *KEY* to authenticate the identity of each merchant. However, the *KEY* may be accidently placed in the *MA* by the developers of the merchant. So the attacker could utilize the leaked *KEY* to query transaction information illegally.

4. Detecting flawed in-app payments

The violation of the seven security rules causes exploitable attacks and leads to serious consequences. In this section, we will describe how to convert these rules into detectable forms in the payment process both on Android and iOS platform. Detecting these violations is helpful to find flawed in-app payments to actual loss. Furthermore, we discuss the feasibilities and details of detecting such flaws.

4.1. Local Ordering

According to the *Security Rule 1, MA* is prohibited to generate payment orders for those adopting process model I. Local Ordering refers to the incorrect ordering behavior implemented by the *MA* rather than the *MS*. It allows the attackers to tamper the payment order. Note that this flaw only appears to apps with WexPay or UniPay, since in their regulations, placing the payment order must be enforced by the *MS*.

To detect this violation of *Rule 1*, we search the existence of a relevant *destination URL* used by the merchant to place a payment order. In detail, app will make request to specific URL[6] for WexPay and UniPay, respectively. The request indicates the incorrect behavior of generating order locally. Therefore, we scan all strings in DEX file and resource file of an APK for Android to find whether the above two strings exist. For iOS apps, we search the strings in the *.ASM* file generated by IDA batch mode. Once the URLs exist, the app is then manually tested to confirm the security flaw.

4.2. KEY leakage

In the two payment process models, several messages transmitted need to be signed. According to our proposed *Security Rule 2*, sensitive information, especially the *KEY* is prohibited to appear in app. Otherwise, attackers can tamper or forge messages with legal signature and camouflage to be certain party and cheat others in the multi-party model.

We combine pattern matching and dynamic testing techniques to detect *KEY* leakage in apps.

- *WexPay*. A hash function with secret key to generate the message signature is adopted by WexPay. The secret key for message signing is a 32-byte string with arbitrary content shared with merchant and cashier. The *MA* uses this key to sign message so we would like to search such hard-coded key in app. However, simply searching 32 bytes length string in an *MA* often gives a huge amount of candidates. To effectively determine the potential key, we utilize a Web API provided by WexPay as an oracle to substantiate the key identity accurately. For Android apps, we make use of the Web API which allows merchant to download the history bill of one day with three necessary parameters: *appid, mch_id*, and *sign* (generated by *secret key*). Therefore, we could leverage the *appid* and the *mch_id* to help identify the *secret key*. Note that the features of these two parameters are apparent: the *appid* is a 18-byte string with a *wx* prefix, and the *mch_id* is a 10-byte string comprised of digits only, and both two parameters are uniquely allocated to merchant. We can first locate strings with similar features in DEX file and resource file (strings.xml), and query the Web API for the identity of the found parameters. If any of the input parameter is incorrect, the response of the query gives a corresponding notification. For instance, if the first *appid* parameter is incorrect, the Web API would directly return a "*wrong appid*" notification without considering the following parameters. Thus we could check each parameter individually until its correctness is identified, which significantly improve the efficiency. And if all three parameters are correct (which means we find a leaked key in app), the Web API responds either the merchant's real bill data, or "*no bill exists*" if no transaction happened on that day. Using this testing approach, we can effectively find leaked WexPay key in an app.
  For iOS apps, we use similar methodology that also relies on a Web API. Since WexPay had deprecated the *downloadbill* API after we reported our findings (Section 6), we choose another web API—the placing order API[7] which should be used by *MS* to generate payment order to WexPay. We fill in the *appid* and *mch_id* with the candidate strings in apps and other parameters such as *total_fee, out_trade_no* with reasonable value, and generate *sign* using suspicious *secret key*. if all parameters are correct, WexPay will response a message containing a *prepay_id*, which means a unique payment order related to the merchant has already generated, and waiting for user payment. If any of the parameter is incorrect, the message will be incorrect appid (APPID_NOT_EXIST), incorrect mch_id (MCHID_NOT_EXIST), or invalid signature (SIGNERROR).
- *BadPay*. Similar to WexPay, BadPay uses a shared secret key to sign its messages. However, no Web API is provided by BadPay for us to verify the potential key candidates. Considering that far fewer *MA*s use BadPay, we could confirm the key through manual reverse engineering.
- *AliPay*. For AliPay, merchant uses an RSA private key within a Based64-encoded standard ASN1 certificate to sign the order information. The certificate format contains remarkable feature (A string with *'MI'* as prefix and at least 300 bytes long) and can be easily located. However, the app may also contain such certificates to fulfill other functionalities. To confirm the application of found certificates, we adopt the following two heuristics for Android apps. We first check whether the variable name of the candidate certificate contains *ali* or *alipay*. Second, we make use of the cross reference searching to find the Java class that refers to the candidate certificate. Since the private key in

[6] https://www.api.mch.weixin.qq.com/pay/unifiedorder for WexPay; https://gateway.95516.com/gateway/api/appTransReq.do for UniPay.

[7] https://api.mch.weixin.qq.com/pay/unifiedorder.

a certificate is used to sign the order of AliPay, the order information is often generated in the same class that uses the private key. This generated order information contains specific feature strings ("*&service=mobile.securitypay.pay*" for example) and can be easily identified. If a certificate corresponds to one of the above properties, we regard it as the signing key of AliPay. For iOS apps, we build a tool based on IDA Pro similar to the heuristics used for Android apps. In addition, we also find a *downloadbill* Web API provided by AliPay, that could be used as an oracle like WexPay, to confirm the *KEY* accurately. However, it could only be applied to apps using the latest version of AliPay API, which is a very small portion of our samples.

- *UniPay*. Similar to AliPay, UniPay also uses RSA private key to sign its messages but the private key is encapsulated in a CER format. We also adopt similar detection methodology to UniPay.

4.3. Incomplete prompt

When an *MA* invokes the TP-SDK and shows the payment UI to users (e.g., between Steps 5 and 6 in Fig. 1), users need to confirm the order and decide whether to pay for it. As the *Security Rule 3* implies, detailed order information should be prompted to user in the payment UI completely. Otherwise, user may suffer deception, resulting in an attack that what user pays is not what he/she really buys (*Order Substituting Attack*).

We detect this security flaw by checking whether TP-SDKs (for both Android and iOS) display enough information about the payment order to user during the payment. In detail, the following fields are checked: (1) payment order ID that represents the order uniquely in both merchant and cashier. (2) what commodity or service that users are going to pay. (3) user that the order belong to in merchant app. (4) merchant that the order belongs to. (5) total money of the payment.

4.4. Transaction verification missing

In a secure payment process, TP-SDK integrated in *MA* need ensure that the received payment order (Step 3 in Fig. 2) or *TN* (Step 5 in Fig. 1) actually belongs to the *MA* according to *Security Rule 4*. Otherwise, malicious merchant can expand the *Order Substituting* Attack and directly get money from users as we mentioned in Section 3.5.

We detect this security flaw on both Android and iOS platform, through testing whether the TP-SDK accepts a payment order that does not belong to the *MA*. First, we place an order using a normal *MA*. Then we intercept the $order_p$ / *TN* message from the *MS* and substitute it with $order_p$ / *TN* message generated from another *MA*. And we check whether the order belonging to another *MA* can be accepted successfully by the TP-SDK. If so, the violation of *Rule 4* is confirmed.

4.5. Insecure communication

According to *Security Rule 5*, network communication between *MS*, *CS*, and *MA* (including its integrated TP-SDK) should adopt secure transmission (e.g., via TLS channel). Otherwise, attackers can intercept, eavesdrop or tamper what users want to buy (e.g., in Step 1 of Fig. 1), the payment order information (e.g., in Step 2 of Fig. 2), or the transaction information (e.g., in Step 4 of Fig. 1). It can also directly cause the *Order Substituting* Attack, as we mentioned in Section 3.5.

According to the adversary model, we mainly concern how to detect the insecure network communication employed between the *MA* (including TP-SDK) and the remote server. We set a proxy to conduct MITM attack against HTTP and HTTPS connection to detect the potential flaw. The insecure communication between TP-SDK and *CS* may cause wide and serious consequence. Since TP-SDK is integrated by a large number of Android and iOS apps, all the *MA*s with this kind of TP-SDK will suffer vulnerabilities (such as payment information leakage, transaction interception and tampering, etc.), if the network communication is insecure. So we adopt a refined policy to detect the flaw in TP-SDK. We try to sniff and attack the network communication during a manually conducted payment process. If the connection between TP-SDK and *CS* is an HTTP connection, we regard it as insecure. Furthermore, if the connection is HTTPS, we will check whether it verifies the SSL/TLS certificate properly, or implements the certificate pinning. If the TP-SDK uses private protocol communicating with its server, we further audit the security of this protocol (since there are only four TP-SDKs, we could audit it manually).

For the communication between *MA* and *MS*, we only consider the situations of HTTP, insecure HTTPS (without certificate validation), and secure HTTPS. Our purpose is to find out the network connection of the exact step when *MS* returns $order_p$ / *TN* to *MA* is secure. Since our result needs high accuracy, we manually trigger the *MA* to the step and monitor the network traffic.

Since Apple introduced "App Transport Security", which defaults apps to requiring an HTTPS connection, our detection to this security flaw mainly focus on Android apps. The limitations of automated analysis methodology to detect insecure network communication for Android apps is so serious that lead to high inaccuracy. The key difficulty is that how to find the URL connection related to the step that *MA* and *MS* transmit the transaction information accurately. Among a variety of URL strings in apps, it's quite impossible to decide which URL is responsible for transmitting the order/transaction information only by name. In addition, it is common for apps to join several substrings to the ultimate URL address, or even use code obfuscation to sensitive URL, which also raise the difficulty of automated detection. Previous work like MalloDroid [26] only gave coarse detection result without identification of target URL's logic function, and Reaves et al. [27] even indicated the inaccuracy of such automated analysis, which further prove the difficulty of this work. Furthermore, finding the target URL with dynamic analysis involves deep human interaction, including registering and login account, clicking products, choosing in-app payment and paying for the order, which is also impossible to be automated and large scale. As a result, we can only do it manually to achieve accurate detection result.

4.6. Notified payment confirmation missing

As *Security Rule 6* implies, *MS* needs to make an extra payment order query (e.g., Step 8 of Fig. 1) to confirm every details of the payment order, even if it receives the payment notification. Since this part of implementation is on *MS*, we can only apply an indirect detection approach to detect the violation. We try to tamper a payment order information which is different from the original payment order but with legal signature, and pay for it. If the *MS* accepts the payment order and ships the commodities, then we can conclude that the *MS* does not re-confirm the notified payment order. Note that the tampered order message should be with correct signature, which means the samples here need to be based on the result of *KEY Leakage*. We perform the dynamic detection on a small portion of the samples manually due to the ethical consideration. Also the process involves much human interaction such as placing orders and checking the payment's status, which makes automated analysis almost impossible.

4.7. Signature validation missing

Security Rule 7 implies that *MS* is supposed to check the integrity of every received message (e.g., in Steps 3, 7, and 9 in

Table 1
TP-SDK distribution.

Cashier	Android	iOS
WexPay	2260	3786
AliPay	1299	1801
UniPay	574	785
BadPay	34	47
Total	2679	3972
Sample	7145	10,000

Table 2
Flaws in merchant apps.

Cashier	KEY leakage		Local Ordering	
	Android	iOS	Android	iOS
WexPay	155	249	104	88
AliPay	398	276	/	/
UniPay	0	0	0	0
BadPay	7	2	/	/

Fig. 1). Otherwise, *MS* would accept messages even with incorrect signature. To detect this flaw, we try to place an order but without actually paying for it, and then forge an order notification to the *MS* with incorrect signature. If the merchant accepts the payment order, Then we can conclude that *MS* fails to check the signature properly. Here the samples are based on the apps that commit *Notified payment confirmation missing*. Because we need to exclude the negative result caused by successfully confirming the notified payment.

5. Empirical study

To investigate the flawed in-app payment implementations, we first conduct our TP-SDK identification to the 7145 most popular Android apps from *Myapp* market and 10,000 popular iOS apps we downloaded from *25PP*. As Table 1 shows, 2679 Android apps and 3972 iOS apps integrate at least one TP-SDK, and most of them contain more than one TP-SDK. The proportion of Android and iOS apps supporting in-app payment is as high as 37.5% and 39.7%, respectively, which proves the prevalence of third-party in-app payment.

Then we detect each security flaws we mentioned in Section 4. We classify these flaws into four categories involving *MA*, TP-SDK, *MS*, and network communication. We find that hundreds of the merchants violate at least one security rule and none of the four TP-SDKs strictly obey these security rules. Besides, we further investigate the official documents and analyze sample codes released by four cashiers in-depth and gain some interesting and unexpected findings, which may imply the root cause of these flaws. Then we choose representative vulnerable apps based on the result of detection and then exploit their security flaws to prove the validity of our analysis. We provide them as case studies to illustrate the complexity of conducting concrete attacks against real world transactions.

5.1. Flaws in MA

We first detect those flaws in the *MA*s of both Android and iOS. The detection result is shown in Table 2. We can see that hundreds of the merchants leak their *KEY*s in *MA*s. Nearly one hundred merchants (Android and iOS respectively) using WexPay generate and send payment order in *MA*s.

Note that our *KEY Leakage* result of WexPay has no false positive since it is based on the response messages from WexPay's Web API as we mentioned in Section 4.2. The result of detecting WexPay destination URL is over 130 of Android and 110 of iOS. However, after our manually confirmation, 104 of the Android and 88 of the iOS apps really do *Local Ordering*, and the rest just hard-code the URL without invoking it. All the *Local Ordering* apps of WexPay leak the *KEY* since they generate and sign the payment order in *MA*. However, another 51 Android and 161 iOS apps either use the *KEY* to sign the received *TN* message or just hard-code the *KEY* without actually using it, both violating the security rules. Also we detect that nearly 500 Android and 430 iOS apps integrated AliPay contain strings with RSA private key features. Among them, we find out that 398 Android and 276 iOS apps actually leak their AliPay private keys using the locating techniques mentioned in Section 4.2, and the rest are the keys of other SDKs. Since only 34 Android and 47 iOS apps integrated BadPay, we do the *Key Leakage* detection manually. We find out 7 of the Android and 2 of the iOS apps leak their keys in several ways, such as hard-coding its *KEY* in apps or receiving *KEY* from server. In addition, we find out that several apps share the same *KEY* of WexPay and AliPay, which means that they are either using the same checkout account or developed by the same company.

Note that while some *MA*s of the other three TP-SDKs just sign a payment and send it to the TP-SDK, leading to the *KEY Leakage*, in our study we found NONE of Android or iOS apps using UniPay leak their *KEY*s or commit *Local Ordering*. One possible reason is that UniPay SDK does not need the signature of *TN* as a parameter, while other SDKs often do. This feature of UniPay does reduce the risk of *KEY* exposure and we will discuss a more direct cause in Section 5.5.

5.2. Flaws in TP-SDK

Since TP-SDKs are provided by the cashiers and integrated by the *MA*, flaws in specific TP-SDK directly affect the host *MA*. We evaluate the four most popular TP-SDKs provided by AliPay, WexPay, UniPay, and BadPay respectively, including Android and iOS version. The result is shown in Table 3. We find out that one type of SDK in Android and iOS has the same result. Only WexPay verifies *TN* correctly. *TN* accepted by WexPay SDK includes parameters of *merchant ID, transaction number*, etc. WexPay SDK achieves the *MA* certificate through system API in Android, and checks the consistency of the APK certificate and *merchant ID*. It also checks whether the *transaction number* belongs to the *merchant ID*. In contrast, we succeed in invoking the other TP-SDKs (AliPay, UniPay, BadPay) integrated in the *MA* by the transaction order of another *MA*. Also, we find that both WexPay and AliPay require the registered merchants to submit their certificates of *MA*, while UniPay and BadPay do not. Obviously, only WexPay makes use of the certificate to verify *TN*.

For *Incomplete Prompt*, we manually check every elements presented on the payment UI of every TP-SDK. We find out that all four TP-SDKs do not present the order's owner in the UI, leading to the risk of phishing. BadPay only shows the total amount of the payment order, which is obviously insufficient. WexPay and AliPay both show the order description submitted by merchants. But they do not require the merchant to submit necessary information about the order such as the order ID, the order owner, etc. UniPay and WexPay show the merchant name of the order while AliPay and BadPay do not. Also for UniPay, order ID and payment time will be shown to users only if a spinner on the UI is clicked. In all, every TP-SDK lacks necessary information more or less on payment UI, which may lead users to be deceived.

We manually check the implementation of network communication of four TP-SDKs. We reverse-engineer the TP-SDK using IDA [18] and JEB [19] as well as sniff the network connection with Wireshark [21] and Burp Suite [20]. Besides, we locate the methods/functions of sending/receiving network message in TP-SDKs and hook them (using Frida [23] and Xposed [22]) to observe how

Table 3
Flaws in TP-SDKs.

Cashier	Transaction verification	Information prompt					Network communication
		OrderID	Commodity	Owner	Merchant	Money	
WexPay	✓	×	✓	×	✓	✓	secure private protocal
AliPay	×	×	✓	×	×	✓	HTTPS pinning
UniPay	×	✓	✓	×	✓	✓	HTTPS pinning
BadPay	×	×	×	×	×	✓	HTTPS validation

Table 4
Flaws of MS and network communication.

Platform	Flaws in MS			Insecure network in MA			
	Sample	Rule 6[a]	Rule 7[b]	Sample	HTTP	HTTPS[c]	HTTPS
Android	15	9	2	87	45	4	38
iOS	10	4	1	/	/	/	/

[a] Notified Payment Confirmation Missing.
[b] Signature Validation Missing.
[c] Https without SSL certificate validation.

these TP-SDKs implement their network protocols. As a result, we find out that SDKs of AliPay and UniPay use HTTPS to connect to their servers and adopt certificate pinning. WexPay SDK uses a proprietary protocol to communicate with its server. After the reverse engineering, we find that it implements its key agreement algorithm based on ECDHE, and the ephemeral keys are authenticated with another public key of WexPay, which is hard-coded in the SDK. Thus, the protocol is secure enough to avoid an MITM attack. The SDK of BadPay validates the SSL certificate properly, thus, is secure. However, compared to the other three TP-SDKs, BadPay does not adopt SSL-pinning, which means it can not be protected against a compromised CA.

5.3. Flaws in MS

We tampered the payment order of 15 Android and 10 iOS apps with correct signature using their leaking *KEY*s, and paid for it to see whether *MS* would accept them. The result is shown in Table 4. Since the action need really exploit the *KEY Leakage* vulnerability, involving a lot human interactions like *MA* account registration, placing a merchant order, tampering the payment order, and paying for it, it's unrealistic to be automated and large-scale. So we did it manually and found that 9 of the 15 Android apps and 4 of the 10 iOS apps that finally accepted the tampered order, which means that their *MS*s miss the notified payment confirmation to *CS*. Among the 13 vulnerable apps, there are *MA* that use WexPay, AliPay or BadPay.

We further checked whether the *MS*s of the 9 Android apps and 4 iOS apps verify the signature properly. We got the notification message format according to cashiers' documents, forged the message with incorrect signature and sent it to the *Notify URL* address of the *MS*. The result is that two of the nine Android and one of the four iOS apps' servers still accept the payment. It indicates that even if the *KEY* is not leaked, attackers can still buy products without paying for it.

5.4. Flaws of network communication

Since Apple introduced "App Transport Security", which defaults apps to requiring an HTTPS connection, we only detect the insecure network communication on Android apps. We manually test 87 most popular Android *MA*s chosen from the 2679 Android apps with embedded TP-SDKs to evaluate the security of their connections to the *MS* during the payment. The result is also shown in Table 4. There are 45 apps using HTTP connection and 42 apps using HTTPS connection. Among 42 apps who use HTTPS connection, four of them fail to validate SSL certificate properly. In addition, although we do not find proprietary protocol used by *MA* to communicate with the *MS*, some apps adopt home-brewed encryption schemes to protect the content in HTTP connection. Since those encryption schemes generally lack a mature session key management, we regard them as insecure without further investigating the security of their encryption. In all, these 49 vulnerable apps increase the risk of suffering *Order Substituting* Attack.

Although we did not test all the 2679 apps due to the inaccuracy of automatic analysis (see details and explanation in Section 4.5), the result of the 87 samples through manual work shows that a large proportion of merchants are still not cautious in implementing secure network communication. It is an astonishing result that there are still so many popular apps use insecure HTTP channel even if all of them are related to financial transaction. Note that the tested samples are the most popular apps with larger user amount and stricter security audit. We believe that those samples with less user amount may perform worse on building secure communication. Moreover, we find that cashiers only request the merchant to adopt HTTPS communication in the *MA* as an optional requirement rather than a mandatory enforcement. So merchants may ignore the request and implement insecure network communication.

5.5. Root cause inquiry

In our opinion, cashier as well as merchant should be blame for the vulnerabilities of in-app payment in mobile platform. For cashiers, they once released ambiguous, confusing and self-contradictory documents and sample codes (although they may correct some of them afterwards). Even the official sample code violates several security rules. As a result, merchants who follow these samples codes suffer our proposed attacks. Also inappropriate implementation of cashiers' TP-SDK may expand the attack effect along with other flaws (as we mentioned in Section 3.5). For merchants, developers didn't update their app in time when some of the cashiers correct their vulnerable sample code. In addition, merchant introduced their in-app payment functionality without fully understanding the third-party mobile payment model, along without necessary security testing and auditing.

We also have some interesting findings after reviewing the sample codes as well as manually checking the documents of the four TP-SDKs, Even though all of the four TP-SDK documents claim that the *KEY* needs to be kept in secret, their sample codes implement the process of message signing in their client apps, leading

to the *KEY* leakage, except UniPay. It can be used to explain that so many *MA*s commit vulnerabilities when using WexPay, AliPay or BadPay (while become secure using UniPay). For example, the sample code released by WexPay directly commits *Local Ordering*, which obviously conflicts with its official process. We also find that the figure describing the process of the payment released by AliPay defines that the order should be signed in client app and so does the sample code, but the code comment in the sample says that the signing step should be in *MS*. We hypothesize that the contradiction between documents and sample code confused merchant developers a lot, leading those developers who follow the sample during their development to commit these mistakes. Only the sample code of UniPay implementation keeps consistent to their documents, making the order generating and signing in the *MS* code, so in our detection none of the APPs are flawed when using UniPay. Also, we find that UniPay provide very detailed materials to instruct developers to deploy their certificates and test development environment, etc. It shows that the cashier is the key factor to the security of merchant implementation. In addition, since the *KEY* of WexPay can be modified to any string as long as merchant notifies cashier, we find that some leaked keys, which are supposed to be random strings, are modified to a weak key such as *12345678912345678912345678912345*, or just the name of merchant, which may suffer brute-force dictionary attack, or social engineering.

We also try to discover the incentive of flawed *MS*s. We find that not all cashiers release the sample code of *MS*, thus merchant needs to implement it without example by themselves. Even if there is sample code for *MS*, server implementation varies a lot compared with the client. Merchant may implement their servers by using Java, PHP,.NET, Python or even native language, which are out of the language scope of sample code released by cashier. Even though cashiers suggest merchants to do these validations in documents or some even implement them in sample code if it has, merchant may also ignore it during the code transplant or without existing correct code examples.

Although the third-party in-app payment involves financial transaction and should have high security level, none of the cashiers emphasize the security in particular. Some cashiers just mention it (e.g., suggesting the merchant to implement the network communication in HTTPS in the end of the documents), which is easy to be ignored by merchants. Not to mention the fact that improper designed TP-SDK and incorrect documents/sample codes released by cashiers. Previous work [1–4] mainly focus on the security of the merchant. Acar et al. [28] ascribe Android code insecurity to informal documentation such as Stack Overflow, while official API documentation is secure but hard to use. However, our work shows that when it comes to the mobile third-party in-app payment, even official documents/sample code released by cashiers lead to the code insecurity, which may be helpful to improve the security of the whole ecosystem of third-party payment.

5.6. Case studies

We choose several flawed merchant apps to perform real attacks. It shows that these detected violations of security rules can directly lead to serious consequence including financial loss in real world.

5.6.1. Order Tampering in iOS app

The first case is an iOS app used by patients to make appointments with doctors. This app requires the patients to pay the registration fee while making an appointment. However, the app violated *Security Rule 1*, signing the order information in the client app. We hooked function used to execute SHA1-RSA signing of the order information in the *MA* with the help of Frida, a universal binary analysis framework that supports iOS app binary code instrumentation [23]. We have implemented an Frida script to tamper the amount money to only one *yuan* in the order information and re-signed the message. Then the TP-SDK successfully accepted the order information, asking for paying this appointment with one *yuan*. After we paid, the app in the *MA* showed that we successfully make an appointment to the doctor with the registration fee paid with *the original price* (30 *yuan*). Even when we went to the doctor afterwards, our "*less paid*" registration fee did not attract any awareness, which indicates that the merchant fails to confirm the notified order details. We then reported this vulnerability to the merchant and helped update the app in a week.

5.6.2. Order substituting

We proved that *Order Substituting* Attack can be *automated* and let user pay for the attacker's order without awareness. We employed the attack on a wireless router of our local area network. Since we control the router, we can conduct MITM attack to the devices in the same LAN. We set a MITM proxy on the router to replace self-signed HTTPS certificate to the original one and decrypt the content of HTTPS connection.

The victim app in this case is a popular e-Book Reader of Android with over 20 million downloads. Users can purchase non-free e-books in this app via payment channels of all four cashiers. We take the BadPay for the case. The app use HTTP connections when users browse book lists in the app. When users want to pay for a book then the connection will turn to HTTPS but the app fails to check the HTTPS certificate correctly (allow all certificates described in [26]). So our proxy can intercept, eavesdrop and tamper the connection. Once a user orders a book and is about to pay, our proxy extracts from the network traffic which book the user want to buy and the order information including price (10 *yuan* in the case). Then our proxy places a new order request using *another* app (a take-out food order app) to buy a burger which is also 10 *yuan*, and get the order information from the *MS* without paying for the burger in the latter step. Instead, the proxy intercepts the payment order response of the book and substitutes it with the attacker's burger order. As a result, the *MA* receives the replaced order response and prompts the user with its payment Activity (UI component of Android apps) of the TP-SDK. Note that information on the payment Activity of BadPay only including price, are exactly the same as the price of the book. User cannot distinguish this replaced order and is cheated to pay for it. Thus after the payment, the burger is paid and delivered to us while the e-book is still kept unpaid. When user paid for the attacker's order and found their own orders are still unpaid, they just believe it is the delay of the *CS* that leads to a temporary unpaid status. Imagine that if the take-out food order app here becomes a malicious *MA* controlled by the attacker, then the attacker can easily generate an order in arbitrary price according to the victim's order and substitute it. Thus, after user pay for it, the money is directly transferred to the attacker's account. Even with those TP-SDKs that display the merchant name (WexPay and UniPay), the attacker can still cheat users to pay for attacker's order in the same app.

Since a thorough fix of this issue is to apply HTTPS to the whole website, it took several weeks for the merchant to release the new version of the app after receiving our report. After that, the vulnerability was eliminated.

5.6.3. Android app with multiple vulnerabilities

Apps may violate multiple security rules and even patch vulnerabilities incompletely, so a relevant transaction could be vulnerable to not only a single type of attack. To illustrate, we demonstrate how to acquire free movie tickets in different ways by exploiting a movie ticket ordering app with an approximate 10 million users. The vulnerable app allows users to select the cinema, the movie,

and the seats they want, and then buy the movie tickets online via an in-app payment. After the payment, users will get a ticket code and when they get to the cinema, they can enter the ticket-code on an automated ticket machine to get the real movie tickets.

Unfortunately, We detected several security flaws in this app when users pay for tickets via WexPay in the app. The first one is that the app commits *Local Ordering* mistake and thus exposes the secret key and notify URL. As a result, an attack can hook the order-generation function based on Xposed in the app and tamper the price to a particular value (e.g., one *yuan*). In addition, with this leaked secret key, an attacker is able to download all bills of each day of the merchant. The bill also contains the details of every transaction in that day, including payer, paying bank, discount, etc. Thus the attacker can know exactly how much people buy movie tickets through WexPay everyday in that app, which obviously should be confidential to any unauthorized visitors. To validate the vulnerability, we conducted a penetrating test to buy a ticket, and paid for it via WexPay with only one *yuan* (or even cheaper if we wish) and received the available ticket-code, fetched the physical tickets and watched the movie successfully.

After we reported the *Local Ordering* flaws to the merchant (and they replied that they have fixed the vulnerability in the next version of the app), we tested the new app again. This time we found that the app is still vulnerable to another kind of attack. In detail, an attacker follows all the normal step of a transaction until the app invokes the WexPay's payment Activity to ask for the payment password to pay for the ticket. Then the attacker terminates the following steps and directly forges a payment notification with the signature signed by the leaked *KEY* in the previous attack to the merchant server. Surprisingly, our ethical test found that we could still get valid ticket-code, which means the merchant only moved the *placing order* step from the app to its server, but never renewed the leaked *KEY*. Even worse, the merchant server still miss the notified payment confirmation as before.

Before informing the merchant and repaying the tickets we have bought this time, we have waited for a certain period (thus the merchant could check the collection of all past orders periodically sent from the cashier) to check whether the merchant verifies it. Disappointingly, until we explained our behavior to the merchant this time, they still had no idea about our two penetrating tests. In the end, we help them fix all these flaws through renewing the secret key, reconfirming the notified payment, and verifying its signature correctly. And we have repaid all fees for tickets bought in our penetrating tests.

6. Ethical consideration

We carefully designed our experiments to avoid ethical problem. First, we reported all our findings and the behaviors we performed during the experiments to the related parties and did what we could do to help them improve the systems. Our effort was appreciated by these organizations. In detail, we reported the mistakes in documents/sample codes to WexPay, AliPay and BadPay. All of them have fixed and updated it. For instance, Official documents of AliPay [29] shows an updated payment process figure. The original figure told the developers to generate and sign the payment order in client app (as we mentioned in Section 5.5), which is obviously insecure. Both AliPay and WexPay updated their official attentions of developing due to our suggestions [30,31]. All the three cashiers expressed their gratitude to us. Also we detected flaws (such as missing order signature validation) of several merchant servers described in Section 5.3 and performed several proof-of-concept attacks described in Section 5.6. We reported all these flaws and explained our behaviors to these influenced merchants as soon as we carried out our experiments, and helped them to fix these vulnerabilities. Since hundreds of merchants suffer flaws in their apps as we mentioned in Section 5.1, it would be very difficult for us to inform all of the merchants directly. Hence, we report the vulnerable ***MA*** list to the Security Response Center of Tencent, Ant Financial, and Baidu, who are responsible for the security of their payment ecosystems (WexPay, AliPay, and BadPay). They informed all the related merchants of their security risks, revoked leaked *KEY* and renewed them.

We use Web APIs provided by cashiers as oracles to help finding the leaked *KEY* in ***MA***, which need to brute-force the parameters of the API and may induce potentially heavy load to cashiers server. We did restrict the frequency and times of invoking the API to avoid potential denied of service attack against the server. After we described our detecting method to cashiers, they confirmed the issue and planned to impose some constraints to invoking the API in future (WexPay even deprecated their download history bill API).

In addition, we ensure no financial damage was inflicted upon the merchants by returning items or re-paying the unpaid orders, etc. The victim user in the *Order Substituting* Attack (described in Section 5.6.2) is actually a colleague of us. We informed him beforehand and later paid for the e-book order for him. We made use of the result of downloaded history bills of merchants to evaluate the feasibility of *Unauthorized Query* attack, and helped to detect *KEY Leakage* in ***MA*** using WexPay and AliPay. We not only described our detecting method in detail to merchants, but also deleted all these data at once to avoid further exposures.

7. Discussion

Though we could analyze more samples, we believe that the quantity is enough since the methodology we introduced in the paper are universal and could be applied to larger quantity of samples. Besides, all samples we choose are popular and influential. We believe that other apps are similar to our samples and our samples are enough to help summarize the status. Although our large scale analysis reveals that the quantity of flawed in-app payment implementations is surprising, we have to point out that we underestimate the actual danger. During the TP-SDK identification, our methodology is based on static code feature. We classify many apps, especially in Android, protected by code packing techniques as not using any TP-SDK, while they do integrate TP-SDKs. In the detection of flaws in *MA*, we adopt static analysis for searching leaked *KEY* candidate, which may ignore those encoded-but-exposed *KEY* in apps. However, only a small portion of apps adopt such self-protection measures according to previous studies [24,32]. Since our work presents a systematic approach to detect those vulnerabilities, we believe the analysts of merchants and cashiers could adopt our approach to audit their products before releasing.

Our security analysis mainly focuses on the interfaces of multi-party involvement in the third-party in-app payment. We pay less attention to the attacks or flaws only involving single party (traditional user-to-merchant payment model), for instance, the merchant order tampering, or denied-of-service attack on order ID or transaction ID.

8. Related work

8.1. Insecure third-party SDK

Meanwhile, vulnerabilities or threats introduced by third-party SDKs in mobile applications have also been studied by many researchers. Chen et al. [33] studied on potentially-harmful libraries across Android and iOS through clustering similar packages to identify libraries and analyzing them using AV systems to find those libs. Wang et al. [34] identified serious authentication and

authorization flaws in applications that integrate Single-Sign-On SDKs. Li et al. [35] aims to understand and analyze the security hazards imported by Push service in Android applications. ClueFinder [36] tries to identify privacy leakage from apps to untrusted third-party libraries. Wang et al. [37] and [38] demystified and assessed the vulnerabilities of OAuth protocol on mobile platform, which often introduced by third-party providers as SDKs in Android applications. However, the payment SDK in mobile applications has never been studied before. We propose a comprehensive methodology to detect various security rule violations in those apps who embed third-party payment SDKs. All of these flaws lead to serious consequence and result in financial loss for different parties involved.

8.2. Mobile app vulnerabilities

The security analysis of vulnerabilities in mobile application has also become hot spot these years, with the dramatic growth in mobile users. It's common that misuse of security libraries leads to flaws in apps. iCryptoTracer [39] and CryptoLint [40] is proposed to identify the misuse of cryptography functions in iOS and Android apps respectively. CRIOS [41] focus on the large-scale app analysis for third-party library usage and network security on iOS. Fahl et al. [26], SMV-Hunter [42] and Reaves et al. [27] all perform their app analysis to detect unsafe SSL/TLS and cryptography usage. Different from all of these work, our analysis of security flaws caused by apps integrating third party in-app payment libraries. We reveal that these flaws during in-app payment caused both by merchant apps and third-party payment SDK providers. Among these flaws, *KEY Leakage* has been studied before. Both PlayDrone [43] and CredMiner [44] try to detect token exposure of AWS and OAuth in Android applications. However, we adopt a more efficient and accurate methodology, combining local program analysis and a remote Web API, to detect such flaws in third-party payment. Besides, our work covers the security threats and flawed implementations throughout the whole payment process, rather than focusing on a particular type of vulnerability detection.

8.3. E-commerce vulnerabilities

The security analysis of e-commerce and payment has attracted the attention of researchers in recent years, since vulnerabilities may cause great impact and financial loss. As far as we know, the only similar work is implemented by VirtualSwindle [45], which can perform an automatic attack against in-app billing service. However, it seems to be just a small part of our work. First, the in-app billing service described in the paper is just one scenario of the in-app third-party payment. The service is simpler and less popular compared to our research targets. Second, VirtualSwindle can only launch one type of attack and the adversary model assumed in the paper is too limited. However, our work describes four types of threats and attack model is more diversified. Third, only 85 Android apps were studied in the paper compared with the thousands of samples in our work. Overall, we perform a more large-scale and systematic analysis to third-party in-app payment. Our work focuses on finding all flawed implementations throughout the whole payment process, not only launching one type of attack.

Another work on mobile payment is [46], which studied the security of mobile off-line payment token (QR code). Our work aims to find the flawed implementation of in-app payment, not the QR code payment in off-line situation.

Wang et al. [1] are the first to analyze logic vulnerabilities in Cashier-as-a-Service based web stores, and found several logic flaws manually. Sun et al. [3] propose to detect logical vulnerabilities in e-commerce application through static analysis of available program code. Pellegrino and Balzarotti [4] proposed the idea of black-box detection of logical vulnerabilities in e-shopping applications Sudhodanan et al. [5] propose an automatic technique based on attack patterns for black-box, security testing of multi-party web applications. InteGuard [2] offers dynamic protection of third-party web service integration including cashier service in merchants' websites. All of the work mainly target on e-commerce in web application, while we focus on mobile platform.

Compared with an automated security analysis against Android apps, that against iOS apps (especially large-scale apps) is more difficult since much less program analysis tools as well as methodologies for iOS apps have been proposed. In comparison to the conference version [47] of this paper we present new contributions especially on the analysis of iOS platform including: (a) new methodology of third-party payment SDK identification for 10,000 iOS apps; (b) automated vulnerability detection method designed for 3972 iOS payment apps; (c) new detection method of certain flaws for both Android and iOS (old method in our previous work was blocked due to our report); (d) new experiment results for vulnerable iOS apps, embedded TP-SDKs and their servers; (e) new vulnerability exploits and attack cases of vulnerable iOS apps in real world.

9. Conclusion

Insecure in-app payment is becoming a main threat to mobile ecosystem as more and more online transactions are transferring from website to app. Different from traditional web payment, in-app payment involves more sophisticated implementation details and the process is often obscure. To demystify processes of popular in-app payments and reveal potential security risks, we conduct a comprehensive analysis on mainstream third-party in-app payment schemes in Android and iOS apps. Our analysis investigates implementations of four in-app payments and concludes a series of security rules that should be obeyed. We not only pinpoint the serious consequence of violating security rules, but also detect these flawed implementations. Our statistics paint a sobering picture–hundreds of apps integrated with third-party in-app payment SDKs are vulnerable. Besides, our further investigation indicates that cashier, as well as merchant are blame for these flawed implementations. We hope our study can remind and guide developers of both merchants and cashiers to build more secure in-app payments.

Declaration of interests

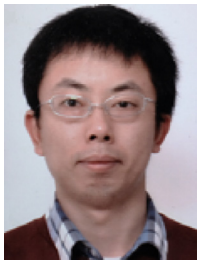
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Wang R, Chen S, Wang X, Qadeer S. How to shop for free online–security analysis of cashier-as-a-service based web stores. In: Proceedings of the 32nd IEEE symposium on security and privacy; 2011.
[2] Xing L, Chen Y, Wang X, Chen S. Integuard: toward automatic protection of third-party web service integrations. In: Proceedings of the 20th network and distributed system security symposium (NDSS); 2013.
[3] Sun F, Xu L, Su Z. Detecting logic vulnerabilities in e-commerce applications. In: Proceedings of the 21st network and distributed system security symposium (NDSS); 2014.
[4] Pellegrino G, Balzarotti D. Toward black-box detection of logic flaws in web applications. In: Proceedings of the NDSS; 2014.
[5] Sudhodanan A, Armando A, Carbone R, Compagna L. Attack patterns for black-box security testing of multi-party web applications. In: Proceedings of the 23rd network and distributed system security symposium (NDSS); 2016.
[6] WexPay. https://pay.weixin.qq.com.
[7] AliPay. https://open.alipay.com; a.
[8] UniPay. https://merchant.unionpay.com/join/index.
[9] BadPay. https://b.baifubao.com.
[10] Myapp Android Market. http://android.myapp.com/;

[11] APPINCHINA CHINESE APP STORE RANKINGS. https://www.appinchina.co/market/app-stores/.
[12] Tencent Open Platform. http://open.qq.com/eng/reg.
[13] 25pp iOS Market. https://www.25pp.com/ios/.
[14] ProGuard. http://proguard.sourceforge.net/.
[15] ios-class-guard. https://github.com/Polidea/ios-class-guard.
[16] AndroGuard. https://github.com/androguard/androguard.
[17] Radare2. https://rada.re/r/.
[18] IDA. https://www.hex-rays.com/products/ida/index.shtml.
[19] JEB. https://www.pnfsoftware.com/jeb/.
[20] Burp Suite. https://portswigger.net/burp.
[21] Wireshark. https://www.wireshark.org/.
[22] Xposed. http://repo.xposed.info/.
[23] Frida. https://www.frida.re/.
[24] Yang W, Zhang Y, Li J, Shu J, Li B, Hu W, et al. Appspear: Bytecode decrypting and DEX reassembling for packed android malware. In: Proceedings of the research in attacks, intrusions, and defenses; 2015.
[25] Damopoulos D, Kambourakis G, Portokalidis G. The best of both worlds: a framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones. In: Proceedings of the seventh european workshop on system security; 2014.
[26] Fahl S, Harbach M, Muders T, Baumgärtner L, Freisleben B, Smith M. Why Eve and Mallory love android: an analysis of android SSL (in) security. In: Proceedings of the ACM conference on computer and communications security; 2012.
[27] Reaves B, Scaife N, Bates A, Traynor P, Butler KR. Mo (bile) money, mo (bile) problems: analysis of branchless banking applications in the developing world. In: Proceedings of the 24th USENIX security symposium (USENIX security 15); 2015.
[28] Acar Y, Backes M, Fahl S, Kim D, Mazurek ML, Stransky C. You get where you're looking for: the impact of information sources on code security. In: Proceedings of the 37nd IEEE symposium on security and privacy; 2016.
[29] AliPay payment process. https://doc.open.alipay.com/doc2/detail?treeId=59%26articleId=103658%26docType=1b.
[30] Alipay's attentions of developing. https://docs.open.alipay.com/59/104027/.
[31] Wexpay's attentions of developing. https://pay.weixin.qq.comhttps://wiki/doc/api/jsapi.php?chapter=23_3&index=3#menu2b.
[32] Duan Y, Zhang M, Bhaskar AV, Yin H, Pan X, Li T, et al. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In: Proceedings of the 25th annual network and distributed system security symposium, NDSS; 2018.
[33] Chen K, Wang X, Chen Y, Wang P, Lee Y, Wang X, et al. Following devil's footprints: cross-platform analysis of potentially harmful libraries on android and IOS. In: Proceedings of the IEEE 37th symposium on security and privacy; 2016.
[34] Wang R, Zhou Y, Chen S, Qadeer S, Evans D, Gurevich Y. Explicating SDKs: uncovering assumptions underlying secure authentication and authorization. In: Proceedings of the IUSENIX security; 2013.
[35] Li T, Zhou X, Xing L, Lee Y, Naveed M, Wang X, et al. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In: Proceedings of the I 21st ACM SIGSAC conference on computer and communications security; 2014.
[36] Nan Y, Yang Z, Wang X, Zhang Y, Zhu D, Yang M. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In: Proceedings of the annual network and distributed system security symposium; 2018.
[37] Wang H, Zhang Y, Li J, Liu H, Yang W, Li B, et al. Vulnerability assessment of OAuth implementations in android applications. In: Proceedings of the 31st annual computer security applications conference; 2015.
[38] Chen EY, Pei Y, Chen S, Tian Y, Kotcher R, Tague P. Oauth demystified for mobile application developers. In: Proceedings of the ACM SIGSAC conference on computer and communications security; 2014.
[39] Li Y, Zhang Y, Li J, Gu D. icryptotracer: Dynamic analysis on misuse of cryptography functions in IOS applications. In: Proceedings of the 8th international conference on network and system security; 2014.
[40] Egele M, Brumley D, Fratantonio Y, Kruegel C. An empirical study of cryptographic misuse in android applications. In: Proceedings of the ACM SIGSAC conference on computer and communications security (CCS); 2013.
[41] Orikogbo D, Büchler M, Egele M. Crios: toward large-scale IOS application analysis. In: Proceedings of the 6th workshop on security and privacy in smartphones and mobile devices; 2016.
[42] Sounthiraraj D, Sahs J, Greenwood G, Lin Z, Khan L. SMV-hunter: large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps. In: Proceedings of the 21st annual network and distributed system security symposium (NDSS); 2014.
[43] Viennot N, Garcia E, Nieh J. A measurement study of Google play. In: Proceedings of the ACM SIGMETRICS performance evaluation review; 2014.
[44] Zhou Y, Wu L, Wang Z, Jiang X. Harvesting developer credentials in android apps. In: Proceedings of the 8th ACM conference on security & privacy in wireless and mobile networks; 2015.
[45] Mulliner C, Robertson W, Kirda E. Virtualswindle: an automated attack against in-app billing on android. In: Proceedings of the 9th ACM symposium on Information, computer and communications security; 2014.
[46] Bai X, Zhou Z, Wang X, Li Z, Mi X, Zhang N, et al. Picking up my tab: Understanding and mitigating synchronized token lifting and spending in mobile payment 26th USENIX Security Symposium; 2017.
[47] Yang W, Zhang Y, Li J, Liu H, Wang Q, Zhang Y, et al. Show me the money! finding flawed implementations of third-party in-app payment in android apps. In: Proceedings of the annual network & distributed system security symposium (NDSS); 2017.

Wenbo Yang is currently a Ph.D. candidate in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, China. He received his B.S. degree from Shanghai Jiao Tong University in 2012. His research interests include system and software security.



Juanru Li is currently a Ph.D. candidate in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, China. He received his B.S. degree from Shanghai Jiao Tong University in 2007. His research mainly involves developing secure crypto software and systems, and hardening existing crypto software against evolving security threats.

Yuanyuan Zhang works as an Associate Professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. After received her Ph.D. from Shanghai Jiao Tong University in 2009, she worked in East China Normal University as Assistant Professor and then in INSA de Lyon as Postdoc. Her research interests cover a wide range of issues in computer system security, program analysis, mobile security, and IoT system authentication and privacy.

Dawu Gu is a professor at Shanghai Jiao Tong University in Computer Science and Engineering Department. He received his B.S. degree in applied mathematics in 1992, and his M.S. and Ph.D. degree in cryptography in 1998, both from Xidian university of China. His current research interests include cryptography, side channel attack, and software security. He leads the Laboratory of Cryptology and Computer Security (LoCCS) at SJTU. He has got over 150 scientific papers in academic journals and conferences, owned 15 innovation patents. He was the winner of Yangtze River Scholar Distinguished Professors Program in 2014, New Century Excellent Talent Program in 2005, both made by Ministry of Education of China. He serves as board members of China Association of Cryptologic Research and Shanghai Computer Society. He also serves as several technical editors for China Communications, J. Cryptologic Research, and Information Network Security, etc. He has been invited as Chairs and TPC members for many conferences and workshops