Efficient Anonymous Message Submission

Xinxin Zhao, Lingjun Li, Guoliang Xue, *Fellow, IEEE,* and Gail-Joon Ahn, *Senior Member, IEEE*

Abstract—In online surveys, many people are reluctant to provide true answers due to privacy concerns. Thus, anonymity is important for online message collection. Existing solutions let each member blindly shuffle the submitted messages by using an IND-CCA2 secure cryptosystem. In the end, the message sender's identities are protected since no one knows the message submission order. These approaches cannot efficiently handle groups of large size. In this paper, we propose an efficient anonymous message submission protocol aimed at a practical group size. Our protocol is based on a secret sharing scheme and a symmetric key cryptosystem. We propose a novel method to aggregate members' messages into a message vector such that a group member knows only his own position in the submission sequence. The protocol is accountable for capturing malicious members breaking the protocol execution. We provide a theoretical proof showing that our protocol is anonymous under malicious attacks. We also discuss our simulation results to demonstrate the efficiency of our protocol.

Index Terms—group messaging submission, secret sharing scheme, accountablility, anonymity, identity protection

1 Introduction

In the real world, anonymity has always been an important societal issue. This issue is becoming increasingly important as more and more people discover the digital world and find the need for anonymous participation, In addition, the new wave of online social networks has created an unprecedented demand for anonymity in the digital world.

Consider a health care product company that wants to do an online survey about its products via an online social network, such as Facebook. The company wants to receive accurate feedback from its users and avoid redundant responses from the same person. Hence, the company would like the participants to login to their social network accounts. This can be easily achieved by having, for example, a "Login with Facebook" button on the survey web page. In this way, the company could get the participants' demographic data and reject repeated submissions. However, the participants may not feel good about this approach, because their social identities are exposed. Particularly, people would rather not participate in the survey when the answers contain private information, such as their health conditions. Another example is that some online career social networks, such as LinkedIn, often conduct surveys like "how do you like your current company". The result of this kind of survey is very helpful to other people's career development. However, people would not like to answer or may not give their truthful thinking if they are worried that their negative answers may link to their real identities and backfire on themselves.

Anonymous message submission has attracted broad attentions [4], [10], [30]. In [30], Yang *et al.* proposed the first construction for online anonymous message submission. Bickell and Shmatikov revisited this problem and proposed a collusion resistant anonymous data collection protocol in [4]. Recently, Dissent protocol [10], [25] has been proposed which is accountable and can send variable length messages [10]. Anonymity in the above protocols is guaranteed by a "shuffle" process, which is collaboratively done by all group members. The existing solutions [4] [10] are based on two rounds of IND-CCA2 secure encryption that are done by group members one by one. The fact that the protocol's communication rounds grows linearly in the group size makes it inefficient to handle groups of scalable size.

The motivation of this paper is to construct an efficient and collusion resistant anonymous message submission protocol for groups of a practical scale (e.g., hundreds of group members). We propose a novel technique which secretly aggregates the group members' messages into a message vector with each message being a component in the vector. Each member knows his selected component index but is unaware of other members' choices. Our work aims to promote the practical application of message submission with strong anonymity guarantees, which will increase privacy protections and anonymity in the digital world.

Our main contributions are as follows:

1) We propose a novel anonymous position application technique, which enables each group member to obtain a position in the final message submission sequence in a manner that each member is oblivious to other members' positions.
2) We propose an efficient anonymous message submission protocol, AMS, for groups of practical scale. Our protocol runs in time bounded by a polynomial in the group size, and has *constant communication rounds*.
3) We prove that the AMS protocol is anonymous

• *X. Zhao, L. Li, G. Xue, and G. Ahn are all with the School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ, 85281. E-mail: {xinxin.zhao, lingjun.li, xue, ahn}@asu.edu*
This research was supported in part by ARO grant W911NF-09-1-0467, NSF grants 1421685, 1461886, and 1527268, and Global Research Laboratory Project grant from National Research Foundation (NRF-2014K1A1A2043029). The information reported here does not reflect the position or the policy of the funding agencies. Part of the results in this paper has appeared in IEEE INFOCOM2012 [32].

and collusion resistant under malicious attacks. The AMS protocol does not rely on any trusted third party. The security of our protocol is based on the security of the secret sharing scheme and the symmetric key cryptosystem.

4) We analyze the performance of our protocol from the aspects of security and efficiency. We conduct comprehensive simulations showing that our protocol is more efficient than existing ones, especially when the group size is large.

The entire system consists of two protocols: AMS and BULK [10], [25]. We use AMS to shuffle N messages of fixed length to disconnect the link between the message and its sender. The extractor is of fixed length. Using AMS to anonymously shuffle the message extractors, BULK can submit variable length messages in an anonymous way. To achieve accountability, we also adopt the approach proposed in [25]. Each group member signs and commits all messages he sent. If any misbehavior is reported, all members disclose their temporary secret in this run and an individual member can replay, verify other members' actions, and blame the misbehaving member.

We adopt the signature based audit protocol and the DC-net based message transmission protocol proposed in [10].

The novelty of our AMS protocol is to let each member efficiently and secretly choose a unique number between 1 and N. This number is then used as the slot position to anonymously submit messages. Specifically, each member has a vector. The component of his choice is set to 1 and the rest are set to 0. AMS uses Shamir's secret sharing [22] to integrate all members' vectors without leaking the content of each vector. The members know the conflicts by checking whether a component in the integrated vector is larger than 1. AMS uses a novel way to remove conflicts and achieves high success (no conflict) probability. Since Shamir's secret sharing scheme mainly uses additions, AMS is lightweight and can handle large groups.

The rest of this paper is organized as follows. We formulate the problem in Section 2. In Section 3, we present the related work. We demonstrate the protocol construction in Sections 5, 6, and 8. We analyze our protocol from the security and efficiency aspects in Section 7 and show our simulation results in Section 9.

2 Problem Formulation

In this section, we present the network model, the threat model, and the security objectives.

2.1 Network Model

Our network consists of $N+1$ parties: a set $\mathcal{M}$ consisting of N group members, and a collector C who collects all group members' messages. Each group member has a unique group ID. Without loss of generality, we assume that an ID is a number from $\{1, 2, \cdots, N\}$. The group members want to submit their messages to the collector without exposing their identities. The initiator of the protocol could be either the collector or any group member. We assume that there are at least two honest group members, since it is impossible to ensure anonymity if there is only one honest member.

We **do not** assume the existence of a trusted third party during our protocol execution. Our protocol runs in a completely distributed manner. All the communications are carried out on secure channels. We assume that each member has a public/private key pair. The secure channels can be set up by using a public key cryptosystem. We also assume that all members keep connected to the network during the protocol execution.

2.2 Threat Model

There are two types of adversaries: *semi-honest* adversaries and *malicious* adversaries [14] [15]. Semi-honest adversaries honestly follow the protocol execution but are curious about people's private information. They will do their best to collect all messages that they can obtain, analyze them, and infer private information. Malicious adversaries do not necessarily follow the protocol and may eavesdrop the communications, modify, replay, or inject messages. Adversaries could be multiple parties or a single party (e.g., the collector). If multiple parties collude, we consider they are controlled by one adversary so that we only need to consider a single adversary hereafter.

In this paper, we consider the security of our protocol in the presence of a malicious adversary. Since a party can always abort from a protocol execution, we do not claim that our protocol can successfully deliver messages under any circumstance, which is the same as in [4] and [10]. Instead, we prove that the security properties are always preserved when the protocol terminates.

2.3 Security Objectives

The security objectives of our work are stated as follows.

- **Anonymity:** If k $(k \leq N-2)$ out of the N $(N \geq 3)$ group members collude with the collector, they cannot infer the sender's identity for a given message. Note that anonymity cannot be guaranteed when there is only one honest group member. Having all $N-1$ messages and identities, the adversary can easily link the remaining message to the honest member.
- **Integrity:** When the protocol terminates, the collector should either receive the honest members' messages or be notified that the messages are modified. In the latter case, the culprit should be exposed.
- **Accountability:** At least one malicious member should finally be exposed by the group members if the protocol execution is broken.

We use an *anonymization game* [4] to formalize our notion of anonymity for the anonymous message submission protocol. The anonymization game is played

between an adversary and an oracle with a security parameter 1^λ, where λ is an integer. Suppose that k out of the N group members and the collector are dishonest. The adversary plays the roles of the collector and the k dishonest members, while the oracle plays the roles of the honest members. A protocol is said to be *anonymous* if the adversary can win the game with only a negligible probability. Prior to the game, the adversary chooses $N - k$ messages and gives them to the oracle, who then participates in the anonymous message submission protocol and submits these messages to the collector on behalf of the honest members. The adversary may repeat this process for a polynomial number of times. Formally, the anonymization game is defined as follows.

1) The adversary chooses two honest members α and β, and two messages d_0 and d_1 in plaintext. He also assigns a message d_i, in plaintext, to each remaining honest member i. The adversary gives these messages to the oracle.
2) The oracle selects a bit $b \in \{0, 1\}$ uniformly at random, and sets $d_\alpha = d_b$ and $d_\beta = d_{\bar{b}}$, where $\bar{b}$ denotes the negation of b.
3) The oracle participates in the anonymous message submission protocol. When a message is needed for an honest member i, the oracle responds with the message d_i.
4) After observing the protocol execution, the adversary outputs his guess about b.

Let $\mathcal{D}$ be a probabilistic polynomial time adversary. Then

$$Pr[\mathcal{D}(1^\lambda, \alpha, \beta, d_0, d_1, 0) = 1]$$

is the probability that $\mathcal{D}$ outputs 1 when $b = 0$, and

$$Pr[\mathcal{D}(1^\lambda, \alpha, \beta, d_0, d_1, 1) = 1]$$

is the probability that $\mathcal{D}$ outputs 1 when $b = 1$. The adversary's advantage is

$$\begin{aligned} Adv_{\mathcal{D}} = &Pr[\mathcal{D}(1^\lambda, \alpha, \beta, d_0, d_1, 1) = 1] \\ &- Pr[\mathcal{D}(1^\lambda, \alpha, \beta, d_0, d_1, 0) = 1]. \end{aligned}$$

Definition 1. *A message submission protocol is anonymous if, for any probabilistic polynomial time adversary $\mathcal{D}$, its advantage in the anonymization game is a negligible function in λ.* □

Again, we note that this definition is valid only when there are at least two honest group members, i.e. $k \leq N - 2$.

3 Related Work

In this section, we review closely related work on anonymous message submission and secure multi-party computation (SMC).

In [4], Brickell and Shmatikov proposed a collusion resistant anonymous data collection protocol. In their protocol, each member generates his primary and secondary public/private key pairs. The users first encrypt their messages using the collector's public key, then encrypt the resulting ciphertexts using each member's secondary public key, and finally encrypt the resulting ciphertexts using each member's primary public key. Next, each ciphertext is randomly shuffled by its owner who also strips off one layer of the encryption during the shuffle. Finally, the ciphertexts are sent to the collector in a random order. The collector decrypts the ciphertexts using the secondary private keys sent by the members and his own private key. In [10], Corrigan-Gibbs and Ford extended the shuffle protocol and proposed an accountable anonymous message submission protocol, called Dissent. Dissent consists of two protocols: shuffle and bulk. They still use the shuffle technique in [4] in their shuffle protocol. In addition, each member in the protocol of [10] creates a log file and updates its state during the protocol execution. If at some point the protocol terminates abnormally, all members execute the protocol again according to their log files so that they can expose the member who is responsible for the abnormality. The bulk protocol enables the members to send variable length messages. It requires $2N + 7$ communication rounds and $O(N^2)$ total computations.

Many anonymous messaging systems focused on end to end anonymous communications. HerbivoreFS [23] is a typical example, which dynamically and independently assigns users to small cliques and applies DC-net protocol in a small clique to achieve end-to-end anonymous file sharing.

Mix-networks [5] provides practical anonymous but high latency communication. Many mix-network designs are vulnerable to active disruptions [13], [17]. Cryptographically verifiable shuffle [21] may be a solution to the disruptions and pursues the similar goals of our work. However, verifiable shuffle focuses on verifying a shuffle's correctness, i.e., whether the final shuffle is a permutation of original messages, rather than its anonymity and accountability. Many current verifiable shuffle designs are not efficient in handling unbalanced message load and large group size. E-voting system [9] is another solution to the disruption. But many proposals are designed for delivery of small size messages and cannot handle unbalanced load efficiently.

Recently, Young and Yung proposed an end-to-end anonymous communication scheme [31]. Their work's concentration is different from that of AMS. The scheme in [31] focused on hiding existence of communication between sender and receiver. AMS is a group messaging system, concentrating on hiding sender's identity while maintaining message integrity and accountability.

Low-latency designs, such as onion routing [5], [26], are independent of upper layer communication applications, but they typically provide weaker anonymity. For example, the onion routing protocol is vulnerable to traffic analysis if an attacker sniffs the network traffic and monitors the streams going into and out of the network [27]. While using ring signature [7] may help to strengthen the anonymity of onion routing, the ring signature cannot protect against an attacker's Sybil attack, i.e., a malicious sender submits more than one message.

Other anonymous data submission schemes, such as Mix-networks and DC-nets [6] [16] [23] [28], also achieve strong anonymity. However, they seem to be a poor match for our scenario, since the collector may need to know who the group members are. In this case, complicated techniques are needed to enable a well-defined group to submit their messages to the collector, and one or more nodes in the network may be compromised, which breaks down anonymity. Our protocol can provide strong anonymity even when k ($k \leq N-2$) out of the N group members are compromised.

To make strong anonymous communications scalable, Corrigan-Gibbs *et al.* proposed a client/server architecture in [29] and then developed it in [11]. In their architecture, clients are divided into several groups and each group is assigned to a server, which is called the group's "upstream" server. Each client only communicates with its upstream server during the entire communication session. An upstream server collects message ciphertexts from downstream clients and communicates with other servers as its downstream clients' agent. Servers then collaboratively put messages together and hand down to their downstream clients. The system's anonymity relies on DC-nets and the assumption that at least one upstream server is trusted. Different from their approaches, the AMS protocol proposed in this work is a peer-to-peer protocol and does not assume the existence of a trusted third party.

Our previous work on anonymous group messaging system was published in [32]. Compared to the work of [32], we make several improvements in this work. First, in [32], all group members can only submit equal length of messages. In this work, we use bulk protocol [10] as a building block, which enables group members to submit variable length of messages. Second, we revised our system construction, and added signature based audit log. Group members can trace to the faulty member who does not follow the protocol execution or maliciously modify, reply, or inject messages. Third, we did extensive simulations to demonstrate the performance of bulk protocol built on C-shuffle and R-shuffle protocols [10], respectively, versus bulk protocol built on our AMS protocol. We also added the performance comparison between the padding technique and the bulk protocol, both of which can be used to transmit variable length of messages. We gave more detailed simulation setup descriptions and result analysis.

4 Secret Sharing Schemes

A (t, N)-secret sharing ((t, N)-SS) scheme is an efficient scheme that shares a secret among N parties. As stated in [22], in the scheme, a secret is divided into N shares and the i-th share is given $[s]_i^{(t,N)}$ to the i-th party. At least t parties are required to reconstruct s. The notation $[s]_i^{(t,N)}$ denotes the share specifically for the i-th party.

When $t = N$, there is a simplified (N, N)-SS scheme which can achieve linear time complexity [24]. Suppose there is a secret $s \in \mathbb{Z}_m$ that is to be shared among N parties. First, we secretly choose (independently at random) $N-1$ elements $s_1, s_2, \cdots, s_{N-1}$ from $\mathbb{Z}_m$, compute $s_N = s - s_1 - \cdots - s_{N-1} \mod m$, and give s_i to the i-th party. In order to reconstruct s, N parties expose their shares, and compute $s = s_1 + s_2 + \cdots + s_N$. In this paper, we use (N, N)-SS in our protocol construction. We use the notation $[s]_i$ to denote the share for the i-th party.

The aforementioned (N, N)-SS is additive homomorphic [3]. Particularly, given two secrets a_0 and a_1, we have $[a_0 + a_1]_i = [a_0]_i + [a_1]_i$, where $[a_0 + a_1]_i$, $[a_0]_i$, and $[a_1]_i$ are the i-th shares of $a_0 + a_1$, a_0, and a_1, respectively.

We say that an (N, N)-SS is *indistinguishable* if it is impossible to learn any information about a secret from any of its $N-1$ shares. The indistinguishability of an (N, N)-SS is defined by the following *distinguishing game*, which is played between an adversary and an oracle.

1) The adversary splits the secret using the (N, N)-SS, and performs any operation on the shares.
2) The adversary submits two distinct secrets a_0 and a_1 to the oracle.
3) The oracle selects a bit $b \in \{0, 1\}$ uniformly at random, splits $a_b, a_{\bar{b}}$ using the (N, N)-SS, and returns $N-1$ shares of a_b followed by $N-1$ shares of $a_{\bar{b}}$. The indices of returned shares are specified by the adversary. Without loss of generality, we assume that the returned shares are $[a_b]_1, \cdots, [a_b]_{N-1}$ and $[a_{\bar{b}}]_1, \cdots, [a_{\bar{b}}]_{N-1}$, respectively.
4) The adversary is free to perform any operation on the returned shares, and, finally, outputs a guess for the value of b.

Let $\mathcal{A}$ be an adversary algorithm. Then

$$Pr[\mathcal{A}(a_0, a_1, [a_b]_1, \cdots, [a_b]_{N-1}, [a_{\bar{b}}]_1, \cdots, [a_{\bar{b}}]_{N-1}, 0) = 1]$$

is the probability that $\mathcal{A}$ outputs 1 when $b = 0$, and

$$Pr[\mathcal{A}(a_0, a_1, [a_b]_1, \cdots, [a_b]_{N-1}, [a_{\bar{b}}]_1, \cdots, [a_{\bar{b}}]_{N-1}, 1) = 1]$$

is the probability that $\mathcal{A}$ outputs 1 when $b = 1$. The adversary's advantage is

$$Adv_{\mathcal{A}} =$$
$$Pr[\mathcal{A}(a_0, a_1, [a_b]_1, \cdots, [a_b]_{N-1}, [a_{\bar{b}}]_1, \cdots, [a_{\bar{b}}]_{N-1}, 1) = 1] -$$
$$Pr[\mathcal{A}(a_0, a_1, [a_b]_1, \cdots, [a_b]_{N-1}, [a_{\bar{b}}]_1, \cdots, [a_{\bar{b}}]_{N-1}, 0) = 1].$$

Definition 2. *An (N, N)-secret sharing scheme is said to be unconditionally indistinguishable if for any two secrets, a_0 and a_1, the advantage of any algorithm $\mathcal{A}$ in the distinguishing game is 0.* □

Theorem 1. *The simplified (N, N)-secret sharing scheme is unconditionally indistinguishable.* □

Proof: Suppose $\mathcal{A}$ gets $N-1$ shares $[a_0]_1, \cdots, [a_0]_{N-1}$ generated by the aforementioned simplified (N, N)-SS, where $a_0 \in \mathbb{Z}_m$. Then for any $a_0' \in \mathbb{Z}_m$, there is a unique $[a_0']_N \in \mathbb{Z}_m$ such that $[a_0]_1 + \cdots + [a_0]_{N-1} + [a_0']_N = a_0'$. Since a_0' is distributed uniformly over $\mathbb{Z}_m$, the construction of $[a_0']_N$ is equally likely. Thus, $[a_0]_1, \cdots, [a_0]_{N-1}$ could be $N-1$ shares of any number $a_0' \in \mathbb{Z}_m$ with equal probability. It is true for any $N-1$ shares of a secret in $\mathbb{Z}_m$. We conclude that the distribution of the $N-1$ shares of a_0 and a_1 is identical. Observing the

$N-1$ shares, the adversary gets no information about the secret. Thus $Adv_{\mathcal{A}} = 0$. □

5 Preludes to Protocol Construction

5.1 Protocol Overview

Our protocol consists of two sub-protocols: an anonymous message submission (AMS) protocol and a bulk protocol. The bulk protocol is used to send variable length messages and is proposed by Corrigan-Gibbs and Ford in [10]. We adopt it as a building block for our protocol. Our contribution is an efficient peer-to-peer anonymous message submission protocol for groups of scalable size. It is comparable to the anonymous shuffle protocol [4]. The purpose of AMS is to submit a sequence of messages to a designated collector without disclosing the message senders' identities.

AMS consists of six phases: application, encryption, anonymization, verification, decryption, and blame. Compared with existing shuffle protocols, AMS saves the computation time in the following aspects: 1) The core of AMS relies on two light-weight operations – a simplified (N, N)-SS and a symmetric key encryption (i.e., AES); and 2) A member's operations in each phase are independent of that of other members. However, AMS requires the messages to be of the same length, which is a common weakness of existing shuffle protocols. We briefly introduce the six phases of the AMS protocol in the following.

- **Application**: The application phase is to make each member choose a position in the final sequence but be oblivious to other members' choices. When the application phase ends, each member i obtains a unique position π_i $(1 \leq \pi_i \leq N)$. Here, $[\pi_1, \cdots, \pi_N]$ is a random permutation of $[1, \cdots, N]$.
- **Encryption**: Each member i encrypts his data d_i using his symmetric key k_i and gets the ciphertext $e_i = Enc_{k_i}(d_i)$ with equal length of r bits.
- **Anonymization**: Each member i constructs a data vector $\vec{e}$ such that the π_i-th component is e_i and the rest components are 0. All members split the data vector using the (N, N)-SS, keep their own shares confidential, and send out the remaining shares.
- **Verification**: Each member i reconstructs $\vec{e}$ and checks whether his e_i is the π_i-th component of $\vec{e}$. If any member's message has been altered, an alert message is broadcasted and all members go to the blame phase; otherwise, all members encrypt their symmetric key using a distributed ElGamal encryption [20] and anonymously broadcast the ciphertext via the same technique as in the anonymization phase. Finally, all members obtain a key ciphertext vector $\vec{k'}$ with π_i-th component being the ElGamal encryption key k'_i $(1 \leq i \leq N)$.
- **Decryption**: After checking the integrity of key ciphertexts, each member retrieves the key ciphertexts from $\vec{k'}$, decrypts them, and uses the keys to decrypt the corresponding messages.
- **Blame**: All members publishes their secret choices of their positions, the message ciphertexts, the key ciphertexts, the messages received from and sent to other members, and their message logs. All members then replay the protocol execution and each member's behaviors to expose at least one culprit.

We use the bulk protocol [10] to send variable length messages. To execute the bulk protocol, each member generates a ciphertext of his data by XORing the data pseudo random numbers generated by himself. Next, the member constructs a message extractor that helps other members to reproduce the corresponding pseudo random numbers. All members' message extractors are of equal length. Hence, we transform variable length messages into equal length ones. All members then execute the AMS protocol to anonymously broadcast the message extractors. Using the message extractors, the collector can reconstruct the messages by XORing necessary pseudo random numbers.

5.2 Anonymous Data Aggregation

A technique used in our protocol is to aggregate data held by group members into a data vector $\vec{v} = [v_{\sigma^{-1}(1)}, v_{\sigma^{-1}(2)}, \cdots, v_{\sigma^{-1}(N)}]$ in a way that a member's position is only known to himself. Here, v_i is the data held by a member i and σ is the permutation

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & N \\ \pi_1 & \pi_2 & \cdots & \pi_N \end{pmatrix}.$$

Each member i obtains the position π_i in the application phase. To achieve this, each member i constructs an individual vector $\vec{v}_i$ such that the π_i-th component is v_i and the remaining components are 0. Each member i generates N shares for $\vec{v}_i$ by splitting each component of $\vec{v}_i$ using (N, N)-SS. The t-th share of the vector is a vector of all components' t-th shares. Next, all members send their j-th share to member j $(1 \leq j \leq N)$ and keep their own share secret. Hence, each member receives $N-1$ shares from other members. Since (N, N)-SS is additive homomorphic, each member i sums up the received $N-1$ shares and his own share to form one share of vector $\vec{v}$. The group members together can reconstruct $\vec{v}_i$.

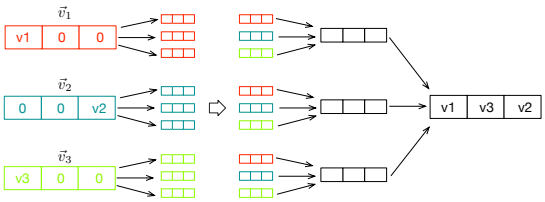


Fig. 1. Data aggregation

Fig. 1 illustrates the process of the data aggregation. There are three members 1, 2, and 3. In this example, $N = 3$. The positions they obtained are $\pi_1 = 1$, $\pi_2 = 3$, and $\pi_3 = 2$. Members 1, 2, and 3 construct individual vectors $\vec{v}_1$, $\vec{v}_2$, and $\vec{v}_3$ as shown in Fig. 1. Using a $(3, 3)$-SS, members $1, 2$ and 3 share $\vec{v}_1$, $\vec{v}_2$, and $\vec{v}_3$ with the other

two members, respectively. In this way, each member gets two shares from the other members. Due to the homomorphic property of (N, N)-SS, each member sums up the two received shares and the share kept by himself, and gets a share of $\vec{v}$. Finally, we can sum up three shares to get the vector $\vec{v}$.

6 Anonymous Message Submission Protocol

In this section, we present the details of each phase in our AMS protocol. Before the protocol execution, each member i generates a signature key pair (u_i, v_i), where u_i is the signing key and v_i is the verifying key. The signature of a message m is denoted as $Sig_{u_i}(m)$. When a signature is sent with its message, we use the notation $Sig_{u_i}\{m\}$ to denote the concatenation, i.e., $Sig_{u_i}\{m\} = m||Sig_{u_i}(m)$.

Each member generates an individual encryption key k_i $(1 \leq i \leq N)$ for a symmetric key system. In addition, the system selects a cyclic multiplicative group $\mathbb{G}$ of prime order p and one of its generators g.

For a member i, the AMS protocol can be viewed as a function $AMS(d_i, \mathbb{K}, n_R, f_i)$, where d_i is the message to be sent, $\mathbb{K}$ is the set of all members' verification keys, n_R is the nonce identifying this run, and f_i is a boolean indicating whether this is a bypass run. The output of this function is either (`success`, D'), where D' is a shuffle of all messages, or (`fail`, $\texttt{blame}_i$), where $\texttt{blame}_i$ is a set of (j, $\texttt{proof}_{ij}$). j is the misbehavior person in i's point of view and $\texttt{proof}_{ij}$ is i's proof logs. For the purpose of accountability, each group member keeps a log file during each protocol execution. Given a step in a protocol phase, a member's log file contains all his previous incoming and outgoing messages along with the member's generated secrets (e.g., secret shares, random numbers used in encryption) during the execution up to the current step. For each protocol step, the hash value of a member's log file is sent out for the commitment purpose. We use $h_i^{\phi.k}$ to denote the hash value of member i's k-th log file in the phase ϕ. All members agree on a nonce n_R to uniquely identify one protocol execution and a cryptographic hash function, denoted as $h(\cdot)$.

Fig. 2 shows the application phase. The purpose of the application phase is to let each of the N members privately pick a unique number between 1 and N. The number will be used to assign a slot for following message submissions. The application flow is: each member i first selects a number α, generates a vector $\vec{p}$, and sets the α-th component to 1 and all other components to 0. Then the protocol adds all vectors together obtaining an integrated vector. If each member selects a unique component index, then the integrated vector is of all 1's. Otherwise, the integrated vector has some component value larger than 1, indicating the occurring of a collision. If $\vec{p}$ is of size N, the probability of collisions is expected to be high. Thus, another novelty in the application phase is that we use a sparse vector of size $M >> N$ to decrease the collision probability. When a collision happens, instead of giving up current results and redoing from the beginning, our protocol checks whether the collision number is less than a number N_c. If so, the protocol is re-executed in a special way, i.e., non-colliding members keep their α unchanged and colliding members randomly select another α. The non-collision probability of the entire phase is determined by M and N_c. We select $M = \max(361 + N, 2N^2 - 2N)$ and $N_c = 6$ obtaining a probability no less than 95%.[1]

In Step 2, each member splits the individual vector into N shares using the (N, N)-SS and sends the j-th share to member j. At the same time, member i also receives shares from other members. Utilizing the additive homomorphic property of the (N, N)-SS, i gets one share of $\vec{p}$: $[\vec{p}]_i = (\sum_{a=1}^{N}[p_{a1}]_i, \cdots, \sum_{a=1}^{N}[p_{aM}]_i)$. With the shares, the group members together reconstruct a

Phase 1: Application
Round 1:
Each member $i(1 \leq i \leq N)$ executes the following:

1) Initially, i initializes an individual position vector $\vec{p}_i = [p_{i1}, \cdots, p_{iM}]$, where $M = max\{361 + N, 2N^2 - 2N\}$. He randomly chooses a component, say $p_{i\alpha}$, and sets $p_{i\alpha}$ to 1 and the rest components to 0, indicating that he wants to occupy index α.
2) Member i splits vector $\vec{p}_i$ by dividing each component into N shares, obtaining N vectors $\vec{p}_{i1}, \cdots, \vec{p}_{iN}$ For member $l(l \neq i)$, i constructs a message $m_l^{1.2} = Con(\vec{p}_{il})||h_i^{1.1}||n_R$, and sends $Sig_{u_i}\{m_l^{1.2}\}$ to member l. $Con(\vec{p})$ concatenates the components of $\vec{p}$.
3) Upon receiving $Sig_{u_j}\{m_i^{1.2}\}$ from members $j(1 \leq j \leq N, j \neq i)$, i computes his share $[\vec{p'}]_i = \sum_{j=1}^{N} \vec{p}_{ji}$, broadcast $Sig_{u_i}\{[\vec{p'}]_i||h_i^{1.2}||n_R\}$ to all other $N-1$ members.
4) Upon receiving other members' shares, member i now has all N shares $\{[\vec{p'}]_j\}_{j=1}^{N}$ and reconstructs $\vec{p'} = \sum_{j=1}^{N}[\vec{p'}]_j$.
5) Each member i counts the collision headcount in $\vec{p'}$, i.e. $c = \sum_{p'_a > 1} p'_a$. If $c > 6$, i broadcasts "failure" to the rest of the group. Then all members repeat Round 1 again. After two failures of running Round 1, all members directly go to Phase 6. If $0 < c \leq 6$, i broadcasts "round 2" to the rest of the group. Then all group members go to Round 2, otherwise, they go to Step 9. If there is no collision, i broadcasts "success" to the rest of the group. If all other users report "success", the entire group goes to the next step.

Round 2:
Each member $i(1 \leq i \leq N)$ executes the following:

6) Since each member has $\vec{p'}$, they know which components are occupied and which ones have collisions. In vector $\vec{p'}$, if the value at index π_i is 1, then i does not change his individual vector $\vec{p}_i$; otherwise i resets all vector components to 0, randomly chooses another index except the occupied ones, and sets it to 1.
7) Repeating Step 2 to Step 4 in Round 1, i can reconstruct another final vector $\vec{p''}$.
8) i checks the number if there is any collision in $\vec{p''}$. If any collision exists, i broadcasts "collision in round 2" to the rest of the group and all members repeat Phase 1. Otherwise, i broadcasts "success in round 2" to the rest. After the second failure of running Phase 1, all members go to Phase 6. If all other users report "success", the entire group goes to the next step.
9) Assuming member i selects the α-th $(1 \leq \alpha \leq M)$ index of $\vec{p'}$ (or $\vec{p''}$), member i's final component index in the position vector is $\pi_i = \sum_{t=1}^{\alpha} p''_t$. In this way, each member obtains a unique component index which is between 1 and N.

Fig. 2. Application phase

1. The proof and intuition behind this selection are given in Section 7.

vector $\vec{p'}$ and check if there is any collision. There are three cases:

- If $(\sum_{p'_i>1} p'_i) = 0$, all members directly go to Step 9.
- If $(\sum_{p'_i>1} p'_i) > 6$, all members repeat Round 1 again.
- If $1 \leq (\sum_{p'_i>1} p'_i) \leq 6$, all members go to Round 2.

Here, p'_i is the i-th component of $\vec{p'}$. In step 9, if no collision occurs, all components in $\vec{p''}$ should be either 1 or 0. In the end, member i computes the selected component index which is the α-th index using the formula: $\pi_i = \sum_{t=1}^{\alpha} p''_t$. Note that in Phase 1-Round 1, each component is of $\lceil \log N \rceil$ bits in case that all members select the same component. However, each component can be shrunk to 3 bits in Phase 1-Round 2 so as to save space.

Now, one might think of a jamming attack, in which an adversary always fills each component of $\vec{p}$ such that the application phase cannot be ended with a non-collision vector. This is the reason why we need a blame phase, which is to expose at least one faulty member. The success probability of finding a non-collision vector is greater than 99.75% after two runs of Phase 1. Therefore, if the members still cannot find a non-collision vector, it is quite possible that some members deviate from the protocol specification. In this case, all members go to the blame phase to find out the culprit.

Phase 2: Encryption
Each member $i (1 \leq i \leq N)$ executes the following:

1) i encrypts d_i using the symmetric key encryption scheme, and gets $e_i = Enc_{k_i}(d_i)$.

Fig. 3. Encryption phase

Fig. 3 shows the encryption phase. In the encryption phase, each member i encrypts his data d_i using the symmetric key encryption scheme. The ciphertext is $e_i = Enc_{k_i}(d_i)$.

Phase 3: Anonymization
Each member $i (1 \leq i \leq N)$ executes the following:

1) i constructs an individual vector $\vec{e}_i$ by putting e_i at index π_i and 0 at others.
2) Member i divides each component of $\vec{e}_i$ into N shares. For each member l, i constructs the message $m_l^{3.2} = Con(\vec{e}_{il})||h_i^{3.1}||n_R$, and sends $Sig_{u_i}\{m_l^{3.2}\}$ to member l.
3) Upon receiving the above messages from all other members, i locally computes $[\vec{e}]_i = \sum_{j=1}^{N} \vec{e}_{ji}$, sends message $Sig_{u_i}\{Con([\vec{e}]_i)||h_i^{3.2}||n_R\}$ to all remaining $N-1$ members.

Fig. 4. Anonymization phase

Fig. 4 shows the anonymization phase. In this phase, each member i first constructs an N-dimensional message vector $\vec{e}$ with the π_i-th component being the ciphertext of e_i and the rest being 0. All members split their message vectors and send shares to other corresponding members. Thus, each member i receives $N-1$ shares. Each member i sums up those $N-1$ shares together with his own share and obtains one share of $\vec{e}$.

Fig. 5 shows the verification phase. After the reconstruction of $\vec{e}$, each member i checks whether the π_i-th component is equal to his ciphertext e_i or not. If

Phase 4: Verification
Each member $i (1 \leq i \leq N)$ executes the following:

1) Upon receiving $Sig_{u_i}\{[\vec{e}\,]_i||h_i^{3.2}||n_R\} (1 \leq j \leq N, j \neq i)$, member i locally reconstructs $\vec{e} = \sum_{j=1}^{N} [\vec{e}]_j$.
2) i checks if π_i-th component of $\vec{e}$ is his ciphertext e_i. If not, i broadcasts a message $b = \perp$; If f_i=TRUE, i broadcasts a message $b = \perp$; Otherwise, i broadcasts b=GO.
3) each member continues to next step if he receives all remaining $N-1$ GO messages. Otherwise, he goes to Phase 6.
4) Each member i picks a random integer $x_i \leftarrow_R \mathbb{Z}_p^*$ and keeps it secret, where p is the order of group $\mathbb{G}$. Member i calculates $y_i = g^{x_i}$ and broadcasts y_i. The group members calculate a common public key $y = \prod_{i=1}^{N} y_i$.
5) Each member i computes an ElGamal encryption of his symmetric key k'_i: $k'_i = g^{\gamma}||k_i y^{\gamma}$, where $\gamma \leftarrow_R \mathbb{Z}_p^*$.
6) Member i constructs an N dimensional vector $\vec{k'_i}$ with π_i-th component being k'_i and the rests being 0. Member i divides each component of $\vec{k'_i}$ into N shares, constructs the message $m_l^{4.6} = Con(\vec{k'}_{il})||h_i^{4.1}||n_R$, for each other member l, and sends $Sig_{u_i}\{m_l^{4.6}\}$ to member l.
7) Upon receiving the above message from all other group members, i locally computes $[\vec{k'}]_i = (\sum_{j=1}^{N} \vec{k'}_{ji}$, and sends $Sig_{u_i}\{Con([\vec{k'}]_i)||h_i^{4.2}||n_R\}$ to the other $N-1$ group members.

Fig. 5. Verification phase

not, member i broadcasts an alarm message $b = \perp$. If there is a $b = \perp$ message, all members go to the blame phase, otherwise all members encrypt their individual symmetric key using a distributed ElGamal encryption system [20]. With the distributed ElGamal encryption, the message is encrypted by a common public key and the decryption requires all members' secret keys. This encryption is necessary because the members will not expose their key in the blame phase if the key transmission is broken by a malicious adversary. We do not require the ElGamal encryption to be IND secure. An encryption scheme that can maintain confidentiality is sufficiently suitable for our protocol. The reason is that the plaintext, i.e., the secret key, is a random number and the resulting ciphertext is thus indistinguishable from a random number. The members secretly construct an N-dimensional key vector $\vec{k'}$ such that the π_i-th component is k'_i. To achieve this, they use the same data aggregation technique that is introduced in Section 5.2.

Fig. 6 shows the decryption phase. After the key vector is reconstructed, all members retrieve each key ciphertext k'_i from $\vec{k'}$. If all ciphertexts are not corrupted, all members publish their secret x_i so that the distributed ElGamal decryption key can be constructed to decrypt the ciphertexts of the secret keys. Finally, the obtained secret keys are used to decrypt the ciphertexts of the messages.

Fig. 7 presents the blame phase. Group members publish all secret information, including the secret choice of component index, all messages received from and sent to other members, their secret shares, message ciphertexts, and key ciphertexts, and replay the protocol execution again to find the culprit.

7 Analysis of AMS Protocol

This section analyzes the security and efficiency of AMS protocol.

Phase 5: Decryption
Each member i executes the following:

1) Upon receiving $Sig_{u_i}\{[\vec{k'}]_i||h_i^{4.2}||n_R\}(1 \leq j \leq N, j \neq i)$, member i locally reconstructs $\vec{k'} = (\sum_{b=1}^{N}\sum_{a=1}^{N}[k'_{a1}]_b, \cdots, \sum_{b=1}^{N}\sum_{a=1}^{N}[k'_{aN}]_b)$.
2) Member i checks if the π_i-th component of $\vec{k'}$ is k'_i. If not, he will send an alarm message and all members go to Phase 6, otherwise, he broadcasts x_i.
3) Upon receiving $x_j(1 \leq j \leq N, j \neq i)$, member i checks its validity by checking whether the equation $g^{x_j} = y_j$ holds. If any x_j is invalid, member i reports member j as a faulty member. Having all valid x_j's, the distributed ElGamal decryption key is $x = \sum_{j=1}^{N} x_j \mod p$. Member i retrieves a secret key $k'_j = g^{\gamma}||k_j y^{\gamma}$ from $\vec{k'}$, decrypts it, and obtains $k_j = k_j y^{\gamma}/(g^{\gamma})^x$. Having k_j, member i decrypts the corresponding ciphertext $d_{\sigma^{-1}(j)}$ from $\vec{e}$.
4) Member i broadcasts a copy of $(d_{\sigma^{-1}(1)}, \cdots, d_{\sigma^{-1}(N)})$ to the group and the collector.
5) If any of the sequence from other member doesn't match i's copy, he broadcasts $b = \perp$; otherwise he broadcasts $b = \texttt{GO}$. Members and the collector enter the blame phase if any $\perp$ message is received; otherwise they exit the protocol.

To ensure the completion of the protocol, a predefined timeout is set for waiting other member's flags.

Fig. 6. Decryption phase

Phase 6: Blame
The members may enter the blame phase from the application, verification, or decryption phase. There should be a corresponding blame for each different entrance. However, the descriptions are similar and we put them together to save space. We use *entrance phase* to denote the phase from which the members enter the blame phase. When they enter this phase, members publish as much secret information as they could and replay the protocol execution as far as they could go.
Each member executes the following:

1) If the entrance phase is Decryption, each member destroys his own secret key x_i and symmetric key k_i.
2) Each member i reveals his own secret shares when he splits vectors before entrance phase, i.e., $[\vec{p}_i]_i$, $[\vec{e}_i]_i$, and $[\vec{k'}_i]_i$. These secret shares are never sent out during the previous phases. Members also reveal all incoming and outgoing messages along with the signatures up to the entrance phase. If the entrance phase is after the encryption phase, members also need to reveal the randomness used in the encryption phase, e.g. initialization vector for AES-CTR. With these information, member i can replay any other member's behaviors and reconstruct their individual vectors.
3) Member i exposes j as faulty if
   - The hash value of any replayed message does not match the corresponding hash value in the log file;
   - Any of the replayed messages does not match the corresponding messages received in the previous phase;
   - In the application phase, the vector $\vec{p}_j$ is ill-formed, i.e., not in the form of one component being 1 and the rests being 0;
   - In the anonymization phase, the ciphertext vector $\vec{e}_j$ is ill-formed, i.e., $\exists n \in \{1, \cdots, N\}\ s.t.\ n \neq j \wedge e_n \neq 0$;
   - In the decryption phase, member j's key vector $\vec{k'}_j$ is ill-formed;
   - If the entrance phase is Anonymization or Decryption, everything is verified to be correct, but j claims that his ciphertext or key is corrupted.
   - j improperly reports $\mathtt{b}_j = \perp$ in 2 of Fig. 5.

Fig. 7. Blame phase

7.1 Security

In this section, we prove that the security properties defined in Section 2.3 can be preserved under malicious attacks.

7.1.1 Anonymity

We prove the anonymity (Definition 1) of our protocol in two aspects. First, we prove that if the collector and some members behave dishonestly and learn some associations between the identities and the ciphertexts, they cannot pass the verification phase and thus they cannot learn the plaintexts. Second, we prove that if the collector and the dishonest members behave honestly, they pass the verification phase and learn the final decrypted plaintexts, but they will not learn the associations between the identities and the plaintexts.

Theorem 2. *In the AMS protocol, if the collector colludes with no more than $N-2$ group members and the symmetric encryption is IND-CPA secure, the collector has only a negligible probability to get the associations between the messages and the identities of the honest group members.* □

Proof: Our proof is done in two parts. First, we show that in the verification phase, either there is exactly one copy of the ciphertext for each honest member, or the deviation from the protocol could be detected by the members before the collector gets the secret keys. Second, we show that an adversary who can win the anonymization game while maintaining the security properties can also win the distinguishing game, which is a contradiction because the (N, N)-SS is unconditionally indistinguishable and the underlying encryption scheme is IND-CPA secure.

Part 1: The honest members' messages cannot appear more than once due to the indistinguishable property of (N, N)-SS. If the adversary can reproduce the honest members' messages, then the adversary can break the (N, N)-SS scheme from less than N shares, which is a contradiction to the property of the (N, N)-SS.

Now we show that if the adversary modifies the honest members' messages, this modification will be detected in the verification phase, since the honest members do not find their ciphertexts in the message vector. Hence, the protocol is abort and the adversary cannot get the secret keys of the ciphertexts. In this case it is infeasible for the adversary to learn the plaintexts without the secret keys since the underlying encryption scheme is semantically secure.

Part 2: Now suppose that the adversary honestly handles all ciphertexts belonging to the honest members. If there is a probabilistic polynomial time algorithm $\mathcal{D}$ that allows this adversary to win the anonymization game with a non-negligible probability, we show how to use $\mathcal{D}$ as a subroutine to the algorithm $\mathcal{A}$ that wins the distinguishing game with a non-negligible probability. Because the underlying (N, N)-SS is unconditionally indistinguishable, this is a contradiction, and we conclude that no such $\mathcal{D}$ exists.

Let the set of $N-k$ honest group members in the anonymization game be $\mathcal{K} = \{1, \cdots, N-k\}$. Let $\mathcal{D}$ be an algorithm that allows the adversary to win the anonymization game with a non-negligible probability. Then there exist honest group members α and β such that for a negligible function in λ, $\varepsilon(\lambda)$, the advantage

$$Adv_{\mathcal{D}} > \varepsilon(\lambda).$$

To apply $\mathcal{D}$, $\mathcal{A}$ must simulate the oracle in the anonymization game to reproduce the view of the adversary. We show how $\mathcal{A}$ is able to achieve this.

Algorithm $\mathcal{A}$ begins by applying $\mathcal{D}$ to learn its choices in Step 1 of the anonymization game. $\mathcal{A}$ therefore learns the following:

- two honest participants α and β, and two plaintext messages d_0 and d_1; and
- a message d_i for each honest member i.

Then, for each honest member i, $\mathcal{A}$ chooses a secret key, k_i. $\mathcal{A}$ selects plaintext messages d_0 and d_1 for α and β, respectively.

$\mathcal{A}$ is now ready to play the role of the oracle in the anonymization game by simulating the messages of the honest members in the protocol execution. For each phase of the protocol, we explain how $\mathcal{A}$ is able to reproduce the messages sent in that phase.

Phase 1: $\mathcal{A}$ has all the necessary information to exactly reproduce this phase.

Phase 2: For all honest members other than α and β, $\mathcal{A}$ encrypts d_i using k_i, which results in the ciphertext e_i. $\mathcal{A}$ then encrypts d_0 and d_1 using the keys k_0 and k_1, respectively, getting $e_0 = E_{k_0}(d_0)$ and $e_1 = E_{k_1}(d_1)$.

Phase 3: $\mathcal{A}$ gives e_0 and e_1 to the distinguishing game oracle, getting back $[e_b]_1, \cdots, [e_b]_{N-1}$ and $[e_{\bar{b}}]_1, \cdots, [e_{\bar{b}}]_{N-1}$.

Suppose that in the application phase, α and β obtain positions π_α and π_β, respectively. $\mathcal{A}$ now constructs $N-1$ shares of $\vec{e}_\alpha$ as follows. The components at positions other than π_α are set to 0 and their shares can be easily constructed. The π_α-th component of $N-1$ share vectors are respectively filled by $N-1$ shares of e_b obtained from the distinguishing game oracle. $N-1$ shares of $\vec{e}_\beta$ are constructed in the same way except that $e_{\bar{b}}$ is used instead of e_b.

At the end of Phase 3, $\mathcal{A}$ needs to compute a share of the message vector $\vec{e}$. $\mathcal{A}$ does not have the N-th share of e_b or $e_{\bar{b}}$. But he has the original ciphertexts e_0 and e_1. With any $N-1$ shares and e_0 (resp. e_1), $\mathcal{A}$ could compute the last share such that all shares are reconstructed as e_0 (resp. e_1). At this moment, $\mathcal{A}$ randomly picks a ciphertext from e_0 and e_1, computes the last share, and puts it on the π_α-th component for a member α. The other ciphertext is used for a member β. We argue that this does not change $\mathcal{D}$'s view when it does the reconstruction. Because all honest group members' positions are chosen randomly and the position sequence is a random permutation on the set $\{1, \cdots, N\}$, the probability that e_0 appears in π_α and e_1 appears in π_β is equal to the probability that e_1 appears in π_α and e_0 appears in π_β. Therefore, randomly allocating e_0 and e_1 on π_α and π_β does not change $\mathcal{D}$'s view.

Phase 4 and Phase 5: $\mathcal{A}$ has all the necessary information to complete the phases.

Now $\mathcal{A}$ simulates the view of the adversary and applies $\mathcal{D}$ to the view. If $\mathcal{D}$ outputs 1, then $\mathcal{A}$ outputs 1; if $\mathcal{D}$ outputs 0, $\mathcal{A}$ outputs 0.

We now analyze the probabilities of $\mathcal{A}$ outputting 1 when the distinguishing oracle chooses $b = 0$ and when the distinguishing oracle chooses $b = 1$. If $b = 0$, then the view of the adversary is $(\alpha, \beta, d_0, d_1, 0)$. If $b = 1$, then the view of the adversary is $(\alpha, \beta, d_0, d_1, 1)$. Based on our assumption that $\mathcal{D}$ wins the anonymization game, we have that $Adv_{\mathcal{D}} > \varepsilon(\lambda)$. Now we make a simple substitution,

$$Adv_{\mathcal{A}} = Adv_{\mathcal{D}} > \varepsilon(\lambda).$$

We conclude that $\mathcal{A}$ can win the distinguishing game with a non-negligible probability, which contradicts with the unconditionally indistinguishable property of the (N, N)-SS. □

7.1.2 Integrity

In the verification phase, the honest members verify whether their ciphertexts appear in the message vector $\vec{e}$. If the verification fails, i.e., the ciphertext of at least one honest member has been altered, the protocol aborts. So the adversary cannot get the secret keys. If the verification succeeds, the honest members share their message decryption keys via distributed ElGamal encryption/decryption systems in the decryption phase. After all messages are decrypted, members send them to the collector. We conclude that the property defined in Section 2.3 is preserved.

7.1.3 Accountability

Accountability of an anonymous messaging system is first proposed and studied in [10]. The proof of our system's accountability follows the similar philosophy.

A member i *cannot* expose another honest member j unless i obtains evidence of j's misbehaviors that are verifiable to a third member. Accountability is maintained if

Condition I: no member can accuse an honest member;
Condition II: at the end of a protocol execution, either (a) the protocol completes successfully; (b) the protocol fails in the application phase, which is not caused by malicious attacks; or (c) at least one faulty member is exposed.

Condition I is shown as follows. An evidence of member j's misbehavior consists of some "incorrect" data content d_j signed by j at the step stp of some phase, together with all his log messages and all members' generated secret shares up to a particular step stp. Member j will be exposed as faulty only if he signed some incorrect messages or his behavior deviates from the protocol. For example, a member j has two non-zero elements in his individual encryption vector $\vec{e}_j$ to block another member's message. However, this contradicts the assumption that the member j is honest. A member i may collude with another member k to accuse an honest member j by forging a message signed by the member k prior to the step stp and the message is different from the one received and used in generating j's message m_j. The message received by j must be valid because

j will check against k's signature. However, since each message is signed and contains a unique nonce, two valid messages at the same step will also expose k as faulty, which weaken i's accusation against j.

Since the application phase cannot guarantee to always find each member a unique position, our AMS protocol may fail even when all members are honest. However, as shown in the following section, the failure probability of two application executions is no more than 0.25%, which is small enough to omit in practice. When the application fails, members still enter the blame phase to check if it is a failure caused by malicious attacks or not. If it is, a faulty member will be exposed.

We now show that our AMS protocol satisfies condition II(c), i.e., a faulty member is exposed in the blame phase. A member enters the blame phase if a) the application phase fails twice; b) some members report that his ciphertext is manipulated in the verification phase; or c) some members report that his encryption key is manipulated in the decryption phase. In case a), the blame phase requires members to recover their secret shares and log messages in the previous two attempts. This enables a member to reconstruct each member's position vector and a faulty member will be immediately exposed. In cases b) and c), all members recover their secret shares and log messages up to the current step. Also it will expose a faulty member or a member i himself if he untruthfully reports that his information is manipulated.

7.2 Efficiency

In this section, we analyze the success probability of finding a non-collision vector in Phase 1, and the communication rounds.

7.2.1 The Success Probability in Phase 1

Theorem 3. *In the application phase, at the end of round 2, the success probability of finding a non-collision vector $\vec{p}$ is greater than 95%.* □

Proof: Let η_{ij} denote the event that members i and j choose the same position, leading to a collision. Let ϕ_i^w denote the event that a member i chooses position $w (1 \leq w \leq M)$. Because all members choose their positions independently at random, the probability $p(\phi_i^w) = 1/M$. Using Bayes' theorem, we have

$$p[\eta_{ij}] = \sum_{w=1}^{M} p[\phi_i^w | \phi_j^w] p[\phi_j^w] = \sum_{w=1}^{M} \frac{1}{M} p[\phi_j^w] = \frac{1}{M}.$$

We define an indicator random variable X_{ij} such that $X_{ij} = 1$ if the event η_{ij} happens and $X_{ij} = 0$ if η_{ij} does not happen. Since X_{ij} only takes the value 0 or 1, it is a Bernoulli variable. We define $\mathcal{X}$ to be the number of collisions, i.e. $\mathcal{X} = \sum_{i<j} X_{ij}$. Then we compute

$$E[\mathcal{X}] = E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}]$$

$$= \sum_{i<j} p[X_{ij} = 1] = \sum_{i<j} \frac{1}{M} = \frac{1}{M} \binom{N}{2}.$$

Assume that $E[\mathcal{X}] = \mu$, from the above equation, we get $\mu = N(N-1)/2M$. If we choose $M = 2N^2 - 2N$, then $\mu = 1/4$. Utilizing the Chernoff bound [8], we have

$$p[\mathcal{X} \geq (1+\delta)\mu] \leq (\frac{e^\delta}{(1+\delta)^{1+\delta}})^\mu$$

for any $\delta > 0$. Let $\delta = 11$, we get $(1+\delta)\mu = 3$ and $p[\mathcal{X} \geq 3] \leq (\frac{e^{11}}{(1+11)^{1+11}})^{0.25} \approx 0.009$. Hence, at the end of Phase 1-Round 1, the probability that there exist no less than 6 collisions is not greater than 0.9%. In Phase 1-Round 2, since there are at most 6 members who collide with each other, we need to have enough positions for the colliding members to choose from. Suppose the positions left at the end of round 1 is M_l, the probability that 6 members' positions do not collide is $p = \frac{M_l(M_l-1)(M_l-2)\cdots(M_l-5)}{M_l^6}$. We let $p > 0.96$, and get the smallest integer which is 361. So the lower bound of M is $361 + N$. We define ϱ to be the event that the members find a non-collision vector in Phase 1. $p[\varrho] \geq p[\text{succeed in round 1} \wedge \text{succeed in round 2}] \geq (1 - 0.009) \times 0.96 = 0.95136$. The theorem is proved. □

We define the total number of execution times of Phase 1 to T. The expectation $E[T] = \sum_{i=1}^{\infty} i(1-p[\varrho])^{i-1} p[\varrho] = 1/p[\varrho]$. Since $p[\varrho] > 95\%$, the average number of executions needed to find a non-collision vector is $\lceil 1/0.95 \rceil = 2$. In fact, the failure probability is less than $5\% \times 5\% = 0.25\%$ after two executions of the application phase.

7.2.2 Communication Rounds

All phases are parallelizable. In Phase 1, the number of communication rounds is 4. There is no communication in Phase 2. Phase 3 needs 2 communication rounds. Phase 4 needs 4 communication rounds. Finally, there are 2 communication rounds in Phase 5. The expected total number of communication rounds is $4/p[\varrho] + 8$, which is approximately 12 and independent of the group size. In contrast, in Brickell and Shmatikov's [4] protocol, the total communication rounds is $2N + 7$. Note that our protocol is probabilistic, while the protocol in [4] is deterministic.

We note that we have ruled out the case where multiple malicious users collude, corrupt each other's messages, but do not report their misbehaviors. Since the corruption is within malicious members, it does not modify honest members' messages. We leave the detection of such mishaviors within colluded members to our future works.

8 Bulk Protocol

On top of the shuffle protocol [25], the bulk protocol is constructed to handle variable length messages. The bulk protocol was first proposed in [10] and improved in [25]. The anonymity property of the entire protocol is guaranteed by the shuffle protocol, which is replaced by the AMS protocol in this work. For completeness of the protocol, we give the description of the bulk protocol here. More details of its analysis can be found in [25].

For each variable length message, BULK builds a fixed length message extractor. Then, BULK uses AMS to shuffle all message extractors and obtains a permutation of message extractors. Finally, BULK concatenates all messages in the permuted order and gives it to the collector.

Each member i has a signature key pair (u_i, v_i) and another encryption key pair (x_i, y_i), where the first one is the private key, i.e. singing key or encryption key, and the second one is the public key, i.e. verification key or decryption key. The bulk protocol is executed in 3 stages: Setup, Anonymize, and Blame, as per [25]. *Setup* is using each member's v_i to generate three nonces: n_R identifying a run of Anonymize, n_{R_1} identifying a run of AMS, and n_{R_2} identifying a run of AMS in blame. This nounce generation may be achieved via standard consensus tools, e.g. Paxos [19]. To obtain a copy of shuffled messages, the collector participates the BULK protocol with an empty message.

In the message extractor generation (Step 2), each member i generates a message extractor $d_i = (L_i, h(msg_i), \vec{H}_i, \vec{S}_i)$ which is used to recover the message. Here, $\vec{H}_i$ is the vector of hash values generated from $h(C_{ij})(1 \leq j \leq N, j \neq i)$, where C_{ij} is the pseudo random number used to XOR member i's message. $\vec{S}_i$ is a vector of the ciphertexts of the pseudo random generator seeds. The message extractor consists of all the information needed to recover the original message. The details of BULK protocol are presented as follows:

Anonymize:

Step 1a: Each member i chooses an encryption key pair (x_i, y_i) and broadcasts $\mu_{i1a} = Sig_{u_i}(y_i||n_R||1a||i)$.

Step 1b: Upon receiving other member's public keys, member i broadcasts all the keys she receives: $\mu_{i1b} = Sig_{u_i}\{\vec{K}_i, n_R, 1b, i\}$, where $\vec{K}_i = [\mu_{11a}, \ldots, \mu_{N1a}]$.

Step 2: Member i generates a message extractor d_i for his message msg_i. The extractor is of fixed length Δ. Member i sets L_i to be the message length or 0 if she does not want to send any message. Member i verifies each set of public key $\vec{K}_j$ she received to ensure that she has the same public keys, if:

case 1 each key set contains the same valid public keys, then i chooses a random seed s_{ij} for each member j and generates an L_i-bit random number $C_{ij} = RAND(L_i, s_{ij})$ $(i \neq j)$. Member i XORs his message with all C_{ij}'s and obtains C_{ii}, $C_{ii} = C_{i1} \oplus \cdots \oplus C_{i(i-1)} \oplus msg_i \oplus C_{i(i+1)} \oplus \cdots \oplus C_{iN}$. Member i computes $H_{ij} = h(C_{ij})$, encrypts each seed s_{ij}, $S_{ij} = Enc_{y_i}(s_{ij}, r_{ij})$, using member j's public key and a secret random number r_{ij}. Member i sets S_{ii} to a random number. Member i collects all hash values and seed ciphertexts to form two vectors $\vec{H}_i = \{H_{i1}, \cdots, H_{iN}\}$, $\vec{S}_i = \{S_{i1}, \cdots, S_{iN}\}$, and gets the message extractor $d_i = \{L_i, h(msg_i), \vec{H}_i, \vec{S}_i\}$. The length of the extractor is Δ.

case 2 key verification fails, member i creates an empty message extractor $d_i = 0^{\Delta}$.

case 3 member i has no message to send, he sets $L_i = 0$ and random values to $\vec{H}_i$ and $\vec{S}_i$. Then i forms his extractor $d_i = \{L_i, h(msg_i), \vec{H}_i, \vec{S}_i\}$.

Step 3: Message extractor distribution. Each member i runs AMS protocol in Section 6 with input $(d_i, \vec{K}, n_{R_1}, f_i)$, where i sets f_i=TRUE if he creates an empty message in the previous step and sets f_i=FALSE otherwise. Message descriptors are the messages to be shuffled in AMS. if AMS succeeds, Each member i gets a sequence of message extractors with d_i appearing at π_i-th position. If AMS fails, the protocol outputs (fail, $\texttt{blame}_i$) and member i saves $\texttt{blame}_i$. If i sets f_i=TRUE, he prepares a proof of a dishonest member j and broadcasts it to other members.

case 1 If j sent an invalid key, proof=(j, c_5, μ_{j1a}), where c_5 denotes the failed check of all 6 checks [25] in BULK.

case 2 If j equivocated, proof=$(j, c_6, \mu_{j1a}, \mu'_{j1a})$, where μ_{j1a} is the message received by i and μ'_{j1a} is the message contained in some other's $\vec{K}_m$.

If there is more than one misbehaving member, i randomly chooses one to blame. Then, i broadcasts $\mu_{i3} = Sig_{u_i}\{\texttt{proof}, n_R, 3, i\}$.

Step 4: If previous AMS fails, member i shares his blame set $\texttt{blame}_i$, sets $\texttt{GO}_i$=FALSE, and broadcasts $\mu_{i4} = Sig_{u_i}\{\texttt{GO}_i, \texttt{blame}_i, n_R, 4, i\}$.

If AMS succeeds, i sets $\texttt{GO}_i$=TRUE. i now holds a message extractor sequence $D = (d_{\sigma^{-1}(1)}, \cdots, d_{\sigma^{-1}(N)})$, where $(d_{\sigma^{-1}(1)}, \cdots, d_{\sigma^{-1}(N)})$ is a permutation of $(d_1, \cdots, d_N)$. Member j recognizes his own message extractor $d_{\sigma^{-1}(i)}$ in D, and sets $C'_{\sigma^{-1}(i)i} = C_{ii}$. For each $d_j \in D$ $(j \neq \sigma^{-1}(i))$, member i extracts i-th component S_{ji} from $\vec{S}_j$ and decrypts S_{ji} using his private key x_i obtaining the seed s_{ji}.

Member i computes the pseudo random number $C_{ji} = RAND(L_j, s_{ji})$ and checks $h(C_{ji})$ against H_{ji}. If the check has passed, i sets $C'_{ij} = C_{ij}$. If S_{ji} is not a valid ciphertext, s_{ji} is not a valid seed, or hash H_{ji} is not valid, j sets C'_{ji}=NULL.

Member i assembles all C'_{ji}'s in the received order, i.e. putting C'_{ji} at the j-th position, and broadcasts the message $u_{i4} = Sig_{u_i}\{\texttt{GO}_j||C'_{1i}||\cdots||C'_{Ni}||n_R||4||j\}$ to the group.

Step 5: Each member i notifies the remaining members about the outcome of the previous step.

case 1 If there is one or multiple $\texttt{GO}_j$=FALSE, member i forms a vector $\vec{V}_i$ =[$\mu_{j4}|$ $\texttt{GO}_j$=FALSE].

case 2 If $\texttt{GO}_j$=TRUE for each member, member i checks each C'_{kj} he receives from every other member j against H_{kj} from d_k.

If C'_{kj} is empty or the hash value is not valid, member i notifies this fact to other group members by adding each corrupted μ_{j4} into vector $\vec{V}_i$.

If $\texttt{GO}_j$ = TRUE for each member and all ciphertexts are correct, member i sets $\vec{V}_i = NULL$.

In each case, i broadcasts $\mu_{i5} = Sig_{u_i}\{\vec{V}_i, n_R, 5\}$ to the entire group.

Step 6:

1) If $\texttt{GO}_j = \texttt{TRUE}$ for each member The collector checks the hash value of each C'_{ij} from member j against H_{ij} from $d_i \in D$. If C'_{ij} is $NULL$ or $h(C'_{ij}) \neq H_{ij}$, the message on the i-th position is corrupted and ignored.
2) For each uncorrupted slot, the collector reconstructs the data by computing $d_i = C'_{i1} \oplus \cdots \oplus C'_{iN}$.

Blame:

Step 7: If i noticed his descriptor d_i is corrupted but received $\texttt{GO}_j = \texttt{TRUE}$, then i generates an accusation $A_i = \{j, \pi(i), s_{ij}, R_{ij}\}$ to blame j. If there are multiple j's, i randomly picks one. If there is no such j, i sets $A_i = 0^{\delta_a}$, where δ_a is the fixed length of accusations.

Member i compares each message μ'_{j4} that he received in Step 5 with the message that he received directly from j in Step 4. If any mismatch happens, member i sets $f_i = \texttt{TRUE}$ to cause AMS to fail in order to inform other members about equivocation. Otherwise, he sets $f_i = \texttt{FALSE}$.

Member i runs AMS(A_i, K, n_{R_2}, f_i). For a member who sets $f_i = \texttt{TRUE}$ because of conflicting of μ'_{j4} and μ_{j4}, i creates a proof of j' equivocation: $p'_i = (j, c_1, \mu_{j4}, \mu'_{j4})$ If $f_i = \texttt{FALSE}$, i sets $p'_i = 0$. Member i then broadcasts $\mu_{i7} = Sig_{u_i}\{p'_i, n_R, 7, i\}$.

Let O_i be the output of AMS protocol for i. After receiving all μ_{j7} from others, i executes the followings:

Case 1 $O_i = (\texttt{fail}, \texttt{blame}_i^{\texttt{A}})$, i sets $\texttt{success}_i = \texttt{FALSE}$. i exams each $\texttt{proof}_j$ in $\texttt{blame}_i^{\texttt{A}}$. If $\texttt{proof}_j$ is not due to improperly reporting b_j (Fig. 7), j has no justifiable way to terminate AMS and i puts (j, c_2) into bulk's blame $\texttt{blame}_i^{\texttt{B}}$. If μ_{j7} contains two versions of the same C'_{lk} for some member k, then i adds (k, c_1) to $\texttt{blame}_i^{\texttt{B}}$. Otherwise, he adds (j, c_2) to $\texttt{blame}_i^{\texttt{B}}$.

Case 2 $O_i = (\texttt{success}, A')$ and $\vec{V}_i$ is empty. i sets $\texttt{success}_i = \texttt{TRUE}$.

Case 3 $O_i = (\texttt{success}, A')$ and $\vec{V}_i$ includes ciphertexts. i checks the validity of each accusation in the received A'. If accusation A_j is valid, i puts (j, c_3) into $\texttt{blame}_i^{\texttt{B}}$. If there is no valid accusation targeting an incorrect ciphertext received by i, he sets $\texttt{success}_i = \texttt{TRUE}$. Otherwise, he sets $\texttt{success}_i = \texttt{FALSE}$.

Case 4 $O_i = (\texttt{success}, A')$ and $\vec{V}_i$ contains $\texttt{GO}_j = \texttt{FALSE}$. i sets $\texttt{success}_i = \texttt{FALSE}$. Member i checks each $\texttt{GO}_j$ in $\vec{V}_i$. i checks validity of $\texttt{blame}_j$, that blames some k's misbehaviors, of Step 4. Since i has all log files and necessary secrets in the failed AMS run, the validity checking is performed by replaying the protocol and checking each step. If it is not valid, i puts (j, c_4) in $\texttt{blame}_i^{\texttt{B}}$. Otherwise, i exams each $\texttt{proof}_k \in \texttt{blame}_j$. If $\texttt{proof}_k$ is not due to improperly reporting b_k, then i puts k into $\texttt{blame}_i^{\texttt{B}}$. Otherwise, i needs to check μ_{k3} to see if k justifiably causes AMS failure to expose l in Step 3. If l sent an invalid key, i adds (l, c_5) to $\texttt{blame}_i^{\texttt{B}}$. If l equivocated, i adds (l, c_6) to $\texttt{blame}_i^{\texttt{B}}$. Otherwise, i adds (j, c_2) to $\texttt{blame}_i^{\texttt{B}}$.

If $\texttt{success}_i = \texttt{TRUE}$, member i gets shuffled messages. Otherwise, he gets a blame list $\texttt{blame}_i$ and corresponding logs.

9 Performance Evaluation

We made a proof-of-concept implementation of our protocol, and compared it with the protocols in [4] and [10]. All the experiments were carried out on an Intel® Core™2 Duo CPU P8600 @ 2.40GHz computer. We used Crypto++ [12] as our underlying cryptographic library.

Dissent [10] uses the same technique as the one in [4] in its shuffle protocol, which requires an IND-CCA2 secure cryptosystem. Since Crypto++ does not have any IND-CCA2 secure algorithm in the standard model, we implemented the Cramer-Shoup cryptosystem by using SHA-1 and a $3,072$-bit integer group, MODP-15 [18]. To compare with the implementation in [10], we also implemented a shuffle based on RSA-OAEP [2], which is CCA-2 secure only in the random oracle model. The modulus for RSA is of $3,072$ bits. The symmetric key cryptosystem in AMS protocol was implemented by using AES-128 in counter (CTR) mode, which is CPA secure in the standard model [1]. The distributed ElGamal encryption used in AMC protocol is also built on top of MODP-15 group. All the modules were implemented in C++ and compiled by g++ 4.4.3 with `-O3` level optimization.

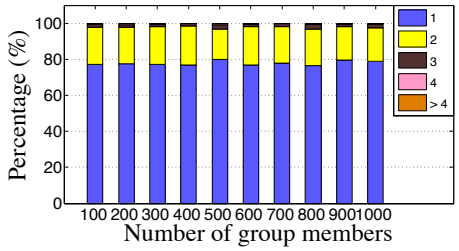


Fig. 8. The distribution of needed rounds in Application

First we tested the number of rounds needed in Phase 1. We increased N from 100 to $1,000$ using an increment of 100. For each value of N, we ran Phase 1 for $1,000$ times and recorded the number of rounds needed to find a non-collision vector $\vec{p}$. We remark that one Application execution contains two rounds. Fig. 8 shows the result. We observe that at least 77% of the tests were completed in 1 round no matter what N is. Furthermore, 97% of them were completed within 2 rounds, finding a non-collision vector $\vec{p}$.

Next, we compared the computational overhead of our protocol to that of the shuffle protocol in [4]. We used "C-shuffle" and "R-shuffle" to denote the Cramer-Shoup based and the RSA based shuffle implementations, respectively. To obtain a point in a plot, we ran the corresponding simulation for 10 times and calculated the average over the 10 runs. Also, we used box plot to visualize the distribution of the 10 results. The top and bottom bars of a box plot represent the max and min values in the collection. A squeezed box denotes that the

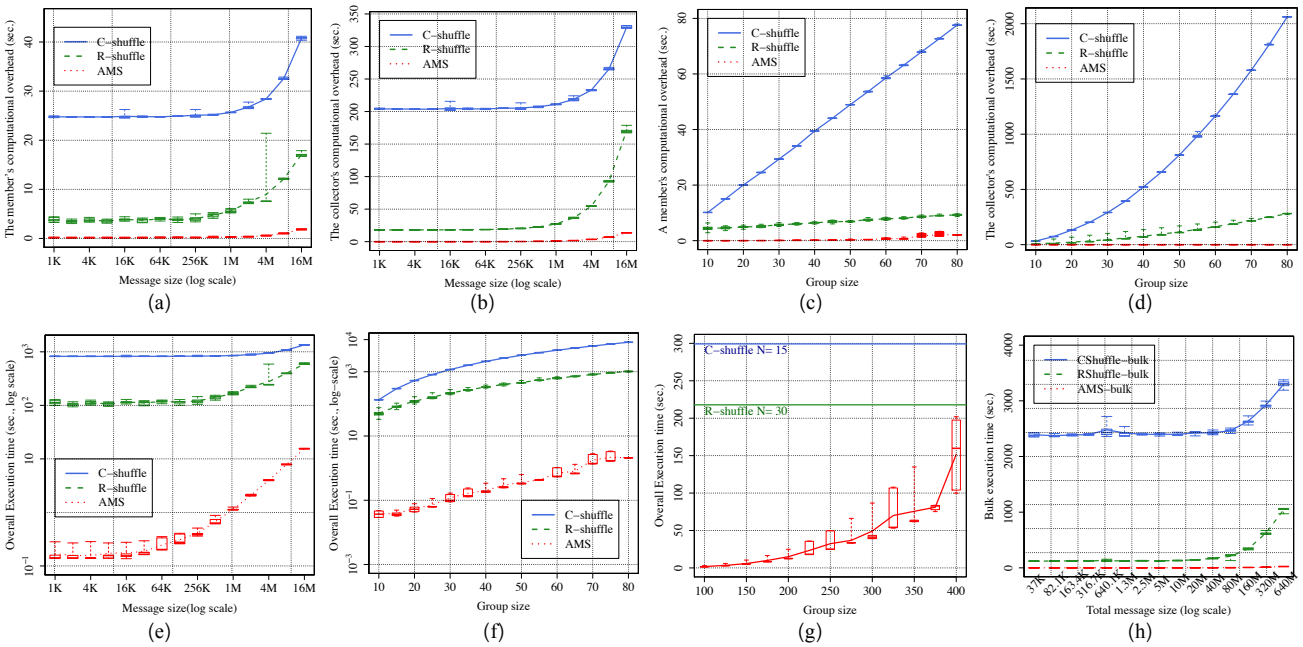


Fig. 9. Protocol computational overhead and execution time. (a) A member's computational overhead vs. message size; (b) Collector's computational overhead vs. message size; (c) A member's computational overhead vs. group size; (d) Collector's computational overhead vs. group size; (e) System execution time vs. message size; (f) System execution time vs. group size; (g) AMS Execution time for scalable groups; (h) Bulk execution time vs. message size

data points are close to the average value. Fig. 9(a)-9(d) show the computational overheads for the collector and the members under different group sizes N and message sizes l. Note that the little bar on each line is a squeezed box plot. First we fixed $N = 40$, changed l from 1KB to 16MB, and compared the computation time needed for each member and the collector in different anonymous message systems. Note that in Fig. 9(b), the computation time of the collector in our protocol is much shorter than that in C-shuffle and R-shuffle. This is because the decryption time of an IND-CCA2 secure cryptosystem is much shorter than that of a symmetric key decryption algorithm. In addition, the shuffle protocol needs N^2+N decryptions on the collector, while the collector in our protocol only needs to decrypt N ciphertexts. From Fig. 9(a) and 9(b), we observe that AMS performs better than the two shuffle implementations, especially when the message size goes big. In Fig. 9(c) and 9(d), we fixed l to 1KB, and changed N from 10 to 80 at an increment of 10. Since the number of communication rounds depends on N in the shuffle protocol, the execution time of each member increases with N, while the execution time of each member in our protocol increases slowly. In both sub-figures, for the shuffle implementations, the execution time increases polynomially in N. However, the execution time of the collector in our protocol is almost constant, which shows that the addition and symmetric key decryption is so fast that we cannot tell the time difference when the group size is small. We remark that a party's computational overhead in our protocol is not the protocol's execution time. In the AMS system, a member's computation is in parallel with other members, and the protocol is finished within constant rounds while the shuffle protocol needs $O(N)$ communication rounds. We next show the execution time for C-shuffle, R-shuffle, and AMS implementations in Fig. 9(e) and 9(f).

From Fig. 9(e) and Fig. 9(f), we observe that the execution time of AMS is much shorter than that of the shuffle implementations. We also ran the simulation of AMS under larger group sizes, from 100 to 400. The results are plotted in Fig. 9(g). The blue and green lines in the plot denote the execution time of C-shuffle under a 15-member group and R-shuffle under a 30-member group, respectively. We observe that the execution time of AMS was much faster even when the group size was increased to 400. The longest time cost by AMS in the tests when the group size was 400 is still better than the average time of R-shuffle when the group size was 35.

To compare end-to-end performaces under different shuffles, i.e., C-shuffle, R-shuffle, and AMS, we implemented Bulk protocol in [25]. Comparing to the previous design [10], the extra overhead of the new Bulk protocol is due to the execution of shuffle protocol in Blame stage (Section 3.4 of [25]). The shuffle protocol is shuffling fixed-length accusation messages. We note that in our evaluation the accusation message length is 10 bytes, where 1 byte for member index, 1 byte for $\pi(j)$, 4 bytes for random seed, and 4 bytes for randomness used in encrypting the seed. We replaced Bulk protocol's shuffle with AMS, C-shuffle, and R-shuffle, respectively, and compared their execution times. The comparison result

is shown in Fig. 9(h).

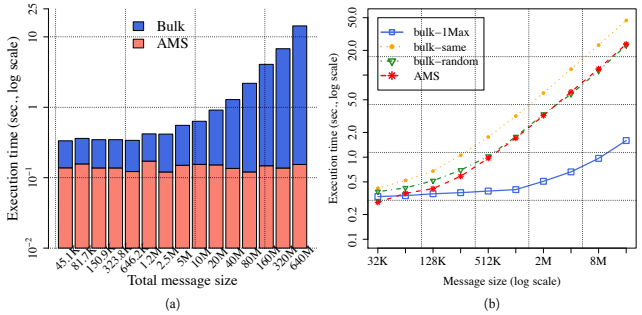


Fig. 10. Bulk performance using AMS. (a) Time cost on AMS and Data Transmission; (b) Comparison between AMS and Bulk

To do the comparison, we fixed the group size as 40 and let each member randomly pick a message size from $[l-1K, l+1K]$, where $l = 1K, 2K, 4K, \cdots, 8M, 16M$. The message size is in terms of bytes. After all members obtained their message size, they executed bulk protocol to submit the messages and repeated the protocol execution for 10 times. In Fig. 9(h), the x-axis represents the total message size and the y-axis represents the bulk execution time. We observe that AMS performs significantly better than the shuffle protocols. For example, to handle a total message size of 640MB, the average running time is 3003.97471 seconds for C-shuffle, 1040.87252 seconds for R-shuffle, and 23.89172 seconds for AMS. We also note that the difference between AMS and R-shuffle is not remarkably big when the transmitted message is of small size. This is because the data transmission in bulk will cost more time than shuffling message extractors when the transmitting data size is small. Hence, we are curious about the time cost on sending message extractors and data transmission by using AMS as a sub-protocol. Again, we fixed group size as 40 and let each member randomly pick their message size in the same way as in the previous test. We separately recorded the time cost on AMS and bulk data transmission and show the result in Fig. 10(a). The y-axis represents the time cost and is in log-scale. First, we observe that the time cost on AMS is almost the same, indicating that transmitted message extractors size is independent of the transmitted total message size. Besides, we note that the time cost on AMS is nearly 0.1 seconds while that on bulk data transmission increases to over 500 seconds.

We introduced the bulk protocol because AMS itself cannot handle variable length message transmissions. If we want to use AMS as a stand-alone system to transmit variable length messages, we have to pad the messages to a fixed maximum length over all to-be-sent messages. Since AMS is so fast, we are curious about its performance against the bulk protocol. To this end, we set up another experiment in the following way. First, we fixed the group size as 40 and set $l = 32K, 64K, 128K, \cdots, 8M, 16M$. For each l, we ran bulk protocol, using AMS as a sub-protocol, under 3 different settings: 1Max, same-length, and random-length. In the 1Max setting, one member set the message length as l and the remaining members set their message length as $l/1024$; in the same-length setting, each member set his message length as l; in the random-length, each member set his message length as a random number $rand$ such that $0 < rand \leq l$. Each member set the message size as l when we ran AMS as a stand-alone system. The result is shown in Fig. 10(b). From the figure, we observe that AMS alone took shorter time than that of the bulk protocol under the same-length setting. We also note that AMS (represented by the red dash line) cost almost the same time as that of the bulk protocol under the random-length setting (represented by the blue doted line). At last, it is obvious that AMS cost more time than that of the bulk protocol under the 1Max setting. This is reasonable and expected because under the 1Max setting, it is actually a special circumstance favoring the bulk protocol. In this case, the AMS system has to do $N-1$ paddings and transmitting more unnecessary contents. Therefore, we suggest to use AMS directly when the sizes of submitted messages are close to one another. This situation is common in the real world life. Online surveys introduced in our introduction section is one of such examples. Answers to a set of questionnaires usually have similar lengths because a set of questionnaires consist of either multiple-choice questions or short comments.

10 Conclusion

In this paper, we have proposed an efficient online anonymous message submission protocol. Utilizing the simplified secret sharing scheme, we have designed a novel position application technique, in which all members secretly select their positions in a position vector, such that a member knows nothing about the other members' message positions. We have introduced a data aggregation technique, in which all members aggregate their messages into a message vector and submit it to the collector without exposing their identities. We have theoretically proved that our protocol is anonymous under malicious attacks. We have also demonstrated the efficiency of our protocol through rigorous evaluations and experiments.

Acknowledgement

We thank the editor and the anonymous reviewers whose comments on an earlier version of this paper have led to improved presentation of this paper.

References

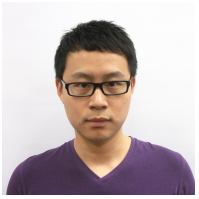
[1] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *FOCS*. Published by the IEEE Computer Society, 1997, p. 394.

[2] M. Bellare and P. Rogaway, "Optimal asymmetric encryption-how to encrypt with RSA," in *Advances in Cryptology-Eurocrypt 94*, vol. 94, 1994, pp. 92–111.

[3] J. Benaloh, "Secret sharing homomorphisms: Keeping shares of a secret secret (extended abstract)," in *Advances in Cryptology-CRYPTO 86*. Springer, 1987, pp. 251–260.

[4] J. Brickell and V. Shmatikov, "Efficient anonymity-preserving data collection," in *Proceedings of the 12th ACM SIGKDD*. ACM, 2006, pp. 76–85.
[5] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.
[6] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of cryptology*, vol. 1, no. 1, pp. 65–75, 1988.
[7] D. Chaum and E. Van Heyst, "Group signatures," in *Advances in Cryptology EUROCRYPT 91*. Springer, 1991, pp. 257–265.
[8] S. Chawla. (2004) Lecture notes of randomize algorithms: Chernoff bounds. http://www.cs.cmu.edu/afs/cs/academic/class/15859-f04/www/scribes/lec9.pdf.
[9] S. S. Chow, J. K. Liu, and D. S. Wong, "Robust receipt-free election system with ballot secrecy and verifiability." in *NDSS*, 2008.
[10] H. Corrigan-Gibbs and B. Ford, "Dissent: accountable anonymous group messaging," in *ACM CCS*. ACM, 2010, pp. 340–350.
[11] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford, "Proactively accountable anonymous messaging in verdict," in *USENIX Security*, 2013.
[12] Crypto++, http://www.cryptopp.com/, 2010.
[13] R. Dingledine and P. Syverson, "Reliable mix cascade networks through reputation," in *Financial Cryptography*. Springer, 2003, pp. 253–268.
[14] M. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004, pp. 1–19.
[15] O. Goldreich, "Secure multi-party computation," *Working Draft*, 2000.
[16] P. Golle and A. Juels, "Dining cryptographers revisited," in *Advances in Cryptology-Eurocrypt 2004*. Springer, 2004, pp. 456–473.
[17] J. Iwanik, M. Klonowski, and M. Kutyłowski, "Duo-onions and hydra-onionsfailure and adversary resistant onion protocols," in *Communications and Multimedia Security*. Springer, 2005, pp. 1–15.
[18] T. Kivinen and M. Kojo. (2003) More modular exponential (modp) diffie-hellman groups for internet key exchange (ike). [Online]. Available: http://tools.ietf.org/html/rfc3526#section-4
[19] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.
[20] L. Li, X. Zhao, G. Xue, and G. Silva, "Privacy preserving group ranking," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. IEEE, 2012, pp. 214–223.
[21] C. A. Neff, "A verifiable secret shuffle and its application to e-voting," in *Proceedings of the 8th ACM conference on Computer and Communications Security*. ACM, 2001, pp. 116–125.
[22] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.
[23] E. Sirer, S. Goel, M. Robson, and D. Engin, "Eluding carnivores: File sharing with strong anonymity," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM, 2004, pp. 19–es.
[24] D. Stinson, *Cryptography: theory and practice*. CRC press, 2006.
[25] E. Syta, H. Corrigan-Gibbs, S.-C. Weng, D. Wolinsky, B. Ford, and A. Johnson, "Security analysis of accountable anonymity in dissent," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 1, p. 4, 2014.
[26] P. Syverson, D. Goldschlag, and M. Reed, "Onion routing for anonymous and private internet connections," 1999.
[27] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr, "Towards an analysis of onion routing security," in *Designing Privacy Enhancing Technologies*. Springer, 2001, pp. 96–114.
[28] M. Waidner, B. Pfitzmann *et al.*, "The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability." Springer-Verlag, 1989, p. 690.
[29] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson, "Dissent in numbers: Making strong anonymity scale," *10th OSDI*, 2012.
[30] Z. Yang, S. Zhong, and R. Wright, "Anonymity-preserving data collection," in *ACM SIGKDD*. ACM, 2005, pp. 334–343.
[31] A. L. Young and M. Yung, "The drunk motorcyclist protocol for anonymous communication," in *Communications and Network Security (CNS), 2014 IEEE Conference on*. IEEE, 2014, pp. 157–165.
[32] X. Zhao, L. Li, G. Xue, and G. Silva, "Efficient anonymous message submission," in *Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012*, 2012, pp. 2228–2236.

Xinxin Zhao (Student Member 2009) received the B.S. degree from Xidian University, Xi'an, China, in 2006, and M.S. degree from University of Science and Technology of China, Hefei, China, in 2009. She received a Ph.D. in Computer Science from Arizona State University in 2015. Her research interests include social network privacy, location based social network privacy, and identity authentication.



Lingjun Li (Student Member 2009) received the B.S. degree from Hefei University of Technology, Hefei, China, in 2006, and the M.S. degree from University of Science and Technology of China, Hefei, China, in 2009. He received a Ph.D. in Computer Science from Arizona State University in 2014. His research interests include network security, computing privacy, and authentication.

Guoliang Xue (Member 1996, Senior Member 1999, Fellow, 2011) is a Professor of Computer Science at Arizona State University. He received the PhD degree in Computer Science from the University of Minnesota in 1991. His research has been supported by the US Army Research Office (ARO) and the US National Science Foundation (NSF). He is the Area Editor of IEEE Transactions on Wireless Communications for the Wireless Networking area, and an Editor of IEEE Network Magazine. He is a TPC Member of ACM CCS'2015, a TPC Member of AsiaCCS'2015, an Area TPC Chair of IEEE CNS'2015, and the Vice President for Conferences of the IEEE Communications Society (2016-2017). He is an IEEE Fellow.

Gail-Joon Ahn (Senior Member, IEEE) is a Professor of Computer Science and Engineering in the School of Computing, Informatics, and Decision Systems Engineering and the Director of Center for Cybersecurity and Digital Forensics at Arizona State University. His research has been supported by National Science Foundation, Department of Defense, Office of Naval Research, Army Research Office, Department of Justice, and private sectors. He is a recipient of Department of Energy Early Career Investigator Award and the Educator of the Year Award given by the Federal Information Systems Security Educators Association in 2005. He received the Ph.D. degree in information technology from George Mason University, Fairfax, Virginia in 2000.