Liu YT, Du D, Xia YB *et al.* SplitPass: A mutually distrusting two-party password manager. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(1): 98–115 Jan. 2018. DOI 10.1007/s11390-018-1810-y

# SplitPass: A Mutually Distrusting Two-Party Password Manager

Yu-Tao Liu[1], *Member, CCF, IEEE*, Dong Du[1], Yu-Bin Xia[1,*], *Senior Member, CCF, Member, ACM, IEEE*
Hai-Bo Chen[1], *Distinguished Member, CCF, Senior Member, ACM, IEEE*
Bin-Yu Zang[1], *Distinguished Member, CCF, Member, ACM, IEEE*
and Zhenkai Liang[2], *Member, ACM, IEEE*

[1]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China*
[2]*School of Computing, National University of Singapore, Singapore 117417, Singapore*

E-mail: {ytliu.cc, Dd_nirvana, xiayubin, haibochen, byzang}@sjtu.edu.cn; liangzk@comp.nus.edu.sg

Received February 24, 2017; revised April 11, 2017.

**Abstract** Using a password manager is known to be more convenient and secure than not using one, on the assumption that the password manager itself is safe. However recent studies show that most popular password managers have security vulnerabilities that may be fooled to leak passwords without users' awareness. In this paper, we propose a new password manager, SplitPass, which vertically separates both the storage and access of passwords into two mutually distrusting parties. During login, all the parties will collaborate to send their password shares to the web server, but none of these parties will ever have the complete password, which significantly raises the bar of a successful attack to compromise all of the parties. To retain transparency to existing applications and web servers, SplitPass seamlessly splits the secure sockets layer (SSL) and transport layer security (TCP) sessions to process on all parties, and makes the joining of two password shares transparent to the web servers. We have implemented SplitPass using an Android phone and a cloud assistant and evaluated it using 100 apps from top free apps in the Android official market. The evaluation shows that SplitPass securely protects users' passwords, while incurring little performance overhead and power consumption.

**Keywords** password manager, privacy protection, mobile-cloud system

## 1 Introduction

Password-based authentication, though well-known for its intrinsic weakness on security and usability[1], still prevails on the Internet web service. As users typically have many web accounts and multiple Internet devices (e.g., PCs, mobile phones, and tablets), they often use a password manager to manage all the passwords, not only for convenience, but also for security[2-3]. A common password manager has two major components: password storage and password filling. The passwords are usually stored in one place, e.g., a database on a PC①, in cloud②, or on a phone[4]. On the device side, plugins or apps are installed to fetch passwords from the database and auto-fill them.

*Problem.* Since a password manager controls all the passwords, it becomes an attractive target for attackers. If a user chooses to save the password database in a local device (e.g., a laptop or a phone, as shown in Fig.1(a)), then an attacker can directly get the database if it manages to get physical access to the device. Even if the password manager encrypts the database, it still has to decrypt it in memory during password filling. Even worse, the passwords may be in memory for a long time after login. For example, the default email app in Android maintains the password in plaintext in memory at all times, and KeePass, another popular password manager, also keeps its database in memory[5] until it terminates. Thus, an attacker who gets full control over

Regular Paper
This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000104, the National Natural Science Foundation of China under Grant Nos. 61572314 and 61525204, and the Young Scientists Fund of the National Natural Science Foundation of China under Grant No. 61303011.

*Corresponding Author

①1password. http://1password.com, Dec. 2017.

②Last pass. http://lastpass.com, Dec. 2017.

©2018 Springer Science + Business Media, LLC & Science Press, China

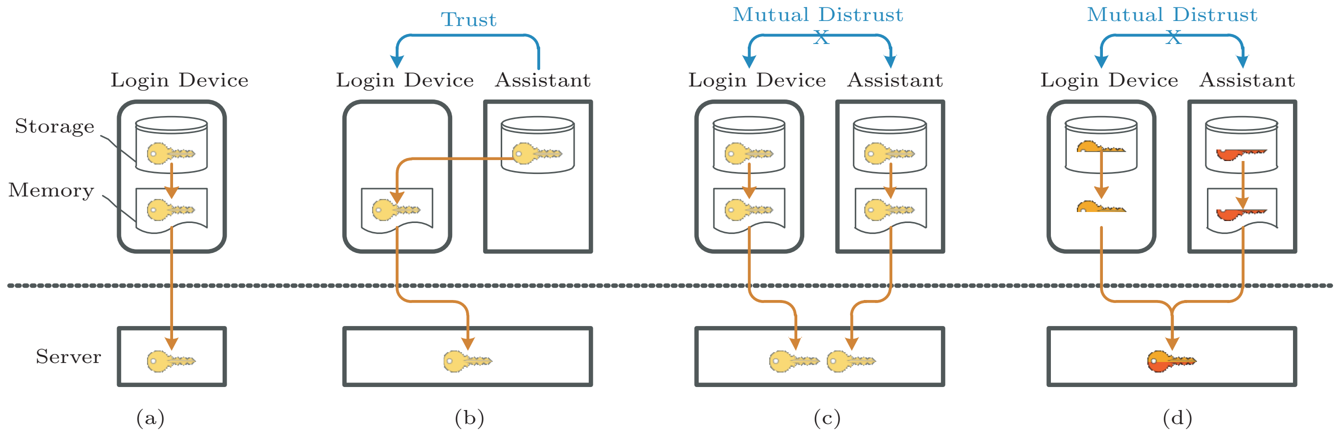


Fig.1. Comparison of different password managers. (a) Local password manager. (b) Assistant-based password manager. (c) Mutual distrust (with server change). (d) SplitPass (no server change).

the device can steal passwords by memory scanning or other sophisticated attacks[6].

The user may alternatively use an assistant-based password manager which stores everything in an assistant (e.g., a cloud or phone), as shown in Fig.1(b). For example, LastPass③ stores users' passwords on a cloud, and only sends passwords to the device during login. However, the cloud itself introduces new attacking surfaces as it may now suffer from a single point of breach. A malicious cloud operator can easily steal the saved passwords[7]. Instead of saving passwords on a cloud, Tapas[4] stores encrypted passwords on a trusted device (e.g., a phone), and saves the key on users' desktop PCs (here the PC is the login device). During the login, the PC will ask the trusted device for the encrypted password, decrypt it in memory and send it to the web server. However, both Tapas and LastPass assume that the memory of the login device is secure, which may unfortunately not be true. For example, an attacker may plant a rootkit to the PC to keep monitoring the memory to retrieve users' passwords.

The auto-filling feature of a password manager can also be dangerous. An attacker can control a WiFi router, inject a malicious form into a web login page, fool a password manager to auto-fill the password, and extract it without the user's awareness[2]. Other known issues include bookmarklet vulnerability, CSRF (cross-site request forgery) and XSS (cross-site scripting) vulnerabilities[3].

*Our Solution.* In this paper, we aim to implement a password manager that leverages two mutually distrusting components for authentication. One naive implementation is to require two independent passwords to a single web site, with one password saved on the login device and the other on an assistant (e.g., in a cloud). The login device and the assistant mutually distrust each other, which means that one can never send its password to the other. Thus, an attacker needs to compromise the login device as well as an assistant to steal both passwords, which significantly raises the bar of a successful attack. Such process of login is shown in Fig.1(c). This, however, requires changes to both the applications and server-side authentication process, which hinders its adoption to existing applications.

To this end, we propose SplitPass, a password manager for web services with transparent protection. Based on a component on the device performing the login and an assistant in the cloud, it requires changes to neither the web server nor the client-side application. In this paper, we focus on a specific scenario in which the login device is an Android device and the assistant is in a public cloud. Our solution can be easily adapted to other environments, such as desktop browsers. A password is then split into two shares: the first half is stored on the device and the second half is on the assistant cloud. During the login, both the device and the cloud assistant will collaborate to assemble their shares of password into the password expected by the server. In this process, neither the login device nor the cloud assistant gets access to the other's share of the password, and the process is transparent to the server.

More specifically, SplitPass not only puts the data of the second half but also offloads the assembling process of the second half. As secure web authentication service is built upon SSL (secure sockets layer), the two components of SplitPass work together to create

③Lastpass. http://lastpass.com, Dec. 2017.

an SSL login request, sharing the process of SSL record encapsulation and network packet framing. SplitPass is transparent to the server. For clients without SplitPass, users can authenticate using the full password.

In order to be transparent to the server, SplitPass needs to keep the integrity of SSL sessions as well as TCP sessions between the device and the cloud assistant. In another word, the cloud assistant should send the second part of the password to the server on behalf of the mobile device. This is implemented by synchronizing states of both SSL and TCP sessions between the device and the cloud assistant. Such state synchronization still assumes the mutual distrust between the two parties.

We have implemented SplitPass on Android by modifying the SSL library in the Android framework. Thus, all the apps using the default SSL library can benefit from SplitPass without modification. Evaluation using real apps shows that SplitPass securely protects users' passwords while incurring little overhead on performance and energy, and as well no impact on user experiences.

Our primary contributions are as follows.

1) We study the current password management schemes on mobile device, develop a threat model, and introduce password splitting to improve the security as well as the usability of password management.

2) We design a system named SplitPass, which is an extension of current password-based authentication mechanism. The server needs no modification to use SplitPass, and in most cases, the apps also need no change, which makes SplitPass a practical system that is easy to deploy.

3) We implement SplitPass based on Android framework, and use real apps for both security and performance evaluation.

## 2 Background and Overview

### 2.1 Threat Model and Assumptions

SplitPass assumes a strong adversary model such that an attacker can physically access a mobile device. Thus, the entire memory and storage are vulnerable through software or even physical attacks. The cloud assistant for storing portion of the password is also untrusted, which can be compromised by hackers or malicious operators. SplitPass assumes that an attacker cannot compromise both the mobile device and the cloud assistant at the same time. The two sides are independent to each other and thus do not collude.

To sign into web-based services, the device needs to have Internet connections. The pervasive cellular and wireless network coverage makes the assumption realistic. SplitPass is mostly applicable to access to passwords with relatively short-term login processes, which require network connection themselves.

We assume that apps on the mobile device log into the servers through SSL connections and use the SSL library offered by the Android system. In these apps, passwords are used for authentication and are sent to the servers directly. We assume that the users are aware of SplitPass and will use SplitPass' password format instead of entering the full password on the device, because an app might be malicious, like a phishing app that fools the user to input passwords.

SplitPass focuses on protecting passwords and assumes that there are other ways to authenticate users to access the passwords, like using fingerprints. It has been shown that users tend to reuse passwords for different web accounts[8]. More than two-third users have only four passwords for all of their web accounts. Some users even use their bank passwords everywhere. Suppose a user reuses a password of her/his bank account as her/his Facebook password, as long as the password is stored on her/his phone, an attacker could get it and uses it to steal the user's money.

### 2.2 Goals and Scope

Our primary goal is to offer strong protection of passwords on mobile devices. More specifically, the goals are as follows.

1) *No Plaintext of the Full Password on Either Mobile Device or the Cloud Assistant*. The two mutually distrust each other.

2) *No Change to the Server*. The server should not be aware of the entire process of password splitting.

3) *No Change to Applications*. The system should mostly retain backward compatibility to support existing apps.

Note that under our threat model, the attacker is able to obtain secrets in devices' memory and storage. For example, when the user has signed into a web session, the session cookie on a stolen device can be extracted by attackers. Under such scenarios, users need to use short-lived sessions, or disable such sessions from the server side when realizing the device is out of their control. Our approach focuses on preventing attackers from obtaining users' passwords, and general device data protection is out of the scope of this paper.

### 2.3 Background on SSL and TCP Sessions

Most applications nowadays use SSL to protect the communication with the server from eavesdropping or being tampered with. Data are first encapsulated to SSL records and then framed to TCP packets. These protocols maintain the related session and enable the receiver to reassemble the records and packets. The two layers of the session, one on the SSL layer and the other on the TCP layer, bring opportunities as well as challenges to our system. As an opportunity, the sessions provide a standard way for data to be split and merged, which makes it nature for SplitPass to send part of the password from the cloud assistant, which also means that the server does not need to be changed. As a challenge, since the sessions involve many internal states to maintain, the non-trivial task to synchronize these states between the cloud assistant and the mobile device is necessary.

Fig.2(a) shows the CBC (Cipher-Block Chaining) encryption, which is commonly used in SSL. When an application uses SSL to communicate with the server, they establish an SSL session first by negotiating the encryption algorithm, the MAC algorithm, the session key for encryption, etc. During SSL transferring, the data are first divided to multiple fragments, called SSL records, whose size cannot exceed 16 KB. For each SSL record, the SSL library will compress the data (optional), calculate a MAC, and append it to the end of the data. Then, the library will encrypt the data using the chosen algorithm with the session key. The basic data unit of encryption is a 16-byte block. If the application uses CBC for stream encryption, then each plaintext block will XOR its previous ciphertext block before encryption. The first block will use a random number as IV (initial vector). In this way, the integrity of the entire session can be ensured. Any replacement of the data block will be considered as attacks.

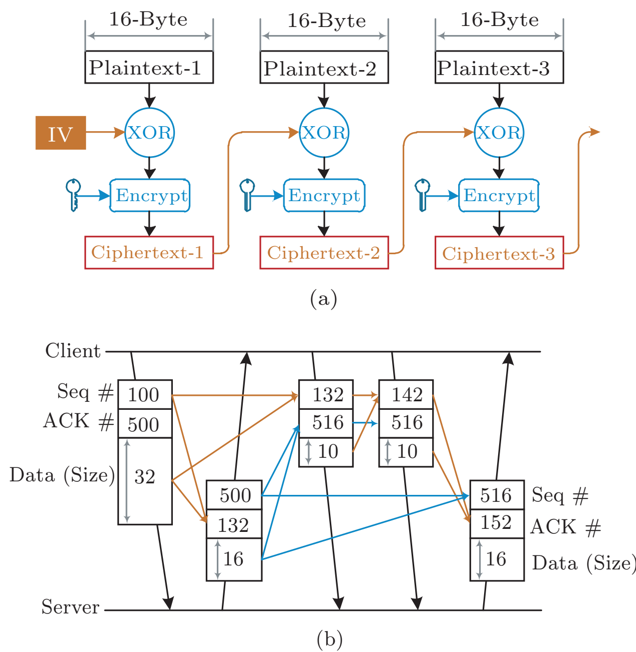


Fig.2. SSL session and TCP session. (a) CBC in SSL session. (b) Sequence number (Seq#) and ACK number (ACK#) in TCP session.

After encryption, an SSL header will be added to each SSL record, and the records will be delivered to the next layer, the TCP layer, for further process. Then the data will go through TCP/IP stack in kernel, be framed at different layers, and may be separated to multiple packets according to the limitation of maximum size, and finally be sent to the network. The session information involves sequence number (Seq #) and ACK number (ACK #) field in the TCP packet header, as shown in Fig.2(b). For each side, the sequence number is the sum of the last sent packet's sequence number and its data size, and the ACK number is the sum of the last received packet's sequence number and its data size. Thus, the order of the packets is ensured by both sides.

Once the server receives these packets, they go through the TCP/IP stack in the opposite direction. The data payload in different packets will be merged, and the merged data is sent up to the user space. Then the SSL layer will decrypt each SSL record, check the MAC, decompress (optional), and merge data from multiple records to a single one, which will be delivered to application logic.

### 2.4 Overview

Fig.3 shows the overview process of cooperative login in SplitPass, as well as all the components involved. In the rest of the paper, we will use the password "abcdef123456" as an example. It is split into two parts: "abcdef" on the mobile device and "123456" on the cloud assistant, which share a unique password ID. The mobile device saves its half of the password together with a placeholder for the other half (we use "XXXXXX" to represent it in this paper), i.e., "abcdefXXXXXX". An app is not aware of SplitPass and uses "abcdefXXXXXX" as the password to send out. When an app uses the password to login, it will send an HTTP request with the password to the SSL layer. The SSL library of SplitPass will identify the password and placeholder, and split the data of the HTTP request into

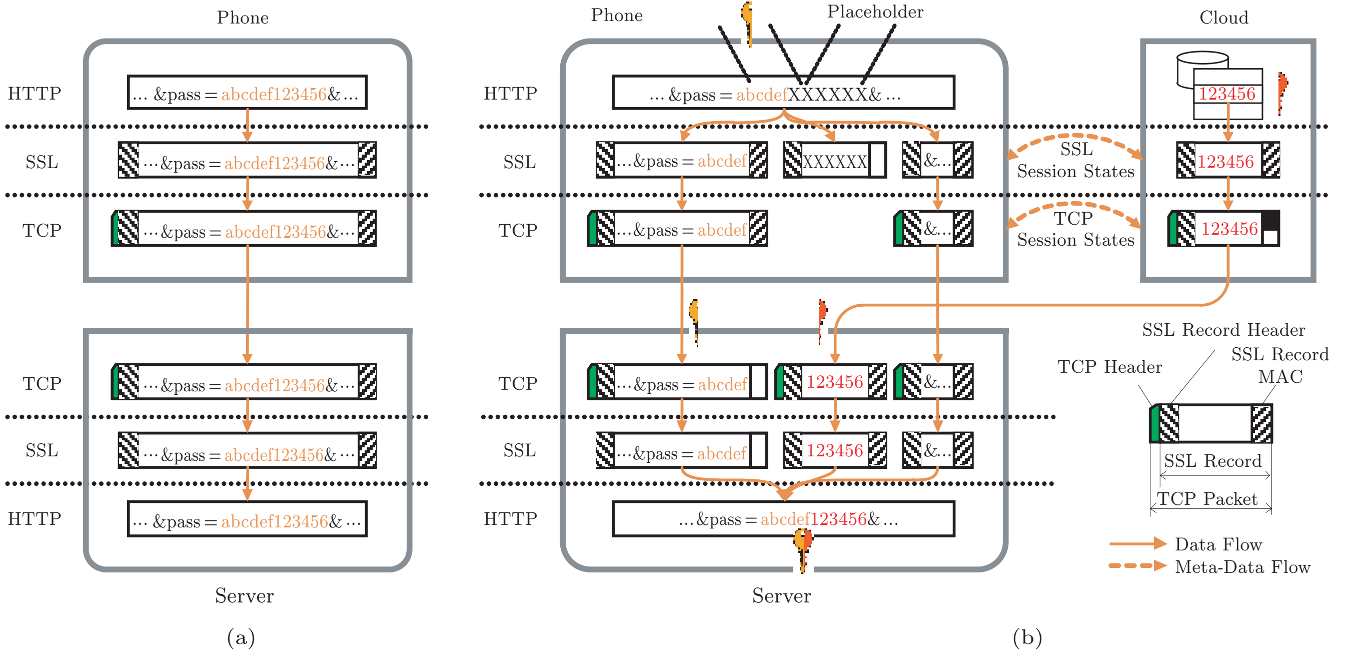


Fig.3. Overview of SplitPass. (a) Traditional system. (b) SplitPass with cloud assistant.

three SSL records: data before placeholder, the placeholder itself, and data after placeholder.

The first record and the third record are directly sent by the mobile device, while the second record will trigger cooperative actions in the cloud assistant to send its portion of the password on behalf of the mobile device. In order to enable the cloud assistant to join the session initiated by the mobile device, it is needed to synchronize the necessary states of the SSL session (e.g., the session key and the encryption method) and some states of the TCP session (e.g., sequence number) between the cloud assistant and the mobile device, so that the cloud assistant is able to generate a packet exactly the same as the one that should be generated by the mobile device, and the mobile device can continue the SSL and TCP sessions afterwards.

The process is completely transparent to the server. It will consider all the packets it received to be sent from the mobile device. The TCP/IP stack and the SSL layer on the server will merge the data automatically since the splitting is done following the protocol of each layer. Once the server gets the entire password, it will do the authentication and rest of the login. The cloud assistant will not get involved after sending the packet containing the second password share.

The most critical part of the cooperative login is the states synchronization between the mobile device and the cloud assistant at both SSL and TCP layers. Other layers, like the IP layer, are stateless and thus do not require state synchronization. In the SSL layer, SplitPass changes the SSL library and develops the SSL session injection technology to retain the integrity of the SSL session between the mobile device and the cloud assistant, which considers different encryption methods. In the TCP layer, in order to avoid changes to TCP/IP stack in the kernel, SplitPass develops the packet reframing technology to synchronize the TCP states between the mobile device and the cloud assistant. These two technologies will be further described in Section 3.

## 3 Design

We have derived inspiration from network routing when designing SplitPass. Our system is divided into two parts: a "control plane" and a "data plane". The control plane is sets of rules that control how the data flows, while the data plane is the mechanism that actually moves the data according to the rules.

More specifically, all the policies of SplitPass' control plane are saved in three tables, as Fig.4 shows. Two of them are saved on the mobile device, named LPT (local password table) and RRT (redirect rule table), and one is saved on the cloud assistant named CPT (cloud password table). These tables are updated when adding or removing passwords. In the data-plane, the password will flow among the mobile device, cloud and server according to these policies of the control plane. In this section, we will first describe the data plane to

demonstrate how the password is split and merged, before showing how to configure the control plane.

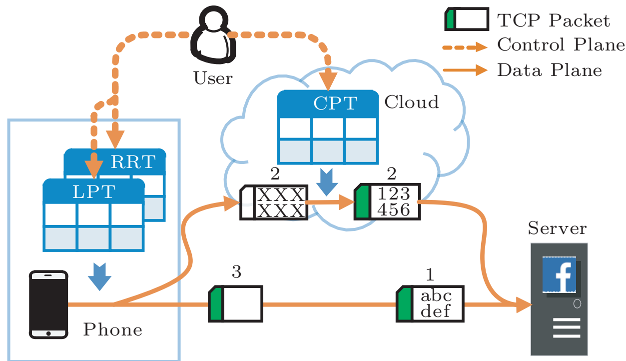


Fig.4. Control plane and data plane of SplitPass.

There are many challenges in the design and implementation, among which the most important one is how to keep SplitPass transparent to the existing system as much as possible. While SplitPass requires no change to the server, the modifications to the software on mobile devices should also be minimized to make the system practical. In SplitPass, we only change the default SSL library on the mobile device. Since most apps use the default library for login, they can use SplitPass transparently.

### 3.1 Data Plane: Cooperative Login

This subsection describes the process of cooperative login with much more details. As mentioned, the mobile device needs to identify and split the password, as well as redirecting the network packet containing the placeholder to the cloud assistant. The cloud assistant first checks the destination address, then reframes the packet with the real half-password, and sends it to the server.

#### *3.1.1 Dataflow on the Phone*

During the login, both the username and the password will be put in an HTTP request in the memory buffer and delivered to the underlying SSL library as shown in Fig.5. The SSL library will check every buffer according to the password table, to find whether the buffer contains the first half of some password and the placeholder. Once found, the SSL library will split the buffer into three sub-buffers. The first sub-buffer is ended with the first half of the password, the second sub-buffer simply contains the placeholder of the second half of the password, and the third one is the rest of the buffer. The three sub-buffers are processed by the SSL library to generate three SSL records, which are later encapsulated into network packets.

The next step is to make the cloud assistant send

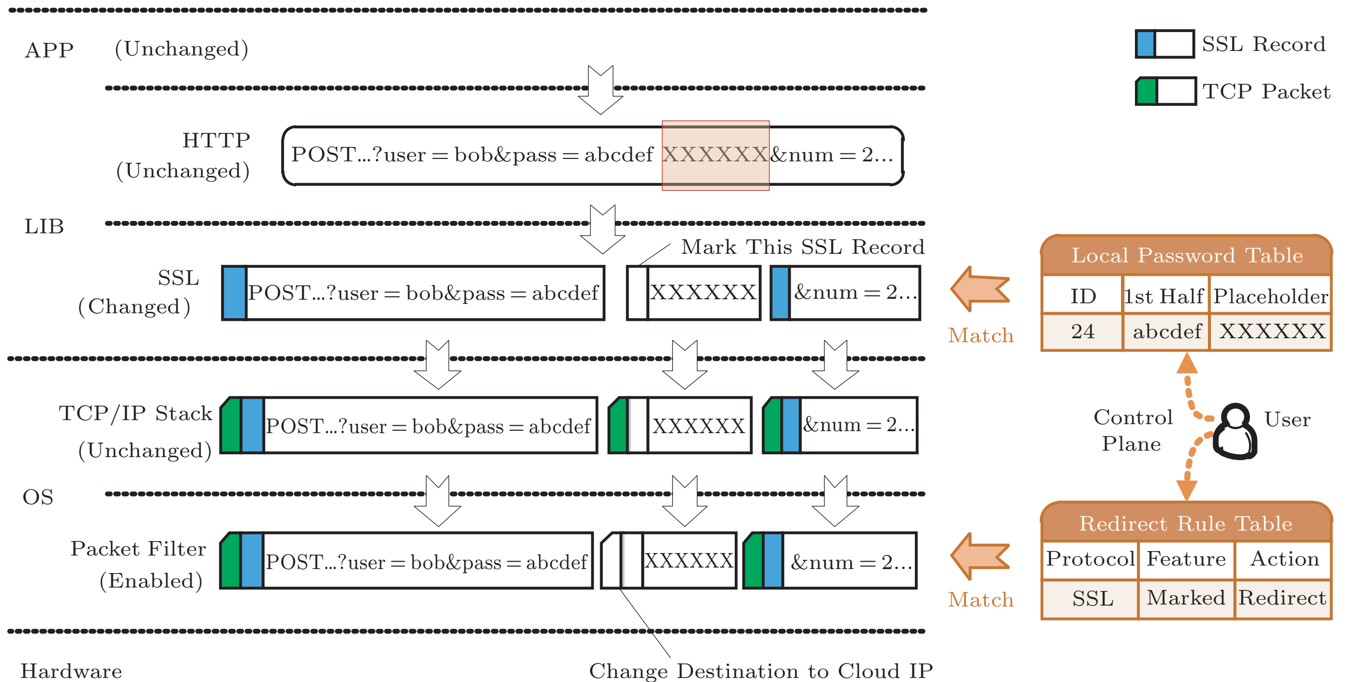


Fig.5. Password splitting and packet routing on the client.

the packet containing the second part of the password to the server on behalf of the mobile device. One naive way is to send all of the necessary states of TCP and SSL sessions to the cloud assistant and make it frame a packet from scratch. However, this solution requires changing the TCP/IP stack on the mobile device to synchronize states like TCP sequence number with the cloud assistant. Such implementation is complex and requires a lot of OS (operating system) hacking.

SplitPass uses a better solution for TCP state synchronization. Instead of skipping the placeholder, the mobile device actually sends it out, thereby there is no need to manually maintain the states in TCP/IP stack. But just before the packet is sent out, SplitPass intercepts the packet with packet filter (which is already supported by Linux kernel, we just enable it) and redirects it to the cloud assistant. In order to do so, SplitPass marks the second SSL record by recording password ID in the type field in the SSL record header, and creates a redirect rule to make the packet filter intercept and redirect the marked packet (more details are in Subsection 3.2).

Meanwhile, the SSL layer sends necessary metadata of the current SSL session to the cloud assistant, which includes the IP address of the server, the password ID, and the SSL internal information like the session key and the encryption method. The cloud assistant needs this information to reframe the redirected packet.

### 3.1.2 *Dataflow on the Cloud Assistant*

Fig.6 shows the dataflow on the cloud assistant. The cloud assistant gets two types of data from the mobile device: the metadata and the redirected packets. Both data contain the password ID. Thus the cloud assistant can pair them and find the record in CPT. It then checks whether the IP in metadata belongs to the domain in the whitelist of the password, to ensure that a password can only be sent to legal server (more details of the whitelist are in Subsection 3.3).

When the checking is done, the cloud assistant will encrypt the second part of the password using the SSL session key and the specific encryption algorithm specified in the metadata. It will generate an encrypted SSL record and replace the payload of the packet. It also changes the destination IP in the TCP header to the server's IP and sends it. The source IP address of the packet is still the mobile device's IP; thus the server will consider the packet as if it is sent by the mobile device.

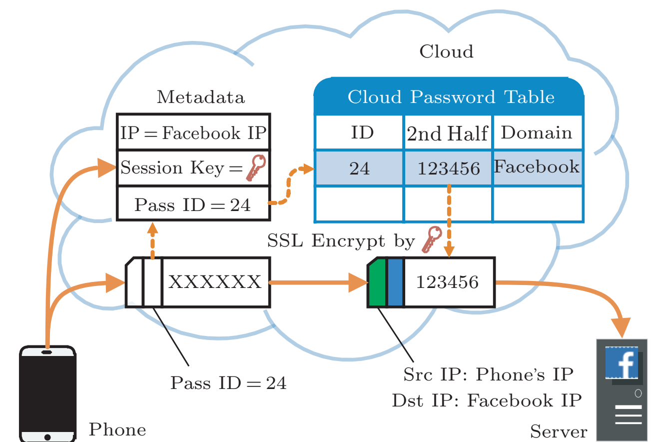


Fig.6. Packet reframing on the cloud assistant.

Finally, the packet with correct substring of the password is sent out by the cloud assistant to the server. The server receives three continuous packets with the mobile device's IP address, and simply combines them into one buffer that contains the complete password. It then proceeds to do authentication, and finally returns the result to the mobile device. The server is not aware of the cloud assistant at all.

The process also works even if the placeholder packet is lost by either the mobile device or the cloud assistant. In that case, the TCP/IP protocol will make the mobile device resend the lost packet, which will be intercepted again by the packet filter and be redirected to the cloud assistant to do the rest.

An alternative design is to leverage MPTCP (Multi-Path TCP)[4], a protocol that is getting more and more popular. MPTCP can greatly simplify the process of TCP session join between the device and the cloud assistant. From a server's perspective, the packet sent by the cloud assistant is from another valid TCP path, and will be merged with other packets sent by the mobile device. Using MPTCP is part of our current work.

### 3.1.3 *SSL Session State Synchronization*

In SplitPass, it is required to migrate a part of SSL session from the device to the cloud assistant. For different stream encryption algorithms (e.g., RC4[5] or CBC[6] as introduced in Subsection 2.3), the states of SSL sessions that need to be synchronized between the mobile device and the cloud assistant are different. For

[4] Multipath TCP. http://en.wikipedia.org/wiki/Multipath_TCP, Dec. 2017.
[5] https://en.wikipedia.org/wiki/RC4, Dec. 2017.
[6] https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#CBC, Dec. 2017.

example, in RC4 algorithm, it only needs to synchronize the metadata of the SSL session, since each ciphertext block is independent from one another. While in CBC algorithms before TLS-1.1 version, it uses the implicit IV mechanism, in which each SSL record uses the last ciphertext block of the previous record as its IV, thus the last ciphertexts of both the mobile device and the cloud assistant are required to be sent back to each other as IV. This process makes the other half of the password insecure.

Take Fig.7(a) as an example, in this situation, after encrypting block-11, the phone is required to send ciphertext-11 to the cloud assistant as IV for encrypting block-12, and the cloud assistant will send back ciphertext-12 to the phone after the encryption. Thus, it is easy for a malicious phone to derive content of block-12 by decrypting the ciphertext of block-12 and then XORing with ciphertext-11:

$$P_{12} = decrypt(C_{12})_{\text{key}} \oplus C_{11}.$$

Here, $P$ stands for plaintext, $C$ stands for ciphertext, and $\oplus$ stands for XOR operation.

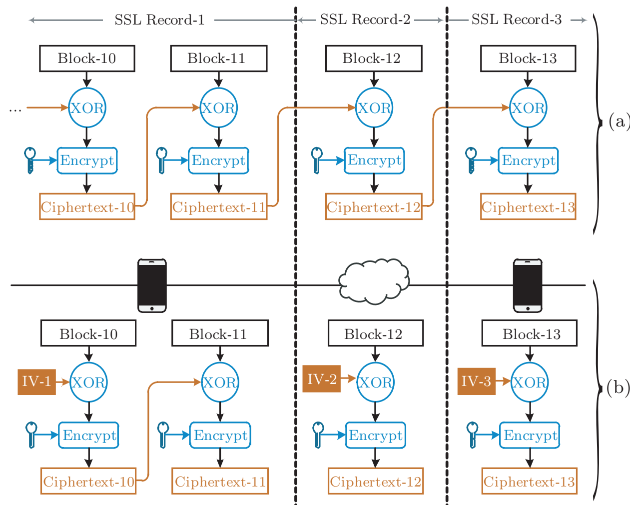


Fig.7. Security issue when using TLS-1.0. An attacker can infer block-12 by block-11 and block-13. (a) CBC with implicit IV (before TLS-1.1). (b) CBC with explicit IV (TLS-1.1 and later verisons).

It means the plaintext of block-12 (the second half of the password) can be derived by XOR the plaintext of ciphertext-12 and ciphertext-11. Now the malicious phone gets the full password.

Re-using the last ciphertext block as IV is known to be insecure[7] for a long time. As a result, from TLS-1.1, each SSL record uses a separated IV, as known as explicit IV. Thus, IV is not needed to be synchronized, as shown in Fig.7(b). We found that before Android 5.0, the default SSL library is TLS-1.0, and from Android 5.0, the default one is TLS-1.2. Thus, we modify Android SSL library to ensure that the SSL version used is newer than TLS-1.0.

## 3.2 Control Plane: Initialization

In the setup phase, the mobile device and the cloud assistant are configured for being paired with each other, as well as for password creating and deleting. The cloud assistant offers four APIs for pairing and operating on passwords, while on the mobile device, we develop an app for configuration. All the configuration information is stored in three tables: CPT, LPT, and RRT. The configuration of the control plane is totally independent with that of the data plane, which means the user can configure the cloud assistant using any device.

### 3.2.1 *Cloud Configuration API*

The four APIs provided by the cloud assistant are listed in Fig.8, and operate the CPT. The user first needs to create a CPT on the cloud assistant through *CREATE_TABLE*(). The cloud assistant will bind the table with the user account. After that, the user can list all the passwords of the CPT (without the plaintext of the password), insert a new password or delete existing ones. Each entry in the CPT contains four fields: ID, the plaintext of the half-password, a whitelist of legal domains that the password could be sent to, and a string description. The domain whitelist is used to limit the destination of the password.

It should be noted that the cloud assistant does not have an *UPDATE_PASS*() API for security reason. If a user does need to update a record, he/she could delete it first and then add a new one. If the cloud assistant offers *UPDATE_PASS*(), an attacker who has controlled the mobile device may use it to add a malicious site to the whitelist of a password, and issue a login using the password. In that case, the cloud assistant's half-password will be sent to the malicious site controlled by the attacker who has already owned the other half-password on the mobile device. In contrast, *ADD_PASS*() and *DELETE_PASS*() do not need to trust the caller, since the former requires a password as a parameter which the attacker does not have, while

[7]Block cipher mode of operation. http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation, Dec. 2017.

**API 1:** *CREATE_TABLE*()**: Creating a Table to Store Password**

1. User initiates a registration request to the cloud assistant using SSL/TLS protocol, provided with his/her self-signed certificate, as well as the account information like username and password.
2. Cloud creates a CPT, and binds it with the user's account and certificate.

**API 2:** *LIST_TABLE*()**: Listing Records of the Cloud Data Table**

1. User initiates a list request to the cloud assistant.
2. Cloud authenticates the user, retrieves all records from the binding operation table, removes all the plaintext of the password.
3. Cloud returns the newly generated records list to the user.

**API 3:** *NEW_PASS*()**: Inserting a Record to the Cloud Password Table**

1. User initiates an insertion request to the cloud assistant, which contains the plaintext of half password, as well as a legal domain that the password is allowed to be sent to.
2. Cloud authenticates the user, inserts the record to the CPT, and generates a unique ID for it.
3. Cloud returns the ID to the user.

**API 4:** *DELETE_PASS*()**: Deleting a Record from the Cloud Password Table**

1. User initiates a deletion request to the cloud assistant with a specific ID of the record.
2. Cloud authenticates the user, deletes the record from the binding operation table.

Fig.8. APIs provided by the cloud assistant.

the latter can only be used to DoS attack which is easy to be discovered by the user.

#### 3.2.2 Mobile-Side Configuration

In order to initialize a password, it is not enough to add only a half-password in the cloud assistant. The other half should be added to the LPT (local password table) on the mobile device. Each record of LPT is composed of the plaintext of the first half of the password, the placeholder of the second half, and the ID of the password which is obtained from the return value of the invocation of *ADD_PASS*() to the cloud assistant. We offer an app on the mobile device to add and remove records to and from LPT. The app also uses *LIST_TABLE*() to get the entire list of passwords stored on the cloud assistant (without the plaintext of the second half).

In order to send the packet with the placeholder to the cloud assistant, SplitPass leverages packet filter to redirect the packet. Thus before using the system, the redirect rules table (RRT) should be initialized. The rule looks like this: if a packet is an SSL packet and is marked (in our implementation, we mark a packet by saving the password ID in the SSL type field in the SSL header), then we change its destination IP to the cloud assistant's IP. If more than one cloud assistant is used, there will be one rule for each cloud in the RRT. Our prototype uses only one cloud assistant, thereby the RRT contains only one rule.

#### 3.2.3 Usability Enhancement: Initializing Both Shares on the Phone

We implement one control app on the mobile device to support secure cloud initialization. Users are required to firstly divide the passwords into two parts to their own liking, and input both parts in the control app, which will then automatically synchronize the second half to the cloud assistant, and eliminate it from the phone afterwards. However this process may introduce some security problems. Consider this scenario: when a user installs a new app and wants to use SplitPass to protect its password, he/she needs to input both the first and the second half of the password into the mobile device. If the mobile device has already been infected by some malware like keylogger, the full password might be stolen.

One possible solution is to use another secure environment (like a PC) to configure the cloud assistant instead of initializing on the mobile device; however, it may reduce the usability of SplitPass. In order to avoid this inconvenience, we extend the control app to use secure hardware, named TrustZone[9], to offer a trust path between the user and the cloud assistant to ensure the privacy and integrity of data along the path. Briefly, TrustZone divides the runtime into se-

cure world and normal world. The control app runs inside the secure world, which is isolated from the normal world that the Android runs in. Any malware in Android cannot access the data of secure world while the control app can access memory or storage of the normal world. The implementation of the trust path is based on our previous work[10] and is out of the scope of this paper.

One question is why not directly use TrustZone to protect apps' passwords instead of using SplitPass. There are mainly three reasons. First, it is not suitable for an app to run inside the secure world, since it may run many other components inside the secure world as well, which will significantly increase the TCB (Trusted Computing Base) and make the secure world not secure. Second, in order to keep the TCB small, the apps have to be changed. For example, a piece of login-related code runs inside the secure world, while the other code runs in the normal world. Thus it cannot keep compatibility with current apps. Third, TrustZone is not physically secure, which means that an attacker can still access the memory of secure world by some physical attack. Thus, an attacker might steal the full password during the process of login.

### 3.3 Whitelist-Based Server Authentication

There are many authentication methods among the mobile device, the cloud assistant, and the web server determined by the different requirements of each of the entities, as well as by the available information they have. From the mobile device's perspective, it authenticates the web server and the cloud assistant by their certificates. The server authenticates the mobile device by users' password. The cloud assistant identifies the mobile device by its certificate, which is bound during initialization. It also needs to recognize the server, since it needs to send half-password to the server. According to our threat model, the cloud assistant does not trust the mobile device. An attacker who has controlled the mobile device may let the cloud assistant send half-password to some malicious server owned by the attacker. Thus, once it receives a request to send half-password to some IP address, the cloud assistant must decide whether the IP is secure for the password.

SplitPass solves this problem by using a whitelist of domains for each half-password, and makes sure that the half-password could be sent to an IP that belongs to one of the domains within the whitelist. However, the granularity of the domain sometimes can be too coarse-grained. For example, if an attacker controls a mobile device, even with domain-level whitelist enforcement, he/she can still steal the user's Facebook password by sending it to his/her own Facebook page as a comment. Since the target IP is within the Facebook's domain, the cloud assistant will not block the network.

Fortunately, we observe that most well-known web sites (e.g., Google, Facebook, Linkedin) have dedicated machines to do the authentication, which means the authentication IP addresses are different from others. Thus, in the cloud assistant, the whitelist granularity can be further optimized to only allow IPs that are responsible for authentication. When such IP addresses are changed by the server, the cloud assistant can update the whitelist automatically without users' intervention. Meanwhile, most of the web sites, like bank sites, do not support users to post contents.

## 4 Security Analysis

In this section, we discuss the security of SplitPass under two attacking scenarios: the mobile device is physically controlled, and the cloud assistant is fully compromised. We also discuss the limitations of SplitPass.

### 4.1 Scenario 1: Fully-Compromised Phone

Once the mobile device is physically controlled by an attacker, the portion of the password on the mobile device is leaked, as well as the certificate of the mobile device. Thus, the attacker will try to issue attacks from the mobile device to the cloud assistant to get the second half. Since the cloud assistant will never send its part of the password to the mobile device, the attacker must use other ways to attack.

First, the attacker may try to infer the cloud portion of the password from the syncing states. Such an attack will fail, as we have discussed in Subsection 3.1.3.

Second, the attacker may try to fool the cloud assistant and let it send the cloud portion to a malicious server controlled by the attacker. This attack will not succeed since the cloud assistant will check the operation table to ensure the destination server is in one of the domains of the password's whitelist. Otherwise, the cloud assistant will refuse to send the password.

Third, the attacker may try to modify the operation table, by deleting all of the entries of the cloud password table. We use backup mechanism on the cloud to defend against such a DoS attack. The attacker may also insert malicious entries into the table, with faulty

ID or/and domain whitelist. All these behaviors can neither steal the second part of the password directly nor send it to some malicious sites.

The length of the placeholder can be different from the length of the second half-password, which means that the mobile device does not know the length of the half-password saved on the cloud assistant. SplitPass only depends on the fact that the length of the TCP packet generated from the placeholder on the mobile device is the same with the one generated from the second half-password on the cloud assistant, in order to keep the sequence number not changed in the redirected TCP packet. Since the encryption in SSL is done on the unit of block, padding will be added to keep data aligned. For example, if the length of the second half-password is 8 bytes, and its placeholder's length is 6 bytes, they will generate SSL records with the same size and also same-sized TCP packets.

### 4.2 Scenario 2: Fully-Compromised Cloud

Once the cloud assistant is fully controlled by attackers or malicious operators, the entire cloud password table is leaked, as well as the metadata including the session key of current SSL session on the mobile device. Then the attacker will try to steal the mobile portion of the password. However, since the cloud assistant only takes command from the mobile device, there is no way for an attacker to issue attacks through our protocol.

Although the attacker can get the SSL session key, it cannot get any other data in the same session, and the key will be useless when the session is time-out.

One possible side-channel attack is that an attacker may get the information of the time and frequency of users' login. However, there is no direct way for an attacker to know the user's identity. Thus such information will be less (if any) useful in other attacks.

### 4.3 Scenario 3: Man-in-the-Middle Attack

MITM (Man-in-the-Middle) attacks can happen between the device and the cloud assistant, between the cloud assistant and the server, and between the device and the server. Since each pair uses (independent) SSL session to protect their communication, an attacker cannot directly get plaintext by an MITM attack.

One case is that an attacker who has already gained full control over the mobile device issues an MITM attack between the cloud assistant and the server. Since the attacker has the session key from the device, he/she can decrypt the SSL communication and get the second half-password. However, it is not trivial to issue an MITM attack between the cloud assistant and the server, and we can slightly extend our design to further increase the difficulty. One extension is that once generating a reframed packet, the cloud assistant will first send it to some random node also within the cloud, which then sends the packet to the server. The random node works as a relay. The communication between the two cloud nodes is protected by a new SSL session to defend against MITM attacks. Since it is hard for an attacker to predict where the random node will be, such a design increases the difficulty of eavesdropping.

Another case is that the attacker compromises the cloud assistant, and tries to set up an MITM attack between the mobile device and the server. However, it is difficult to locate the users' devices and redirect users' connection to the attacker's server.

### 4.4 Scenario 4: Phishing Attacks and Others

SplitPass can defend against phishing attacks. Even if a user opens a phishing web site and enters the password (which is the first half and the placeholder for the second half), the malicious site can only get the mobile portion of the password. Because once the mobile device requires the cloud assistant to send its portion, the cloud assistant will check the whitelist of the password and refuse to send its portion of the password to the phishing site. Similarly, if an attacker uses attacks listed in [2-3], the cloud assistant will also reject to send its password share to the malicious sites.

### 4.5 Discussion and Limitations

*User Impersonating Attack.* One possible scenario is that an attacker steals the phone and pretends to be the user to access the password to login. This problem is about how to authenticate the user and is orthogonal to the problem of password protection that SplitPass addresses, which is out of the scope of this paper. There are various methods for user authentication, e.g., by using PIN code, fingerprints, users' faces. We could simply extend SplitPass to leverage such authentication mechanisms so that both the phone and the cloud assistant can check current user's identity every time when a password is accessed. The basic idea is to let the server send a nonce to the device, which requires users to input their fingerprints. The fingerprint scanner will check the fingerprint, then encrypt the nonce with a private key, and send it to the cloud assistant.

The cloud assistant will check the nonce with the corresponding public key. The privacy and public keys are deployed during initialization, and the privacy key is stored on device in sealed storage like TPM, which can defend against physical attacks. If the device does not have a fingerprint scanner, it can use PIN code and similar protocol implemented in secure hardware like TrustZone. Even if the device does not use any of the methods, SplitPass can still offer better password protection than any current password managers.

*Deployability.* In our experience of deploying SplitPass to real cloud environments, we find that some cloud providers treat packet reframing as IP spoofing, which makes the reframed packet be filtered by firewall, while some other cloud platforms do not. SplitPass requires the cloud to allow to send the reframed packets. A user can either choose an available public cloud, or build his/her own private cloud to do so. Another way is to use another machine or mobile device as a relay, as described above, so that the cloud itself does not deliver the packet directly. Meanwhile, using MPTCP can also mitigate the IP spoofing problem, as we mentioned in Section 3.

*User Experience Issues.* Before using SplitPass in an app, the user needs to initialize two parts of the password on the mobile device and the cloud assistant, which might increase the burden of using. We argue that such initialization is a one-time effort and can be done in a batch. Meanwhile, the password initialization has already be automated by the control app in the mobile device as explained in Subsection 3.2, so that the user only needs to input the two halves of the password and the placeholder of the second half into the textboxes provided by the control app. For password entering during login, the user is required to input the first half plus the placeholder of the second half (as shown in Subsection 2.4). This does not influence the users' experience. If the user does not want to use SplitPass protection, he/she can still choose to input the complete password to bypass the SplitPass process.

*Generalization.* SplitPass is a general framework that can be used not only for mobile devices, but also for other environments such as desktop PC and laptop, as long as the server uses the password for authentication. Our system is also not limited to use only two nodes. If users require more secure solutions, they can split the password into more than two parts and use more cloud assistants accordingly.

## 5 App Compatibility Evaluation

In this section, we evaluate the compatibility of SplitPass with existing Android apps. SplitPass requires that the apps should use Android default OpenSSL library, and they should send the password to SSL in plaintext so that SplitPass is able to match and replace. We pick up 100 Android apps from the top of each category to check how many apps satisfy these two requirements. Meanwhile, we also analyze the cipher algorithm and version used.

### 5.1 Compatibility Evaluation

We choose 100 apps from the top free apps in different categories on the Google Play, which require authentication. We analyze which encryption algorithm and version they use, whether they use the default SSL library (OpenSSL), and whether they send the password in plaintext to the SSL layer. From Table 1, we can conclude that most of the apps (98 out of 100) use Android default SSL and CRYPTO libraries, while in the other two apps (i.e., HealthKartPlus, and TNP), we cannot intercept any record in the SSL layer. Among

**Table 1**. Apps Usage of SSL and CRYPTO Libraries, SSL/TLS Version, Cipher Algorithm in Android 4.1.2

| App | # | Use Default | Password | TLS Version | | | Encryption Algo. | | | Hash Algo. | | Supported |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Category | | Libraries | Invisible | 1.0 | 1.1 | 1.2 | RC4 | AES128 | AES256 | SHA | MD5 | |
| Social | 20 | 20 | 1 | 20 | 0 | 0 | 12 | 5 | 3 | 17 | 3 | 19 |
| Communication | 20 | 20 | 4 | 20 | 0 | 0 | 15 | 3 | 2 | 15 | 5 | 16 |
| Productivity | 10 | 10 | 1 | 9 | 0 | 1 | 5 | 1 | 4 | 9 | 1 | 9 |
| Finance | 10 | 10 | 0 | 10 | 0 | 0 | 6 | 3 | 1 | 9 | 1 | 10 |
| Music & audio | 10 | 10 | 0 | 10 | 0 | 0 | 8 | 2 | 0 | 9 | 1 | 10 |
| News & magazines | 10 | 9 | 2 | 9 | 0 | 0 | 4 | 4 | 1 | 7 | 2 | 7 |
| Shopping | 10 | 10 | 0 | 10 | 0 | 0 | 4 | 5 | 1 | 9 | 1 | 10 |
| Medical | 5 | 4 | 0 | 4 | 0 | 0 | 1 | 2 | 1 | 4 | 0 | 4 |
| Photography | 5 | 5 | 0 | 5 | 0 | 0 | 4 | 0 | 1 | 3 | 2 | 5 |
| Summary | 100 | 98 | 8 | 97 | 0 | 1 | 59 | 25 | 14 | 82 | 16 | 90 |

Note: Algo.: algorithm; #: the number of tested applications.

these 98 apps, there are eight apps (e.g., Skype, Evernote) that may do some hashing of the password before sending it to the SSL library, so that the SplitPass mechanism cannot support them since we need the plaintext of the first half of password to identify specific SSL record. Thus, SplitPass can support 90 out of the 100 apps without any modification.

### 5.2 SSL Usage Summary and Discussion

Another interesting topic is how most Android apps use SSL and CRYPTO libraries. The analysis done by Fahl *et al.*[11] revealed that there are various forms of SSL/TLS misuse in current Android apps, and about 8% apps examined contain SSL/TLS code that is potentially vulnerable to MITM attacks. In our evaluation we find that even if there is no misuse caused by developers, there are still some security issues in the phase of SSL version and cipher algorithm selection.

In Table 1, we use TLS 1.0 to TLS 1.2 to represent SSL 3.1 to SSL 3.3, respectively. For the encryption algorithm classification, RC4 is a widely used stream cipher that uses a pseudo-random keystream to encrypt plaintext using bit-wise exclusive-or. AES128 and AES256 are other two kinds of block ciphers used by some of the mobile apps, with 128-bit key and 256-bit key, respectively. In our experiment, the AES cipher algorithm uses CBC encryption by default, as described in Subsection 3.1.3. The CBC encryption algorithms can be classified into two categories: one with implicit IV and its padding must be less than the cipher's block length (TLS 1.0), and the other with explicit IV and its padding can be any integral multiple of the block cipher's block length, up to 255 bytes (TLS 1.1 and above).

From Table 1, we find that there are 59 apps using RC4 encryption algorithm, which is considered problematic since 2002[12], and among the 39 apps which use AES CBC algorithms, 38 of them are using TLS 1.0. In TLS 1.0, CBC uses implicit IV and its padding must be less than the cipher's block length, where there is a well-known theoretical client-side SSL attack called BEAST⑧ (CVE-2011-3389) targeting at it. Among these 100 apps, we find that only Dropbox uses AES CBC algorithms with TLS 1.2 version.

In order to figure out why so many applications use RC4 encryption algorithm, we intercept the network flow in SSL Handshake phase and find that the CipherSuite list⑨⑩ puts RC4-MD5 cipher first, and uses the order like: RC4-MD5 > RC4-SHA > AES128-SHA > AES256-SHA. That is why most of the app servers actually choose RC4 as their encryption algorithm, even if it is the most unsecure one.

## 6 Performance Evaluation

In this section, we first select representative apps from the 90 supported apps to evaluate their end-to-end login time overhead and additional network traffic in both Wi-Fi and 3G network environment. We also measure the daily use performance overhead, including network latency and power consumption.

The mobile part of SplitPass is implemented on a Samsung Galaxy Nexus smartphone, with one 1.2 GHz TI OMAP4460 CPU, 1 GB memory, 16 GB internal storage and one 1 750 mAh battery. The cloud assistant part of SplitPass is deployed in a PC with 2.8 GHz Intel i5-2300 quad-core CPU, 8 GB memory, 500 GB disk, and 100/1 000 Mbps NIC. The smartphone is with Android 4.1 installed as the original system, and the cloud assistant is with Linux kernel 3.13.7. It is noted that the mechanisms used in SplitPass do not rely on the hardware or Android version, and the main factors to consider are the algorithms used in the SSL record generation.

### 6.1 End-to-End Performance Overhead

To measure the latency of the login process of different apps, we select five apps as shown in Table 2, among which each one represents one combination of the cipher algorithm and SSL/TLS version (ver.). The end-to-end performance evaluation is conducted under both Wi-Fi and 3G network, using the original Android and SplitPass.

**Table 2.** Selected Apps for End-to-End Performance Evaluation

| App Name | Cipher Algo. | TLS Ver. | Metadata (Byte) |
|---|---|---|---|
| Bankdroid | RC4-SHA | v-1.0 | 502 |
| Ask.fm | RC4-MD5 | v-1.0 | 494 |
| Instagram | AES128-SHA | v-1.0 | 488 |
| Tumblr | AES256-SHA | v-1.0 | 488 |
| Dropbox | AES256-SHA | v-1.2 | 488 |

⑧BEAST Attack on client-side SSL. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3389, Dec. 2017.

⑨The TLS Protocol Version 1.0, Client Hello Section. http://tools.ietf.org/html/rfc2246#section-7.4.1.2, Dec. 2017.

⑩CipherSuite list is a list passed from the client to the server in the client hello message, which contains the combinations of cryptographic algorithms supported by the client in the order of the client's preference (favorite choice first).

The measurements of SplitPass are done after the one-time effort initialization (i.e., the cloud CPT, mobile LPT and RRT have already been configured). The results are shown in Figs.9 and 10. The overall overhead is less than 10% in both Wi-Fi and 3G network environment, which is imperceptible.

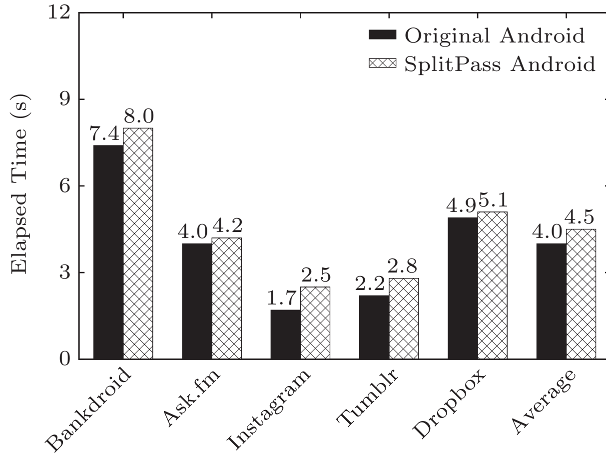


Fig.9. Apps login time evaluation in Wi-Fi network environment after warming up.

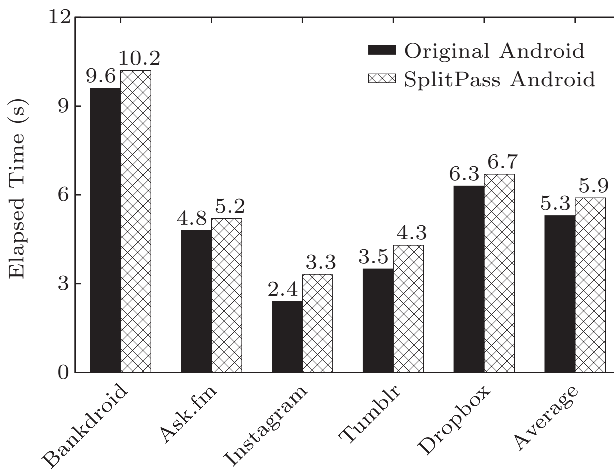


Fig.10. Apps login time evaluation in 3G network environment after warming up.

The additional traffic between the mobile device and the cloud assistant is equal to or less than 520 bytes for most of the apps, as shown in Table 2. The additional network traffic caused by the SplitPass mechanism is mainly due to the cipher metadata. For encryption algorithm, the size of required data structure is 258 bytes for RC4, 244 bytes for AES, and 16 bytes for each IV. For hashing algorithm, the size of metadata context is 192 bytes for SHA, and 184 bytes for MD5. We also send some other metadata, e.g., the IP address of the server, the password ID, and the corresponding encryption method, to the cloud assistant.

## 6.2 Daily Use Performance Overhead

The performance overhead of SplitPass mainly comes from two sources: the string matching in the SSL layer and the packet matching for filtering. Thus, we focus on how these operations on the phone will affect the network traffic latency and cause additional power consumption in mobile devices for daily use. In order to better illustrate the performance overhead caused by the string searching and matching, we conduct the experiment to test the latency of normal network traffic (without password authentication), as well as power consumption overhead.

### *6.2.1 Network Traffic Latency*

The additional latency of normal network traffic caused by SplitPass mechanism is mainly due to buffer matching and searching in the SSL library, as well as the packet filter enabled in the kernel. We conduct a stress test in the system with and without SplitPass mechanism enabled. Since this experiment is tested on normal network traffic other than login, there is only overhead of matching, but no splitting or redirecting. We write a test app to send 1 000 HTTPS requests and receive replies, and calculate the total time. For the original system, the total latency is about 4.9 seconds, while using SplitPass, the latency is about 5.2 seconds.

### *6.2.2 Power Consumption*

To test the power consumption impact on the login process, we consecutively run PayPal login for 30 minutes in both the original Android system and SplitPass, and read the battery every second. As shown in Fig.11, after 30 minutes, the Android system has 94% battery left, while SplitPass has 92%.

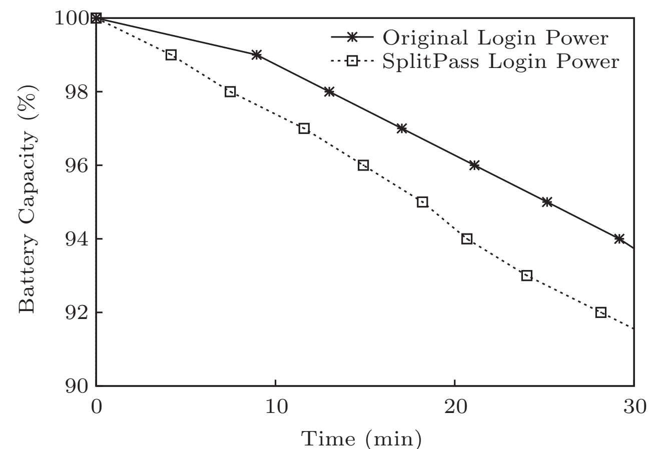


Fig.11. Battery level changing running stress test on login operation for 30 minutes.

We also evaluate the power consumption of operations other than login. To trigger string matching and packet filtering, we simply skim web pages with HTTPS protocol for 30 minutes and record the battery level every second. We perform the test on Android as well as using SplitPass, and the results are shown in Fig.12. The curves indicate that SplitPass only occurs small amount of power overhead.

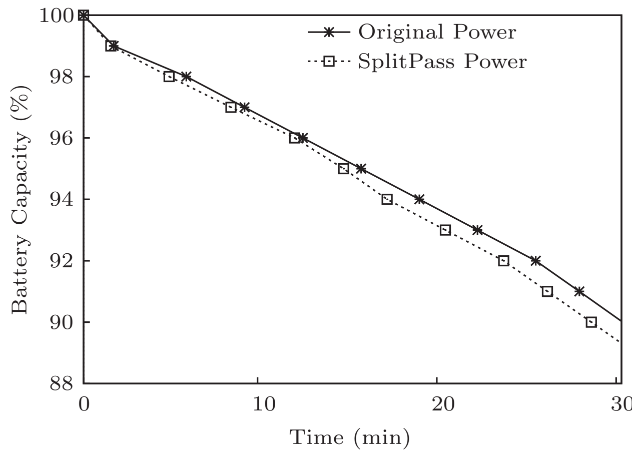


Fig.12. Battery level changing surfing HTTPS web pages for 30 minutes, without login operations.

## 7 Related Work

*Password Manager.* Password security has been extensively studied[13-14], and there are many types of password manager. Local password managers like 1password⑪, Mozilla Firefox, and Apple Keychain, store passwords on users' device. Online password managers like LastPass, Mozilla Weave Sync⑫, Tapas[4], and PIN-Audio[15], store passwords on other device, either a remote server or a portable device.

McCarney *et al.* proposed Tapas[4], a password manager that applies dual-possession authentication. Two independent devices are involved, named Manager (typically a desktop PC) and Wallet (e.g., a phone). The password is encrypted and saved in Wallet while the key is held by Manager. Thus, either device lost will not cause the password leakage. During login process, Manager asks the ciphertext of password from Wallet, decrypts it, and sends the password to the server. Tapas is not designed for mobile devices. Manager and Wallet cannot be installed on one device otherwise if the device is lost the password will be leaked. Thus, if a user wants to login Facebook on his/her phone (as Manager), he/she must carry another device as Wallet, which is not practical. Meanwhile, during the process, the password is in plaintext in memory, which might be stolen by malware.

*Two-Factor Authentication.* TFA (two-factor authentication) technology is an effective way to enhance the security of password by using a hardware or software token in addition to the password, like RSA SecureID⑬, Google Authentication⑭, and PhoneAuth[16]. Unlike TFA, SplitPass does not require users to go through two steps, but just use one password as usual. It also requires no new hardware like a token or another mobile device. Furthermore, the servers do not have to change to use SplitPass.

*Cloud-Based Security Services.* SplitPass continues the line of research of leveraging a cloud to enhance mobile platforms[17-18], especially for security[5,19-25]. Paranoid Android[26] mirrors the phone in the cloud using record and replay mechanisms, so that it can protect the phone by doing cloud security enforcement. SmartSiren[21], CloudAV[22], and ThinAV[23] are those systems to offer cloud-based antivirus service. Mackenzie and Reiter[20] leveraged capture-resilient cryptography to provide cloud-based authentication. TinMan[27] is a system that also protects critical data like password on mobile devices. Like SplitPass, TinMan introduces a remote node to save and access the password, and only saves placeholders on the device instead of password plaintext. As long as an app needs to access the password, the system will migrate the entire app to the remote node to continue execution, thereby the password will never appear on the mobile device. TinMan has a different threat model with SplitPass. It fully trusts the remote node which stores and accesses the passwords. In SplitPass, the cloud and the mobile device mutually distrust each other.

*Mobile Data Protection.* There are many researches on protecting critical data on mobile devices[28-29]. Keypad[19] and CleanOS[5] leverage cloud as backend service to encrypt and store critical data of mobile devices. $\pi$Box[30] extends Android platform to prevent applications from mis-using users' private information, including the password, by using sandbox spanning users' device and a cloud. SpanDex[31] and Pebbles[32]

⑪1password. http://1password.com, Dec. 2017.

⑫Mozilla Labs. Weave Sync. http://labs.mozilla.com/projects/weave, Dec. 2017.

⑬RSA SecureID: Worlds Leading Two-Factor Authentication. http://www.emc.com/security/rsa-securid.htm, Dec. 2017.

⑭Google Authenticator for Two-Step Verification. http://code.google.com/p/google-authenticator/, Dec. 2017.

track the flow of critical data to enforce the protection. DroidVault[33] leverages hardware features (e.g., TrustZone) to enforce strong control over the sensitive data in the mobile with minimal TCB. These systems focus on enhancing security and attack detection on mobile devices. However, there would still be residue of critical data left on devices which could be stolen by attackers.

*Memory Encryption: Decrypting In-Use.* CryptKeeper[34] encrypts all memory pages except for a small working set. Similarly, CleanOS[5] encrypts sensitive data on the local device and keeps the key on the trusted cloud. Such data only gets decrypted when it is accessed, and the plaintext is evicted after a specified period of non-use by the garbage collector. Although CleanOS can significantly reduce the exposure time of sensitive data, it must trust all of the hardware and software on the device. Keypad[19] is an auditing file system that provides file-level auditing. All the critical files are encrypted locally, and the keys are stored in a trusted cloud. The files are not decrypted unless they are accessed. These systems cannot defend against rootkit malware that is able to monitor the memory and steal the plaintext of password from memory when being used. In SplitPass, the second half of the password never exists on the device and thus cannot be stolen by such malware.

## 8 Conclusions

Typing passwords is unpleasant for users, especially when they are using mobile devices with small screens. However, it is not safe to save them on local storage of the mobile devices since these devices are prone to getting lost or stolen; thus an adversary may issue all kinds of attacks to steal data from the storage and memory. It is also not safe to store them on a cloud since the cloud itself may be attacked. In this paper, we present SplitPass, which splits a password into two parts, one on a cloud assistant and the other on the device. It ensures that the cloud assistant or the device will never get the full password. In order to avoid server change, SplitPass adopts two technologies to make the server not aware of the splitting of password. SplitPass also minimizes the modification requirement of apps, and supports unmodified apps if they use the system's SSL library, which is the common case. In the future, we intend to continue the development of SplitPass to make it support more types of credential data beside the passwords.

## References


[1] Bonneau J, Herley C, van Oorschot P C, Stajano F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. IEEE Symp. Security and Privacy* (*SP*), July 2012, pp.553-567.

[2] Silver D, Jana S, Boneh D, Chen E, Jackson C. Password managers: Attacks and defenses. In *Proc. the 23rd USENIX Conf. Security Symp.*, August 2014, pp.449-464.

[3] Li Z W, He W, Akhawe D, Song D. The emperor's new password manager: Security analysis of web-based password managers. In *Proc. the 23rd USENIX Conf. Security Symp.*, August 2014, pp.465-479.

[4] McCarney D, Barrera D, Clark J, Chiasson S, van Oorschot P C. Tapas: Design, implementation, and usability evaluation of a password manager. In *Proc. the 28th Annual Computer Security Applications Conf.*, December 2012, pp.89-98.

[5] Tang Y, Ames P, Bhamidipati S, Bijlani A, Geambasu R, Sarda N. Cleanos: Limiting mobile data exposure with idle eviction. In *Proc. the 10th USENIX Conf. Operating Systems Design and Implementation*, October 2012, pp.77-91.

[6] Müller T, Spreitzenbarth M. FROST. In *Applied Cryptography and Network Security*, Jacobson M, Locasto M, Mohassel P, Safavi-Naini R (eds.), Springer 2013, pp.373-388.

[7] Zhang F Z, Chen J, Chen H B, Zang B Y. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. the 23rd ACM Symp. Operating Systems Principles*, October 2011, pp.203-216.

[8] Das A, Bonneau J, Caesar M, Borisov N, Wang X F. The tangled web of password reuse. In *Network and Distributed System Security Symp.*, February 2014, pp.23-26.

[9] Alves T, Felton D. Trustzone: Integrated hardware and software security. *ARM White Paper*, 2004, 3(4): 18-24.

[10] Li W H, Ma M Y, Han J C, Xia Y B, Zang B Y, Chu C K, Li T Y. Building trusted path on untrusted device drivers for mobile devices. In *Proc. the 5th Asia-Pacific Workshop on Systems*, June 2014.

[11] 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 *Proc. the ACM Conf. Computer and Communications Security*, October 2012, pp.50-61.

[12] Mantin I, Shamir A. A practical attack on broadcast $RC_4$. In *Fast Software Encryption*, Matsui M (ed.), Springer, 2002, pp.152-164.

[13] Morris R, Thompson K. Password security: A case history. *Communications of the ACM*, 1979, 22(11): 594-597.

[14] Zhang Y Q, Monrose F, Reiter M K. The security of modern password expiration: An algorithmic framework and empirical analysis. In *Proc. the 17th ACM Conf. Computer and Communications Security*, October 2010, pp.176-186.

[15] Saxena N, Voris J. Exploring mobile proxies for better password authentication. In *Information and Communications Security*, Chim T W, Yuen T H (eds.), Springer, 2012, pp.293-302.

[16] Czeskis A, Dietz M, Kohno T, Wallach D, Balfanz D. Strengthening user authentication through opportunistic cryptographic identity assertions. In *Proc. the ACM Conf. Computer and Communications Security*, October 2012, pp.404-414.

[17] Satyanarayanan M, Bahl P, Caceres R, Davies N. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 2009, 8(4): 14-23.

[18] Gordon M S, Jamshidi D A, Mahlke S, Mao Z M, Chen X. COMET: Code offload by migrating execution transparently. In *Proc. the 10th USENIX Conf. Operating Systems Design and Implementation*, October 2012, pp.93-106.

[19] Geambasu R, John J P, Gribble S D, Kohno T, Levy H M. Keypad: An auditing file system for theft-prone devices. In *Proc. the 6th Conf. Computer Systems*, April 2011.

[20] MacKenzie P, Reiter M K. Networked cryptographic devices resilient to capture. *Int. Journal of Information Security*, 2003, 2(1): 1-20.

[21] Cheng J, Wong S H Y, Yang H, Lu S W. SmartSiren: Virus detection and alert for smartphones. In *Proc. the 5th Int. Conf. Mobile Systems, Applications and Services*, June 2007, pp.258-271.

[22] Oberheide J, Cooke E, Jahanian F. CloudAV: *N*-version antivirus in the network cloud. In *Proc. the 17th Conf. Security Symposium*, August 2008, pp.91-106.

[23] Jarabek C, Barrera D, Aycock J. ThinAV: Truly lightweight mobile cloud-based anti-malware. In *Proc. the 28th Annual Computer Security Applications Conf.*, December 2012, pp.209-218.

[24] Puttaswamy K P N, Kruegel C, Zhao B Y. Silverline: Toward data confidentiality in storage-intensive cloud applications. In *Proc. the 2nd ACM Symp. Cloud Computing*, October 2011.

[25] Satyanarayanan M, Lewis G, Morris E, Simanta S, Boleng J, Ha K. The role of cloudlets in hostile environments. *IEEE Pervasive Computing*, 2013, 12(4): 40-49.

[26] Portokalidis G, Homburg P, Anagnostakis K, Bos H. Paranoid Android: Versatile protection for smartphones. In *Proc. the 26th Annual Computer Security Applications Conf.*, December 2010, pp.347-356.

[27] Xia Y B, Liu Y T, Tan C, Ma M Y, Guan H B, Zang B Y, Chen H B. TinMan: Eliminating confidential mobile data exposure with security oriented offloading. In *Proc. the 10th European Conf. Computer Systems*, April 2015, Article No. 27.

[28] Zhu S W, Lu L, Singh K. CASE: Comprehensive application security enforcement on COTS mobile devices. In *Proc. the 14th Annual Int. Conf. Mobile Systems, Applications, and Services*, June 2016, pp.375-386.

[29] Huang Y, Chapman P, Evans D. Privacy-preserving applications on smartphones. In *Proc. the 6th USENIX Workshop on Hot Topics in Security*, August 2011.

[30] Lee S, Wong E L, Goel D, Dahlin M, Shmatikov V. $\pi$Box: A platform for privacy-preserving apps. In *Proc. the 10th USENIX Conf. Networked Systems Design and Implementation*, April 2013, pp.501-514.

[31] Cox L P, Gilbert P, Lawler G, Pistol V, Razeen A, Wu B, Cheemalapati S. SpanDex: Secure password tracking for Android. In *Proc. the 23rd USENIX Conf. Security Symposium*, August 2014, pp.481-494.

[32] Spahn R, Bell J, Lee M Z, Bhamidipati S, Geambasu R, Kaiser G. Pebbles: Fine-grained data management abstractions for modern operating systems. In *Proc. the 11th USENIX Conf. Operating Systems Design and Implementation*, October 2014, pp.113-129.

[33] Li X L, Hu H, Bai G D, Jia Y Q, Liang Z K, Saxena P. DroidVault: A trusted data vault for Android devices. In *Proc. the 19th Int. Conf. Engineering of Complex Computer Systems* (*ICECCS*), August 2014, pp.29-38.

[34] Peterson P A H. Cryptkeeper: Improving security with encrypted RAM. In *Proc. IEEE Int Conf. Technologies for Homeland Security* (*HST*), November 2010, pp.120-126.

**Yu-Tao Liu** received his B.S. degree in computer science from Fudan University, Shanghai, in 2012. He is currently a Ph.D. candidate of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a member of CCF and IEEE. His research interests include virtualization, system security, and mobile security.

**Dong Du** is currently an undergraduate student of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. His research interests include virtualization and system security.

**Yu-Bin Xia** received his B.S. degree in computer science from Fudan University, Shanghai, in 2004, and Ph.D. degree in computer science from Peking University, Beijing, in 2010. He is currently an associate professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a member of CCF, ACM and IEEE. His research interests include virtualization, computer architecture and system security.

**Hai-Bo Chen** received his B.S. and Ph.D. degrees in computer science from Fudan University, Shanghai, in 2004 and 2009, respectively. He is currently a professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a distinguished member of CCF, and a senior member of ACM and IEEE. His research interests include software evolution, system software, and computer architecture.

**Bin-Yu Zang** received his Ph.D. degree in computer science from Fudan University, Shanghai, in 1999. He is currently a professor and the director of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a distinguished member of CCF, and a member of ACM and IEEE. His research interests include compilers, computer architecture, and systems software.

**Zhenkai Liang** received his B.S. degree from Peking University, Beijing, in 1999, and Ph.D. degree from Stony Brook University, New York City, in 2006. He is currently an associate professor of the School of Computing, National University of Singapore, Singapore. He is a member of ACM and IEEE. His main research interests are in system and software security, web security, mobile security, and program analysis.