Socialized policy administration

Zeqing Guo a, Weili Han a,*, Liangxing Liu a, Wenyuan Xu b, Minyue Ni a, Yunlei Zhao a, Xiaoyang Sean Wang a

a Software School, and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China
b Department of Electronic Engineering, Zhejiang University, Zhejiang Sheng, China

ARTICLE INFO

Article history:
Received 4 August 2016
Received in revised form 26 January 2017
Accepted 6 March 2017
Available online 14 March 2017

Keywords:
Policy based management
Socialized policy administration
Mobile applications
Social computing
Android

ABSTRACT

With the rapid development of mobile applications and online social networks, users often encounter a frustrating challenge to set privacy and security policies (i.e., permission requests) of various applications correctly. For instance, in an Android system, it is hard for users, even programmers, to identify malicious permission requests (policies) when they install a third-party application. To simplify the task of policy management, in this paper, we propose a novel policy administration method where the policy settings from users’ friends will be used as a key recommendation to guide policy administration, and the security of friends’ privacy will be protected. We propose to let a user invite his or her friends to help with policy setting in applications, and we call such a method socialized policy administration (SPA for short). We designed two types of SPA: basic SPA and composite SPA. Both types of SPA are equipped with a privacy preserving mechanism that enables users’ friends to help users without leaking the friends’ preferences. In our prototype based on Telegram, i.e., one of the most popular instant messaging applications, we utilize partially homomorphic encryption cryptosystems to implement our framework. Based on the performance evaluation, SPA is able to configure almost all types of policies of current popular Android applications with a modest performance overhead.

© 2017 Elsevier Ltd. All rights reserved.

Keywords:
Policy based management
Socialized policy administration
Mobile applications
Social computing
Android

1. Introduction

The rapid development of mobile applications and social network services raises the demand of user-friendly methods for policy administration, because these applications and services require their users to set various confusing yet obscured policies (i.e., approve or reject permission requests of applications). Unfortunately, users and even applications’ developers (Fang et al., 2016) are usually unskilled at managing the applications (Enck et al., 2008; Barrera et al., 2010; Felt et al., 2011; Zhang et al., 2014; Fang et al., 2014). The common practice is that the people who are not good at security management might invite their friends and family members who are professional to help to set their applications. For example, one could invite his friends who major in Computer Science to install applications; an elder could ask his or her grandson to configure a smart phone. During these help session, it is common for the friends to show their configurations. Given that Social Network Services (SNSs for short) mimic relationships between...
humans in reality, we extend this common practice to the cyberspace. In particular, a user may ask his or her friends to help with setting his applications via Social Network Services. This extension may appear to be simple and intuitive. However, such a method would not be acceptable if the privacy of a user’s friends cannot be preserved, especially when a user consults them about sensitive settings, because if such a method cannot prevent personal information leakage, it can be used to gather friends’ private information.

Existing methods of collaborative policy authoring (Wishart et al., 2010) or administration mechanisms (Han et al., 2014) do not protect friends’ privacy. The prior work (Wishart et al., 2010) did aim to protect content privacy, e.g., how to protect the content of a user’s posts while sharing among friends on an SNS, such as Facebook. However, prior work did not consider the contextual privacy introduced as friends interact with each other. In addition, CPA (Collaborative Policy Administration) (Han et al., 2014) utilizes settings of similar applications from friends to set a user’s application without considering the privacy protection. To fill in the gap, we aim at designing a user-friendly policy administration method that does not leak user privacy.

This paper, therefore, presents socialized policy administration, whereby users ask their friends to help with privacy settings via SNSs without breaching friends’ privacy. Our approach allows users with little knowledge of policy administration to set their privacy settings automatically. The main contributions of this paper are as follows:

- We design socialized policy administration (SPA for short) where a user can request his or her friends to help with setting up sensitive policies. We design two types of SPA: Basic SPA that treats each friend equally and Composite SPA that allows users to add weights to friends.
- We propose a privacy preserving method leveraging partially homomorphic encryption algorithms, which enable order comparison between two ciphertexts without decryption. In particular, comparison between ciphertexts supports majority/minority high level policies, and achieves a better performance of the merging algorithm for setting types (e.g., Switch, Single Select, Multiple Select) than the one using prior algorithms (Guo et al., 2015).
- We implement a prototype of Composite SPA on an Android client of Telegram, which is a popular instant messaging (IM) app with 100 millions monthly active users in February 2016 (TechCrunch, 2016). The composite SPA prototype allows users to request friends’ settings and label a weight for each friend according to their professional knowledge about policy administration. Our evaluation of the prototype on Telegram illustrates its validity. The source code of prototype is uploaded on github.

Note that, although the policies of applications and services can be automatically set by SPA, users may view the settings as decision supports, and adjust the policies on their devices by themselves. As a result, professional users can also obtain useful references from their friends.

The rest of this paper is organized as follows: Section 2 introduces the background and describes the problem. Section 3 formally defines the SPA models. Section 4 describes the design and implementation of SPA. We then present our experimental process and evaluation results in Section 5. Next, we discuss the vulnerabilities of SPA and security of homomorphic encryption in Section 6. Section 7 introduces related work. Finally, Section 8 summarizes this paper and outlines our future work.

2. Background and motivation

2.1. Homomorphic encryption

Homomorphic encryption is a form of encryption that allows a set of computations to be carried out on ciphertext and generates an encrypted result which, when decrypted, matches the result of operations performed on the plaintext (Wikipedia, 2016). That is, A may encrypt a message m and send the ciphertext $E(m)$ to B. B then take the ciphertext $E(m)$ and evaluate a function $F$ on $E(m)$ to obtain the encrypted result $E(F(m))$. A decrypts the result, and obtain the expected functionality on $m$. Meanwhile B learns nothing about the data $m$.

Gentry showed the first fully homomorphic encryption scheme using lattice-based cryptography in 2009 (Gentry, 2009a; 2009b). Such a scheme allows one to compute arbitrary functions over encrypted data without the decryption key, i.e., given encryptions $E(m_1), \ldots, E(m_n)$, one can compute a composite ciphertext that encrypts $F(m_1, \ldots, m_n)$ for any computable function $F$. Although the fully homomorphic encryption (FHE) which supports an arbitrary function $F$ on ciphertexts was proposed several years ago (Wang et al., 2015; Brakerski and Vaikuntanathan, 2011; Stehlé and Steinfeld, 2010; Van Dijk et al., 2010), its performance is hard to meet the requirements for a practical business service.

As a result, partially homomorphic cryptosystems are used in practice because they are faster yet provide partial homomorphic properties. These popular partially homomorphic cryptosystems include the following.

- Paillier (Additive): The Paillier cryptosystem, invented by Pascal Paillier in 1999, is a probabilistic asymmetric algorithm for public key cryptography (Paillier, 1999). The cryptographic algorithm generates a key pair, consisting of a public key and a private key. The public key is used to encrypt plaintext; whereas the private key is used to decrypt ciphertext.

The scheme is an additive homomorphic cryptosystem (Damgård and Jurik, 2001), which has the following property.

$$F(E(m_1), E(m_2)) = E(m_1 + m_2)$$

Here, $E$ refers to encryption function, and $F$ is a function defined by the partially homomorphic cryptosystem that cal-

---

1 SPA prototype is accessible on https://github.com/SocializedPolicyAdministration.
culates the encryption of the sum of $m_1$ and $m_2$. Here, $m_1$, $m_2$ are two plaintexts.

- **RSA (Multiplicative):** RSA is one of the first practicable public-key cryptosystems and is widely used for secure data transmission. RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described the algorithm in 1977 (Rivest et al., 1978). In such a cryptosystem, the encryption key can be public and differs from the decryption key, which is kept secret.

The scheme is a multiplicative homomorphic cryptosystem (Fontaine and Galand, 2007), where the multiplicative homomorphic property can be represented as follows:

$$F(E(m_1), E(m_2)) = E(m_1 \times m_2)$$

Here, the definitions of $E$, $F$, $m_1$, $m_2$ are the same as those in Paillier.

- **ElGamal (Multiplicative):** The ElGamal encryption system is an asymmetric key encryption algorithm for public-key cryptography. It was described by ElGamal (1985). The scheme is also a multiplicative homomorphic cryptosystem, and its multiplication homomorphic property is the same with RSA’s.

### 2.2. Motivating scenario

A user, e.g., Alice, may set her applications with little professional knowledge, assisted by her friends whose settings can remain as a secret. Below is a typical scenario:

Alice, Bob, Cindy, Dale, Eric are in a group named “Classmates” in a SNS. Alice downloaded an application, which was recommended by her classmates. After she installed the application, she was confused about how to configure security settings, because she has never used this application. To set these policies properly, she has to spend time to understand all guides. To make her life easier, it will be promising if Alice can finish the setup automatically, as below.

Alice sends a request to each of her classmates in the SNS group, respectively. Upon receiving the requests, her classmates inform her their configurations CONFIDENTIALLY. Then Alice can set the application according to the statistics of her classmates’ configurations, e.g., if the majority of her classmates set “Last Seen” as “My Contacts”, then she sets it “My Contacts”. In addition, if Alice applies a high-level policy, e.g., follow the major settings in “Classmates”, the settings in the application can be configured automatically without explicit clicks from Alice.

In addition, Alice can modify settings by herself if she finds that her preferences are different from what has been set automatically by SPA. By using such a system, she can instantaneously configure the application without professional knowledge of application settings.

We define this process as socialized policy administration (SPA for short) where social relations are used to help users set sensitive policies. We envision that this method would become valuable to those who lack security awareness and experiences of policy administration on mobile devices, e.g., smartphones, wearable smart devices.

![Composite SPA](image1)

### 3. Definitions of SPA

As shown in Fig. 1, we define two types of SPA models: (1) Basic SPA that treats each friend equally; (2) Composite SPA supports multiple friend groups whereby users divide his or her friends into multiple groups (similar to Facebook) and send SPA requests to multiple groups. It allows the users to set the weights to each group according to each friend’s ability of managing policies.

#### 3.1. Basic SPA model

As shown in Fig. 2, the formal definition of Basic SPA is as follows.

**Definition 1. Basic SPA Model:**

Basic SPA := (E, g, R, P, SPAPolicy)

Here, E, g, R, P, SPAPolicy refer to a set of entities, a group, a set of roles, a set of processes, a high level policy, respectively.

- **Entity (E)** refers to the set of users who have their customized setting for applications (e.g., Android apps in mobile devices). Let $e = (id, policies)$ denote an entity, where id is the unique identification of a user $e$, policies consist of every setting policy of an application, each of which includes attributes as well as values, e.g., $e_{id}\{ (id, policies) \}$ represents that $e_{id}$ is an entity assigned to Alice with identification $id_{Alice}$, and her setting policies are represented by $policies$.

![Basic SPA Model](image2)
the motivating scenario, there are five entities: Alice, Bob, Cindy, Dale, Eric.

- **Group** ($g$) is created by a user (e.g., Alice in the motivating scenario) to carry out the socialized policy administration. Let $g = \{e_1, e_2, ..., e_n\}$ denote a group, where each element $e_i$ is an entity. In Basic SPA model, a user, e.g., Alice, can send requests to an entire group $g \subseteq \{g|e_{Alice} \in g\}$. In the motivating scenario, $g = \{Alice, Bob, Cindy, Dale, Eric\}$. Note that $|g| \geq 3$.

- **Role (R)** refers to the entities’ roles in Basic SPA Model. There are two types of roles: requester and respondent. The requester refers to the one who sends an SPA request and asks his or her friends to help with policy settings, while the respondent refers to the one who receives an SPA request. For instance, if Alice asks her friends for help, Alice is a requester and her friends (Bob, Cindy, Dale, Eric) could be respondents.

- **Process (P)** denotes either an SPA request or an SPA response in Basic SPA model. There are two types of processes: request and response. Both request and response can be described as a tuple $(e, r, p)$, where $e$ refers to an entity, $r$ refers to a role, and $p$ is a policy, which is the setting policy that the user asks for, is a part of $e$.policies. For instance, in the motivating scenario, Alice is confused about how to configure the settings on Instagram. Thus, she sends a request to her classmate group, including Bob, Cindy, Dale, Eric, for help. This request can be represented by $(Alice, requester, Instagram)$, and these responses replied by her friends in the group can be represented by $(Bob, respondent, Instagram)$, $(Cindy, respondent, Instagram)$, etc.

- **SPAPolicy** refers to a high level policy supported by the Basic SPA model. For instance, Basic SPA supports average value policy where the requester can obtain the average number based on all respondents’ settings. For a sound volume setting, each respondent can report his or her value with confidentiality to a requester, then the requester can utilize average value to set his or her sound volume.

There exists a relation between the entity and the role. Let

$$\text{role} : \text{Entity} \times \text{P} \rightarrow \text{R}$$

denote a function mapping each entity to a role, where $\text{role}(e, p) = r \land p = (e, r, p) \in \text{P}$. For example, Alice is an entity and in a process. We can use the function $\text{role}(e_{Alice}, p)$ to find the role of $e_{Alice}$ in a process $p$.

Note that, in order to ensure the confidentiality of respondents’ privacy, more than one response is required before the result is sent back to the requester to prevent him or her from identifying the settings of any respondent. In addition, the requester knows only “a little”, “some”, “most” of the respondents instead of the exact number of respondents. As such, we prevent the requester from knowing the choices of all respondents when all respondents happen to reply the same responses to a request.

### 3.2 Composite SPA model

As is shown in Fig. 3, Composite SPA is extended from Basic SPA. We change group to multi-group to allow the requester to send requests to multiple groups. We utilize the following intuition:

- How much expertise on managing privacy respondents have determines how reliable their settings are. Given that a requester is likely to be familiar with the respondents, we can enhance the quality of the aggregated setting if the requester assigns each respondent a weight according to each friend’s ability. The formal definition of Composite SPA is as follows.

**Definition 2.** Composite SPA Model:

$$\text{Composite SPA} := (E_n, MG, R, P, \text{SPAPolicy})$$

Here, $E_n$, MG, R, P, SPAPolicy refer to a set of weighted entities, a multi-group, a set of roles, and a high-level policy respectively.

- **Multi-group (MG)** is a set of several groups of friends. Let $MG = \{g_1, ..., g_n\}$, where $n \geq 1$, denote a multi-group. If and only if an entity $e \in g_1 \cap \ldots \cap g_n$, where $g_1, ..., g_n \in MG$, role($e$, $p$) can be a respondent of MG in a process $p$. If and only if an entity $e \in g_1 \cap \ldots \cap g_n$, where $g_1, ..., g_n \in MG$, role($e$, $p$) is a requester of MG in a process $p$. For instance, Alice may create two groups named highschool and university in the Composite SPA Model to organize her friend list. The set \{highschool, university\} is a multi-group. She is allowed to be a requester in this multi-group and other entities in this multi-group are respondents.

- **Weighted Entity ($E_n$)** defines an entity, which includes an additional attribute weight (the value of weight is 100 by default in our experiments), and a requester can change it before sending requests. Let $e_{Alice} = (id, policies, weight)$ denote an entity in Composite SPA, where id and policies are the same as those in Basic SPA, and weight is a number determined by a requester.

### 4. SPA: enforcement framework

#### 4.1 Composite SPA framework

The key flow of the proposed framework for SPA models is illustrated in Fig. 4. Before the processes, a key management server...
will disseminate the public and private keys of homomorphic encryption algorithms, such as Paillier, to the entities in the group. We propose the framework to support Composite SPA. We analyze SPA models’ security in Section 6.1.

A socialized policy administration process includes the following steps:

1. A requester sends a request to the friends in a group, and the request contains the weights determined by the requester. In order to ensure the confidentiality of respondents, the number of friends requested by the requester should be larger than a threshold, e.g., three. Note that, to prevent the respondent from inferring the requester’s personal judgments about himself, the weight values will be encrypted before being sent to the respondents.

2. A respondent receives the request, and he or she may respond to the request with a response encrypted by public key of homomorphic encryption.

3. A semi-trusted cloud service merges corresponding responses. Once the cloud service receives all respondents’ responses or timeout occurs and the number of responses is more than the preset threshold, e.g., three, the cloud service will send the merged result to the requester.

4. The requester receives the merged result from the cloud and decrypts it with private key of homomorphic encryption.

In our SPA framework, we support four of popular policies as follows.

- **Switch**. The status of a setting consists of “on” and “off”, e.g., “find me through email address”.

- **Single Select**. There are multiple choices for a setting, but only one choice can be selected. For instance, there are 3 choices (“Off”, “From People I Follow”, “From Everyone”) for the setting “Comment Notifications” on Instagram.

- **Multiple Selects**. There are multiple choices for a setting, and more than one choice can be selected simultaneously.

- **Continuous**. The numeric value of a setting is continuous. We define two types of high-level policies to merge responses below.

- **Majority/Minority preferred**. These two types of merging policies are applied to the policies with Switch, Single Select and Multiple Select types of policies. When a majority preferred merging policy is set, the final merging result is the majority of choices made by friends. Furthermore, when a minority preferred merging policy is set, the final merging is the minority one.

- **Average value**. This type of merging policy is applied to the policy of the continuous type. The final merging result is the average value of choices made by friends.

Table 1 shows the mappings from a combination of policies and merging policies to required homomorphic properties and recommended algorithms. Majority/Minority is used for Switch, Single Select, Multiple Select, while Average policy is suitable for Continuous. All these merging policies are supported by additive homomorphic encryption algorithm Paillier.

### Table 1 - Mappings from policies and merging policies to encryption properties.

<table>
<thead>
<tr>
<th>Privacy policies</th>
<th>Merging policies</th>
<th>Encryption properties</th>
<th>Recommended algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch</td>
<td>Majority/Minority preferred</td>
<td>Additive</td>
<td>Paillier</td>
</tr>
<tr>
<td>Single select</td>
<td>Majority/Minority preferred</td>
<td>Additive</td>
<td>Paillier</td>
</tr>
<tr>
<td>Multiple select</td>
<td>Majority/Minority preferred</td>
<td>Additive</td>
<td>Paillier</td>
</tr>
<tr>
<td>Continuous</td>
<td>Average value</td>
<td>Additive</td>
<td>Paillier</td>
</tr>
</tbody>
</table>
values with additive or multiplicative partially homomorphic encryption. It is easy to achieve comparison using fully homomorphic encryption, but the overhead of fully homomorphic encryption is prohibitive. Thus, we propose a method to compute the largest and the smallest values with privacy preserving using partially homomorphic encryption.

We address all the technical challenges with data schemes and novel algorithms as follows.

4.3. **Key data schemes in SPA**

Our SPA framework includes three key schemes: request, response, result.

4.3.1. Request

Requests are sent by a requester to respondents. Each request contains five data fields:

- **Requester.** The requester field denotes a unique id of the requester and is used for recording the requester who sends this request.
- **Weight.** The weight field contains the encrypted weight which is the requester’s personal assignment about the professional level of the targeted respondent.
- **Settings.** The settings field refers to all settings that a requester asks respondents to respond.
- **SPAPolicies.** The SPAPolicies field is a set of SPAPolicies corresponding to the settings. The cloud service will merge responses according to these policies, e.g., majority, minority, or average.

4.3.2. Response

Responses are sent from respondents to a cloud service where responses will be merged later.

- **Requester.** The requester field presents the unique id of a requester in this SPA transaction.
- **Weight.** The weight field denotes the respondent’s encrypted weight assigned by a requester.
- **Settings.** The settings field contains settings determined by requester.
- **Values.** The values field contains a set of the encrypted settings’ values that a respondent sets in the applications.
- **SPAPolicies.** The SPAPolicies field is a set of SPAPolicies to guide the cloud service on how to merge settings.

4.3.3. Result

Result is sent from the cloud service to the requester.

- **Values.** The values field is a set of merged setting values to help the requester set the application.
- **Weights.** The weights field contains the sum of weights of respondents who respond to requests.
- **Settings.** The settings field is the same as settings field in Sections 4.3.1 and 4.3.2, which enables requester know the relationship between settings and merged values.

- **SPAPolicies.** The SPAPolicies field is also the same as the one defined in Sections 4.3.1 and 4.3.2.

4.4. **Key algorithms**

In this section, we introduce how we choose an appropriate partially homomorphic encryption for our algorithms and describe three key algorithms in the SPA framework: the response encryption algorithm, the response merging algorithm, and the majority/minority computing algorithm.

4.4.1. Comparison of three encryption schemes

It may appear that three encryption schemes: order-preserving encryption (OPE for short), fully homomorphic encryption, and partially homomorphic encryption are applicable to solve our problem. However, we show below why only partially homomorphic encryption suits.

OPE is a deterministic encryption scheme whose encryption function preserves numerical ordering of the plaintexts (Boldyreva et al., 2009). Because the order between the plaintexts is preserved in OPE, the encrypted settings of each respondents can be compared by cloud service without being decrypted, which makes majority/minority policies possible. However, because OPE is deterministic, if a malicious respondent colludes with the cloud service, they can obtain all settings of the respondents whose encrypted setting values are the same. What is worse is that if the malicious respondent tries each possible combinations of setting options and compares the generated encrypted setting values with the encrypted setting values from other respondents, he can obtain the settings of all respondents. Thus, OPE is not suitable for SPA framework.

Fully homomorphic encryption is the most suitable encryption scheme for our framework in theory. However, it is too time-consuming to satisfy the requirements of practice. Therefore, fully homomorphic encryption is not the practicable solution for SPA framework.

Section 2 introduces several partially homomorphic encryption schemes. However, not all of them satisfy our requirements. To support encrypted weights, it is important to allow respondents to multiply their plaintext settings by their encrypted weight values labeled by requester. Multiplicative homomorphic encryption schemes like RSA and ElGamal seem to be suitable for this requirement. Nevertheless, cloud service needs to merge these encrypted results by adding them up for the average policy. Given these requirements, the additive homomorphic encryption Paillier is appropriate for our framework, because Paillier not only is an additive homomorphic encryption, but also allows a plaintext value to be multiplied by an encrypted value (details are shown in Section 2). Additionally, we find multiplying property is useful to implement the comparison operation for majority/minority policies as described by Algorithm 3 in Section 4.4.4.

4.4.2. Response encryption algorithm

As shown in Algorithm 1, the response encryption algorithm is designed for respondents to create an encrypted response.
The method \texttt{getSettingSize()} returns the numbers of setting elements in a response. The method \texttt{mulHomomorphic()} operates a homomorphic multiplication on the value of a plaintext setting and an encrypted weight. Thus respondents cannot read or change weight value. The value of a setting is decided in the following scheme:

- For the \texttt{Switch} types of a setting, the value is an ordered array, where \([0, 1]\) stands for “off” and \([1,0]\) for stands “on”.
- For the \texttt{Single Select}, \texttt{Multiple Select} types of setting, the setting value is an ordered array corresponding to each options. When an option is selected, the value of this option in array is 1, otherwise 0.
- For the \texttt{Continuous} type of a setting, value is the corresponding integer of the setting.

The time complexity of Algorithm 1 is \(O(mn)\), where \(m\) is the number of setting elements in a response, and \(n\) is the average number of options in each setting (we assume the number of options in settings with average policy is 1).

4.4.3. Response merging algorithm

As is shown in Algorithm 2, the response merging algorithm is designed for the cloud service to merge the responses.

When a response returns, the \textit{cloud service} processes the response as follows.

- When the \textit{cloud service} receives a response corresponding to a new request, the \textit{cloud service} generates a new record in the database, storing the response related to the specific request.
- When the \textit{cloud service} receives a response corresponding to an existing request, the \textit{cloud service} starts a merging process, which merges the newly-arrived response to an existing response, and updates the record in the database.

Algorithm 2 shows how to merge two responses. If the number of responses from friends is over two, algorithm 2 can process them iteratively. The method \texttt{getSettingSize()} returns the number of setting elements in a response. \texttt{getOptionSize()} obtains the number of option elements in a setting. The method...
addHomomorphic() operates a homomorphic addition on two encrypted values.

The time complexity of Algorithm 2 is $O(mn)$, where $m$ is the number of setting elements in a response, and $n$ is the average number of options in each setting (suppose that the number of options with average policy is 1).

4.4.4. Majority/minority comparison algorithm

Algorithm 3 shows how to get the setting option that is selected by the majority or minority of respondents.

The method getSettingSize() returns the number of settings in a result. The method getOptionSize() obtains the numbers of options of a setting. The method mulHomomorphic() operates a homomorphic multiplication on a plaintext number and an encrypted number. The method addHomomorphic() operates a homomorphic addition on two encrypted numbers. The method random() generates a positive number in the set $\{0, 1\}$.

Differences of any two options are computed from lines 12 to 15. For instance, there are three options in a setting and the encrypted sum computed in Algorithm 2 for each option are $\text{sum}_1, \text{sum}_2, \text{sum}_3$, and the differences of any two options are $\text{sum}_1 - \text{sum}_2, \text{sum}_2 - \text{sum}_3, \text{sum}_3 - \text{sum}_1$, which record the order relations of these three values. Then in lines 16 to 17, we remove all information of difference except the size relationship between these two numbers by multiplying each difference by a positive random number in $(0, 1)$. Thus the sign of the result is the same as the differences and this method prevents the requester from inferring any privacy information of the respondents from these differences.

The time complexity of Algorithm 3 is $O(mn^2)$, where $m$ is the average number of the settings in the result, and $n$ is the average number of options in each setting.

After receiving the result, the requester can decrypt the elements in the array and get the size relationship of the value of options if this setting type is a Switch, Single Select, Multiple Select. SPA will apply the setting result according to Majority or Minority policy automatically.

5. SPA: evaluation

5.1. Prototype implementation

The prototype of SPA builds on a partially homomorphic encryption algorithm, Paillier, which can support partially homomorphic multiplication and homomorphic addition, and run on an open source IM application, named Telegram. The integration of SPA into Telegram shows SPA is feasible and any Android users can download Telegram with SPA from Github and use it. The SPA prototype consists of a client application and a server service.

- The client application is released as an .apk file. After installing the application on a smartphone, a user can log in using a Telegram account, and perform the SPA functionality when he or she wants to set privacy-aware policies.
- The server service forwards requests and responses between entities. It serves as a semi-trusted cloud service to merge responses corresponding to a request.

Note that, our prototype supports all types of policy settings of Telegram except the black list setting.

5.2. Performance evaluation

In order to provide strong security strength of the SPA framework, both the client application and server service have to perform computational tasks. For the client application, it needs to encrypt a response before sending it to the server; for the server service, it takes time and space to perform a homomorphic merging operation.

We conducted the first experiment to evaluate the performance of the server service. The server service ran on a PC running Ubuntu 15.10 with a 11.7 GB memory and an Intel®
Core i7-4790 CPU at 3.60GHz processor. We varied the number of responses that the server service received, and recorded the time that the server service processed these responses. The experimental result is shown in Fig. 5.

In Fig. 5, the horizontal axis represents the total number of responses that the server received, and the vertical axis represents the time that the server dealt with these responses. We compared our Composite SPA prototype with the Basic SPA one (Guo et al., 2015) which only supports average policy. Fig. 5 shows when the server service merges responses using the average policy (both Basic SPA and Composite SPA prototypes support) or the majority policy (only Composite SPA supports), and we observed that the number of responses and the time server consumes are in positive linear relationship. The majority/minority policies (Composite SPA) take a longer duration than the average policy (Composite SPA) because according to Algorithm 2, more than one option values (in the experiment there are three option values) need to be merged when using the majority/minority policies while only one value and one weight are required to be merged when using average policy.

We conducted the second experiment to evaluate the performance of the client application. The two metrics of performance are: the time to process requests and CPU occupancy rate on Android smartphones. We conducted the experiment on Android 5.1.1, Nexus 4 with a 2.00 GB storage. The majority/minority policies and the average policy are evaluated, and we compared the result with Basic SPA prototype (Guo et al., 2015). The experimental result is shown in Fig. 6.

In Fig. 6, the horizontal axis represents the total number of requests that a client application received, and the vertical axis represents the time that the client application spent on processing the requests. We compare the average policy and the majority/minority policy supporting by Composite SPA with the average policy supporting by Basic SPA (Guo et al., 2015). Fig. 6 shows that the number of responses and the time client consumes exhibit a linear relationship. The majority/minority policy consumes the longest time among all three policies, because it involved more than one option values and requires a longer computational time to be merged than those of Basic SPA. Though policies of Composite SPA consume more time than those of Basic SPA, respondents will not notice the extra time because it is asynchronous.

We further measured the performance of the client by connecting the smartphone device to a computer, and executing “adb shell top” command to view ongoing tasks. We monitored the status several times on a Google Nexus 4 running Android 5.1.1. We observed that the maximum of CPU usage of the client application did not exceed 11%, including encrypting a response before sending it out.

In summary, we can see that the implemented SPA supports automated policy setting of the current popular applications with modest performance overhead.

6. Discussion

6.1. Privacy of respondents

One vulnerability will exist when the requester acts as an attacker to gather the privacy of his or her friend. The requester sends a request to only one friend and other accounts which are also managed by the requester himself. When the only friend creates a response, it is possible for the requester to infer the settings of the friend based on the final merging result and his or her own settings. The vulnerability also exists when a requester colludes with other friends to crack into the settings of a targeted friend.

Respondents may breach his or her privacy if we do not provide protection in our framework. When requesters use the average policy in Composite SPA, it is possible for a requester to obtain all settings from respondents who send responses separately. The requester can use carefully designed weights to store all responded respondents’ setting values in different digits of weighted sum. For example, Alice wants to get each friend’s setting value, which ranges 0 to 100, from Bob, Cindy, Dale, who are her friends. She can set Bob’s weight to 1, Cindy’s weight to 1000, and Dale’s weight to 1,000,000, and send requests to them. These three weight values ensure Alice’s friends’ setting values store in weighted sum separately. (If there are some
friends who do not respond to Alice’s request, the value will be 0, and it will not affect others setting values) The cloud service will send the encrypted average value of the weighted sum to Alice. Then Alice can obtain the setting values from the respondents who responded to her request. As a result, the mechanism to support the Composite SPA requires extra concerns to protect friends’ privacy. To guarantee the privacy of respondents, our SPA framework provides a restriction of the Basic SPA when requesters use the average policy. That is, respondents can choose only to respond the average policy when the SPA framework is the Basic SPA. When respondents receive a request that includes an encrypted weight value and an average policy, they will be warned about the privacy risk of SPA.

Sometimes respondents are happy to share their settings with the requester, while they do not want their settings to be publicly available. Our Composite SPA supports the requester to send requests to ask respondents to share their unencrypted setting values.

6.2. Errors propagation

It is possible that some unprofessional or malicious settings propagate in our SPA framework. To reduce the errors propagation, the SPA framework enables a requester to send the decrypted SPA result to respondents. Respondents may respond whether the result is reasonable. According to these assessments, the requester decides whether to use the SPA result or not. If the result is assessed as malicious, SPA allows the requester to ask the respondents to disclosure of their settings. This helps requesters to determine whether the result is trusted and prevents the errors from propagating on the social network.

As unprofessional or malicious SPA results propagates in the social network, the result will be noticed by increasing number of respondents and the possibility of unprofessional requesters being warned increases. Therefore, in our SPA framework, errors might propagate, but the impact on the social networks is limited considering that majority of users on social network is unmalicious.

6.3. Security of homomorphic encryption

The security of homomorphic encryption affects the security strength of the SPA framework. Shannon formalized the security of encryption schemes for the first time in the literature (Shannon, 1949). Shannon introduced the notion of perfect secrecy/uncidental security, which characterized encryption schemes in which the knowledge of a ciphertext does not give any information about either the corresponding plaintext or the key (Fontaine and Galand, 2007). The highest security level a homomorphic encryption can reach is IND-CPA (Fontaine and Galand, 2007). IND stands for indistinguishability whereas CPA are acronyms for chosen plaintext attack. A chosen plaintext attack (CPA) in cryptanalysis presumes that the attacker has the capability to choose arbitrary plaintexts to be encrypted and obtain the corresponding ciphertexts (Anderson, 2001).

Paillier and ElGamal achieve the highest security level for homomorphic encryption schemes. RSA cannot achieve a security level of IND-CPA. Nevertheless, RSA is still considered strong enough.

7. Related work

This paper proposes a novel policy administration methodology, i.e. socialized policy administration (SPA), to manage personal policies, where users can invite their friends to help with setting sensitive policies without breaching friends’ privacy.

Policy administration is an effective approach to protect and operate information systems (Sloman, 1994; Lymberopoulos et al., 2003). The literature (Moore et al., 2001) specifies four core components in a traditional framework of policy-based management: Policy Decision Point (PDP), Policy Enforcement Point (PEP), Policy Administration Point (PAP), and Policy Repository (PR). In the traditional administration model, a professional expert or group will take responsibility of the policy administration, whose functions include policy design, policy verification, and policy deployment (Han et al., 2014). Many researchers proposed their policy administration methods (Sandhu and Munawer, 1999; Li and Mao, 2007). However, smartphones and mobile applications challenge the existing trust model in the policy administration, where common users do not possess professional knowledge of policy-based management.

Therefore, much work has been done to incorporate new requirements in policy administration. In the literature, Squicciarini et al. (2009) pointed out, in spite of the fact that content sharing represents one of the prominent features of existing Social Network sites, Social Networks are yet to support any mechanism for collaborative management of settings for shared content. Squicciarini et al. modeled the problem of collaborative enforcement of privacy policies on shared data by using game theory. In particular, they proposed a solution that offers automatic ways to share images based on an extended notion of content ownership. The approach utilizes the concept of shared ownership of data. This is achieved by having the originator of the data, i.e. the user responsible for uploading the data, to specify other potential owners of that data. The system then holds an auction on the possible privacy policy to apply to the data in which all the owners submit a vote for their desired policy. The literature claims to be the first research to discuss a novel model for privacy management across social networks, where data may belong to many users.

In the literature, Wishart et al. (2010) pointed out that content sharing on social network services may lead to privacy issues. The literature proposes a privacy-aware social networking service and then introduced a collaborative approach to authoring privacy policies for the service. The approach permits the originators of content on the social network to specify policies for the content they upload. The conditions under which the policy is applied can then be edited by nominated users of the social networking service.

Shehab and Marouf (2012) proposed policy recommendation. They proposed a multicriteria recommendation model that utilized application-based, user-based, and category-based collaborative filtering mechanisms. Collaborative filtering mechanisms are based on previous user decisions, and application permission requests to enhance the privacy of the overall site’s user population.

Han et al. (2014) proposed a policy administration mechanism, referred to as collaborative policy administration (CPA
for short), to simplify the policy administration. In CPA, a policy administrator can refer to other similar policies to set up their own policies to protect privacy and other sensitive information. To obtain similar policies effectively, a text mining-based similarity measure method is presented.

Existing approaches of collaborative policy authoring or administration involve cooperators or friends. However, these methods rarely focus on protecting the privacy of those who participate in the collaborative policy administration process. Instead, this paper pays attention to the privacy of friends who help to set users’ applications. We implemented an enforcement SPA framework by using homomorphic encryption, and experimental results show that the proposed mechanism can support all high level policies we propose with modest performance overhead.

8. Conclusion and future work

To the best of our knowledge, this paper is the first work to present the method of socialized policy administration (SPA for short), where a user can request his or her friends to help with sensitive policies setting without breaching friends’ privacy. We formally define two types of SPA models; then propose an SPA framework that supports Composite SPA for mobile applications, and implement the SPA framework using a partially homomorphic encryption, Paillier. We integrated with a popular IM app, Telegram. Our evaluating results show that the SPA framework supports all high level policies that we propose, and the performance can support these policies in practice.

As a direction of future work, we will improve the efficiency of our framework. In addition, we will explore methods for a user to find other users who may have similar usage patterns to customize the result for the requester. And we will systematically analyze the infrastructural requirements to employ the SPA framework in practice. Last but not least, we will empirically study the users’ behaviors when users operate their policy settings and how errors propagate through the overlay social network, because users’ behavior model and the propagation of errors can help us to understand the robustness of SPA.

REFERENCES


TechCrunch, Encrypted messaging app telegram hits 100m monthly active users, 350k new users each day; 2016. Available from: https://techcrunch.com/2016/02/23/encrypted-messaging-app-telegram-hits-100m-monthly-active-users-350k-new-users-each-day/.


Zeqing Guo is a graduate student at Software School, Fudan University. His research interests focus on policy-based management and system security.

Weili Han is an associate professor at Fudan University. His research interests are mainly in the fields of Access Control, Digital Identity, IoT security. He is now a member of the ACM, SIGSAC, IEEE, and CCF. He received his Ph.D. at Zhejiang University in 2003. Then, he joined the faculty of Software School at Fudan University. From 2008 to 2009, he visited Purdue University as a visiting professor funded by China Scholarship Council and Purdue University. He serves in several leading conferences and journals as PC members, reviewers, and an associate editor.

Liangxing Liu received the Bachelor’s degree from East China Normal University, Shanghai, China, in 2012. He is currently working toward the Master’s degree in the Software School, Fudan University, Shanghai.

Wenyuan Xu received her Ph.D. degree in electrical and computer engineering from Rutgers University in 2007. She is a professor at Zhejiang University. Her research interests include wireless networking, network security and privacy. Dr. Xu received the NSF Career Award in 2009. She is a member of the ACM and IEEE, and has served on the technical program committees for several IEEE/ACM conferences on wireless networking and security.

Minyue Ni is a graduate student at Software School, Fudan University. Her research interests focus on policy-based management and information security.

Xiaoyang Sean Wang is the professor at Fudan University. He obtained his Ph.D. degree in 1992 in Computer Science from the University of Southern California, Los Angeles, California, and earned his MS and BS degrees in Computer Science earlier from Fudan University, Shanghai, China. In 1992, he joined the faculty of the Information and Software Engineering Department at George Mason University, Fairfax, Virginia, and in 2003, he moved to UVM. His research areas include database systems, system support for temporal data and time series data, data mining, temporal reasoning, and information security. He is the Principal Investigator or Co-Principal Investigator of a number of federally sponsored research projects, and was a recipient of both the National Science Foundation Career and Research Initiation Awards. He has published widely in the general area of database systems in journals and at conferences such as ACM Transactions on Database Systems (TODS), IEEE Transactions on Knowledge and Data Engineering (TKDE), ACM SIGMOD International Conference on Management of Data (SIGMOD), and Very Large Data Bases Conference (VLDB). He has served on Program Committees (PC), as PC chair, or in other capacities for many conference organizations, and is on the editorial boards of a number of technical journals.