Leveraging Smart Contracts for Secure and Asynchronous Group Key Exchange Without Trusted Third Party

Victor Youdom Kemmoe, Yongseok Kwon, Rasheed Hussain, *Senior Member, IEEE*, Sunghyun Cho, and Junggab Son, *Senior Member, IEEE*

Abstract—Group Key Exchange (GKE) is an important tool to develop secure multi-user applications such as group text messages, ad-hoc networks, and so on. Most of the currently deployed GKE schemes are synchronous, i.e., they require all the participants to be online during their execution. However, with more battery-powered devices being used in such applications, the synchronicity requirement is challenging to fulfill. To fill the gaps, asynchronous GKE schemes have been introduced in the literature. Nevertheless, the currently available asynchronous and synchronous GKE schemes rely on Trusted Third Parties (TTPs) for key establishment and management. To this end, reliance on TTPs is a serious shortcoming since TTPs are well known to be the single point of failure. Furthermore, the existing GKE schemes require participants to perform all computations, which can degrade the performance of resource-constrained devices such as Internet of Things (IoT) devices. To solve these problems, in this paper, we propose an asynchronous GKE scheme that uses blockchain and smart contracts to store the security keys-related material and reduce the computational load of the participants. Furthermore, our proposed scheme provides Perfect Forward Secrecy (PFS) and Post-Compromised Security (PCS). Our implementation on Ethereum shows that the proposed scheme can scale to more than 100 participants when combined with a distributed storage system.

Index Terms—Asynchronous GKE, blockchain, group key exchange (GKE), Internet of Things, perfect forward secrecy (PFS), post-compromised security, security, smart contract

1 Introduction

Group Key Exchanges (GKEs) are protocols that allow two or more participants to agree on a common secret key (session key) over an insecure communication channel in the network in such a way that one participant cannot derive the session key without the contribution of others. If a GKE protocol ensures that only involved participants can derive the session key, it is said to be *authenticated*. GKE is one of the core components in the security of multi-user systems such as group text message, ad-hoc network applications, Internet of Things (IoT), and so on. To date, many synchronous GKE schemes have been introduced in the literature [2], [3]. One of the limitations of the synchronous schemes is that they require all the participants to be always online during the execution of the GKE protocol. However, this requirement is difficult to fulfill in the presence of abundance of battery-powered devices that can enter into hibernation mode to prolong their lifetime [4]. Therefore, such resource-constrained devices may not always be online. In addition, these schemes require participants to remember multiple keys and to carry-out all the compute-intensive operations, which can be burdensome for computationally-limited devices such as IoT devices. To address this issue, asynchronous GKE schemes (e.g., [5], [6], [7]) have been proposed in the literature that allow some participants to be offline during the execution of the protocol. These offline participants can come online later at some point and derive the session key. However, such schemes still require participants to carry out all the computations. It is also worth noting that the existing synchronous and asynchronous GKE schemes rely on Trusted Third Parties (TTPs) such as Certificate Authorities (CAs) to allow the participants to authenticate each other and relay messages. Such reliance on TTPs is another significant drawback of the existing GKE schemes. For instance, if a TTP is not available during the execution of a GKE scheme, it can render availability problem for the underlying application and become a single point of failure. Furthermore, TTPs are vulnerable to attacks such as rogue certificate [8] and key compromise impersonation [9] that allow attackers to impersonate them.

In addition, a GKE scheme needs to support the following two important security requirements when it is used in

- *Victor Youdom Kemmoe is with the Department of Computer Science, Brown University, Providence, RI 02906 USA. E-mail: victor_youdom_kemmoe@brown.edu.*
- *Yongseok Kwon and Sunghyun Cho are with the Department of Computer Science and Engineering, Hanyang University, Seoul 133-791, Republic of Korea. E-mail: {totoey200, chopro}@hanyang.ac.kr.*
- *Rasheed Hussain is with the Department of Electrical and Electronic Engineering, University of Bristol, Bristol BS8 1TH, U.K. E-mail: rasheed.hussain@bristol.ac.uk.*
- *Junggab Son is with the Department of Computer Science, University of Nevada, Las Vegas, NV 89557 USA. E-mail: junggab.son@unlv.edu.*

Manuscript received 15 June 2021; revised 15 April 2022; accepted 6 June 2022. Date of publication 12 July 2022; date of current version 11 July 2023. This work was supported in part by the Ministry of Science, ICT (MSIT), Korea, through High-Potential Individuals Global Training Program under Grant 2021-0-01547-00 supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP), and in part by IITP, Korea Government (MSIT) under Grant 2021-0-00368, for the Development of the 6G Service Targeted AI/ML-Based Autonomous-Regulating Medium Access Control (6G STAR-MAC).
(Corresponding Author: Junggab Son.)
Digital Object Identifier no. 10.1109/TDSC.2022.3189977

a vulnerable environment where devices are more easily compromised than desktop computers, e.g., IoT environment [10]. The first requirement is the Perfect Forward Secrecy (PFS), i.e., the compromise of the long-term identity and key of a participant does not reveal the previously established session keys [11], and the second requirement is the Post-Compromise Security (PCS), i.e., participants can re-establish the security of a session even after one group member was compromised [12]. A failure to satisfy these additional requirements will result in creating critical vulnerabilities such as cloning attack [13]. Therefore, it is imperative to develop an efficient and robust GKE mechanism to fulfill these requirements.

Our Contributions. Our work aims to answer the following question: *is it possible to construct a secure asynchronous GKE scheme that does not rely on a TTP, and that can reduce the computational load of the participants?* To answer this question, in this paper, we present an asynchronous GKE scheme that leverages blockchain and smart contracts to store the security key-related materials and reduce participants' computational workload. In contrast to the traditional TTPs, blockchain provides a distributed architecture that easily allows to ensure the integrity of data stored on it and is resilient against attacks such as Denial of Service (DoS) [14], [15]. Furthermore, with the support of smart contracts, one can implement the logic of a TTP on a blockchain without the risk of attackers spoofing the smart contracts [16]. Our contributions can be summarized as follows:

- We propose an asynchronous GKE protocol that uses blockchain to store the security key-related material and uses smart contracts to reduce the number of operations that the participants must perform. The proposed protocol ensures PFS and PCS and allows the addition and removal of group members.
- We analyse the security of the proposed protocol and show that an attacker cannot obtain the session key from the key materials stored in the blockchain and that our protocol is secure under the standard attacker model.
- We present two implementations of our protocol based on Ethereum [17]. In the first implementation, all key-related materials are stored in the blockchain while for the second implementation, only the key-related materials necessary for the smart contracts are store on the blockchain with the rest kept in a distributed storage.

Organization of the Paper. In Section 2, we provide a technical overview of our solution. We define the important notations and introduce the readers to some necessary background in Section 3. The baseline system and adversarial models for based our proposed scheme are discussed in Section 4, and in Section 5, we give a detailed description of our proposed scheme. In Section 6, we perform a security analysis, and in Section 7, we describe the implementation of our proposed scheme. In Section 8, we present the related works followed by conclusions in Section 9.

2 Overview and Background

In this section, we provide a technical overview of our proposed smart contracts-based asynchronous GKE mechanism.

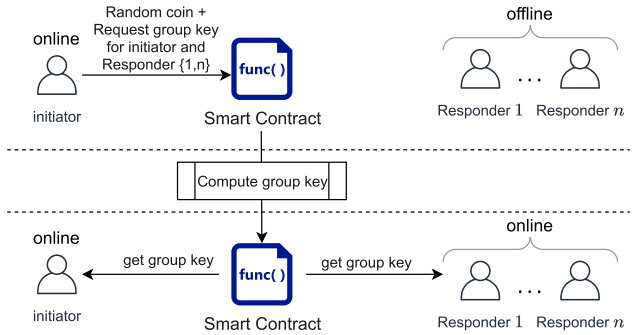


Fig. 1. Naïve Group Key Exchange (GKE) protocol.

Fig. 1 depicts a naïve approach in which an initiator sends a request to form a group with n responders and a random coin to a smart contract. The smart contract uses the random coin to compute the group key and then the initiator and the responders come online and request the group key from the smart contract. However, such a construction is fundamentally insecure. Since the instructions of a smart contract can be accessed by anyone, an attacker who obtains the random coin sent by the initiator will be able to reproduce the computations performed by the smart contract and obtain the group key. Also, one could simply read the state of the smart contract and get the computed group key. Furthermore, having the initiator being the only one providing the random coin does not satisfy the definition of GKE since all group members must contribute.

Using 2-Party Key Exchange. We need all the group members to contribute key materials to the GKE, but as we can see from Fig. 1, we consider the initiator to be the only one online during the first execution phase of the protocol, and since the responders are offline, they cannot contribute. To solve this issue, we borrow the idea from Cohn-Gordon *et al.* [7] where a 2-party key exchange (2KE) is used to allow the initiator to pre-compute the contributions of responders. In brief, following Fig. 2, before the execution of the protocol, group members come online and upload ephemeral keys to the smart contract. After that, the initiator comes online and requests the ephemeral keys uploaded by the responders. Once the initiator has those ephemeral keys, it executes an asynchronous 2KE between itself and each responder and then it considers the results as contributions from the responders. Next, after sampling its contribution, the initiator sends a request to compute the group key to the smart contract with its own and responders' contribution as input. The use of a 2KE prevents an attacker that accesses keys stored in the smart contract from computing the group key before the initiator makes its request. Furthermore, it allows the responders to check that their contribution was included by executing the 2KE with the same inputs as the initiator and use the results in the re-execution of the protocol.

Using Homomorphic Encryption. Simply using a 2KE does not prevent an attacker from *seeing* the group key that will be computed after the initiator makes its request. To solve this issue, we use a Homomorphic Encryption (HE) scheme. (An encryption scheme $\Gamma = (\Gamma.\mathsf{Enc}, \Gamma.\mathsf{Dec})$ is homomorphic if it accepts an operator $\circ$ such that for two messages $m_1, m_2 \in \mathcal{M}$, $\Gamma.\mathsf{Enc}(m_1) \circ \Gamma.\mathsf{Enc}(m_2) = \Gamma.\mathsf{Enc}(m_1 \circ m_2)$, where $\mathcal{M}$ is the message space of Γ). Using an HE scheme, the initiator

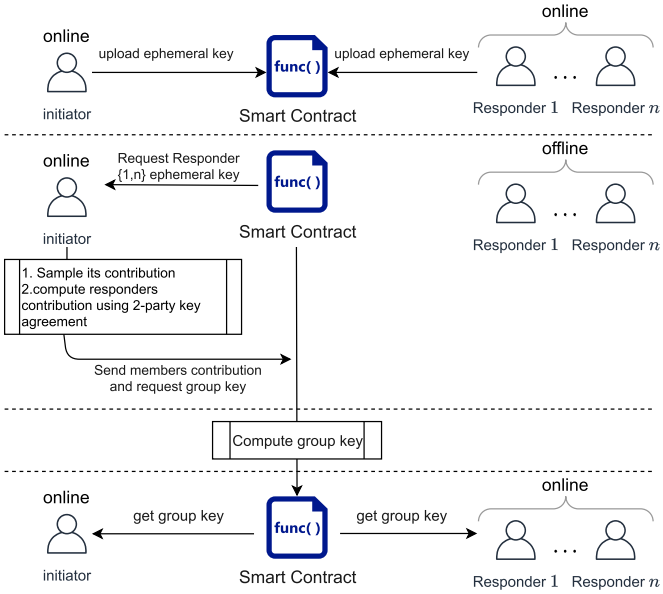


Fig. 2. Abstract protocol.

encrypts the group members' contribution before sending them to the smart contract. Then, the smart contract computes the group key using the encrypted contributions. However, because the computed group key is encrypted, the responders will need the decryption key. To solve this issue, the initiator derives a symmetric encryption key for each responder using the responders' contribution. Then, it uses those symmetric keys to encrypt the key that will allow the responders to *obtain* the group key. Hence, in addition to sending the encrypted contribution, the initiator also sends the encrypted key in the request sent to the smart contract.

Achieving PFS and PCS. Even though the previous approach, i.e., integrating 2KE and HE into smart contracts, seems to be effective, it cannot guarantee PFS and PCS. To satisfy PFS, we use ephemeral keys and secret nonces. The computation of the session key requires the use of ephemeral keys and secret nonces in addition to a long-term key. Once the computation is done, all ephemeral keys used in the computation are erased from participants' devices. Therefore, a comprise of long-term keys will not be enough for an adversary to get previously established session keys. To satisfy PCS, we devise a key update procedure that allows the participants to update their local state and the session key. If an adversary compromises the local state of a participant but allows that participant to successfully execute the key update procedure (by not interfering with the transmitted messages), then the participant's local state and the session key will be updated to new values, unknown to the adversary. Hence, the security of the group can be reestablished.

3 Preliminaries

In this section, we discuss the preliminaries that will be helpful in understanding our proposed mechanism.

3.1 Notations

For an elliptic curve E defined over a finite field $\mathbb{F}_p$ following the equation $E : y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_p$ and p is a large prime, we denote by $E(\mathbb{F}_p)$ an additive abelian group defined over E. We use $\Pr[X]$ to denote the probability that an event X takes place and $\Pr[X : Y]$ to denote the probability that event Y happens given that event X occurs. We use $x \xleftarrow{\$} \mathcal{X}$ to denote that x is sampled uniformly at random from a set $\mathcal{X}$. Any function $f : \mathbb{N} \rightarrow \mathbb{R}$ is defined as negligible if for all constant $c \in \mathbb{R}$, there exists an $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq n^{-c}$. We use $\mathsf{negl}(.)$ to denote an undefined negligible function. For a user u_i, we use sk_i to denote its long-term private key and pk_i its long-term public key. We use bolded notations ($\mathbf{A}$) to denote sets of specific elements.We make use of a Key Derivation Function (KDF) that will be interpreted as a random oracle (RO) [7].

3.2 Digital Signature

Definition 1. *A digital signature scheme is a triplet of probabilistic polynomial time algorithms* $\Phi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *with the following properties:*

- $\Phi.\mathsf{Gen}(1^\lambda) \rightarrow (sk, pk)$. *On the input of a security parameter* 1^λ, *the key generation algorithm* $\Phi.\mathsf{Gen}$ *outputs a private-public key pair* (sk, pk).
- $\Phi.\mathsf{Sign}(sk, m) \rightarrow \sigma$. *On the input of a private key* sk *and a message* m, *the signature algorithm* $\Phi.\mathsf{Sign}$ *outputs a signature* σ.
- $\Phi.\mathsf{Verify}(pk, m, \sigma) \rightarrow b$. *On the input of a public key* pk, *a message* m, *and a signature* σ, *the verification algorithm* $\Phi.\mathsf{Verify}$ *outputs a bit* $b \in \{0, 1\}$. *If* $b = 1$, *then* σ *was generated by applying* $\Phi.\mathsf{Sign}$ *on* m *with the secret key related to* pk.

A digital signature scheme Φ is $(t_{\mathsf{CMA}}, \epsilon_{\mathsf{CMA}})-$Chosen Message Attack (CMA) secure if the probability that an adversary $\mathcal{A}$ running in time t_{CMA} and having access to pk and $\Phi.\mathsf{Sign}$ outputs a message-signature pair (m, σ) such that $\Phi.\mathsf{Verify}(pk, m, \sigma) = 1$ is less than or equal to ϵ_{CMA}, i.e.,

$$\Pr[(m, \sigma) \leftarrow \mathcal{A}^{\Phi.\mathsf{Sign}}(1^\lambda, pk) : \Phi.\mathsf{Verify}(pk, m, \sigma) = 1] \leq \epsilon_{\mathsf{CMA}}.$$

3.3 Blockchain and Smart Contract

Blockchain. A blockchain, in essence, is a distributed data structure similar to a linked-list, made of blocks (or nodes) and in which a block has a field that contains the hash value of the block that precedes it, that field serves as a link. A blockchain is maintained by a set of independent users who decide (by using a consensus algorithm) what block to add to the blockchain. In general, beside the field serving as link, a block contains a set of transactions, and each transaction is signed (using a digital signature scheme) by its issuer. Blockchain has some salient features such as *persistence* (once a block is added to the blockchain, it cannot be removed), and *public verifiability* (anyone can verify the integrity of the blockchain) that makes it attractive to develop applications that do not rely on trusted third parties [18], [19].

Smart Contract. A Smart Contract (SC) is a program whose instructions are stored in a blockchain and are publicly accessible. To execute a functionality of a SC, one has to send a transaction that references the required functionality to the blockchain [17], [20]. Once in the blockchain, that transaction will be executed by the users who maintain the blockchain, and if the execution is valid, the transaction that triggered it, will be included in the blockchain. Furthermore, the overall

state of the blockchain will be updated. In addition, an SC can have an internal state or memory that also resides on the blockchain and is part of the blockchain's state. Relying on the blockchain means that anyone can *explore* old blocks to access previous internal states of an SC.

For the rest of the paper, we assume all transactions sent to a blockchain to be signed by their issuers using a CMA secure digital signature scheme, i.e., each transaction is of the form $\mathsf{tx} = \{payload, \sigma\}$.

3.4 Hardness Assumption

Given $E(\mathbb{F}_p)$ an elliptic curve group with base point $G \in E(\mathbb{F}_p)$ and prime order n, we consider a Probabilisitic Polynomial-Time (PPT) adversary $\mathcal{A}$.

Hardness 1 (Discrete Logarithm). *Let $(x, y) \leftarrow \mathsf{Gen}(1^\lambda, G)$ be a function that takes as inputs a security parameter 1^λ, the base point G and outputs $x \in \mathbb{Z}_n^*$ and $y = x.G \in E(\mathbb{F}_p)$. Given $1^\lambda, y$ and G as inputs, the probability that $\mathcal{A}$ outputs x' such that $x' = x$ should be negligible. More specifically*

$$\Pr[x' \leftarrow \mathcal{A}(1^\lambda, y, G) : x' = x] \leq \mathsf{negl}(\lambda).$$

Hardness 2 (Decisional Diffie-Hellman). *Given $a, b \in \mathbb{Z}_n^*$, let $m_0 \leftarrow (a.G, b.G, ab.G)$ and $m_1 \leftarrow (a.G, b.G, c.G)$, with $c \xleftarrow{\$} \mathbb{Z}_n^*$. The probability that $\mathcal{A}$ distinguishes m_0 from m_1 should be negligible, i.e.,*

$$|\Pr[1 \leftarrow \mathcal{A}(m_0)] - \Pr[1 \leftarrow \mathcal{A}(m_1)]| \leq \mathsf{negl}(\lambda),$$

where λ is the security parameter that was used to generate $E(\mathbb{F}_p)$.

3.5 ElGamal Encryption Over Elliptic Curve

Definition 2. *Following the construction of El Gamal [21], ElGamal Encryption scheme over Elliptic Curve is a triplet of probabilistic polynomial time algorithms $\mathsf{ElGamal} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec})$ with the following properties:*

- $\mathsf{ElGamal.Setup}(1^\lambda) \rightarrow (n, G, E(\mathbb{F}_p), pk, sk)$. *On the input of a security parameter 1^λ, the setup algorithm* ElGamal.Setup *generates a group $E(\mathbb{F}_p)$ with base point G and prime order n. Next, it generates a public-private key pair (pk, sk) such that $sk = \alpha \in \mathbb{Z}_n^*$ and $pk = \alpha.G \in E(\mathbb{F}_p)$. It outputs the tuple $(n, G, E(\mathbb{F}_p), pk, sk)$.*
- $\mathsf{ElGamal.Enc}(pk, m) \rightarrow ct$. *The encryption algorithm* ElGamal.Enc *takes as input a public key pk and a message $m \in E(\mathbb{F}_p)$. It samples at random a secret value $r \in \mathbb{Z}_n^*$ and computes $c_1 = r.G$. Finally, it computes $c_2 = m + (r.pk)$ and outputs a ciphertext $ct = (c_1, c_2)$.*
- $\mathsf{ElGamal.Dec}(sk, ct) \rightarrow m$. *On the input of a secret key sk and a ciphertext ct, first the decryption algorithm* ElGamal.Dec *computes $k = sk.c_1$. Then, it computes $m = c_2 - k$.*

ElGamal has a homomorphic property over addition. Given two ciphertexts ct and ct' encrypted using the same public key pk

$$\begin{aligned} ct + ct' &= (c_1 + c_1', c_2 + c_2') \\ &= ((r + r').G, M + M' + (r + r').pk). \end{aligned}$$

For the rest of the paper, instead of generating r during the execution of ElGamal.Enc, we generate r before its execution, and use the notation $\mathsf{ElGamal.Enc}_{(r,pk)}(m)$ to denote the encryption of the message m using the secret value r and the public key pk. Furthermore, during the execution of ElGamal.Enc, we make the use of a collision-resistant hash function $H' : \{0,1\}^* \rightarrow E(\mathbb{F}_p)$ to map binary strings to group elements in $E(\mathbb{F}_p)$.

3.6 Asynchronous Biparty Key Exchange

Definition 3. *An asynchronous biparty key exchange is a protocol π_{ABKE} between an initiator (online) and a responder (offline) that allows both to generate a common session key. It takes the secret key of the initiator sk_I, the public key of the responder pk_R, a secret ephemeral key of the initiator x_I, and a public ephemeral key of the responder y_R as input such that when reversing the roles, the following statement is verified*

$$\pi_{\mathsf{ABKE}}(sk_I, pk_R, x_I, y_R) = \pi_{\mathsf{ABKE}}(sk_R, pk_I, x_R, y_I) = \alpha.$$

For our proposed scheme, we assume π_{ABKE} to be a strong one-round authenticated key exchange protocol. Examples of such protocol can be found in the literature [22], [23].

4 System and Adversarial Models

In this section, we present our system and adversarial models based on the multi-stage key exchange model of Fishclin *et al.* [24] and the group key exchange model of Cohn-Gordon *et al.* [7].

4.1 System Model

System Participants. Let $\mathcal{U}$ denotes the set of all users that participate in the system. We use the notation $u \in \mathcal{U}$ to denote a participant in the set of all participants and $u_i \in \mathcal{U}$ to denote a participant at position $i \in \mathbb{N}$ in a group. After creating an account on the blockchain, a user u obtains a long-term private-public key pair (sk, pk).

Smart contract. Participants interact with smart contracts (SCs) that are stateful to manage groups. The states and set of instructions of SCs can be accessed by anyone. For simplicity, we assume that an SC can only be used to manage one group.

Miners. The blockchain on which SCs are deployed is maintained by a set of independent *miners*/computers. We assume that miners are honest, i.e., they process transactions in the order they received them. *Sessions.* Users can run multiple instances of the proposed scheme at the same time with different group members. We use the term *session* to denote an instance between two or more users. During a session's lifetime, its state can be updated based on the messages issued by the group members. We refer to those points of update as *stages*. We use Π_u^t to denote the tth session of user u and ${}^d\Pi_u^t$ to denotes the tth session at stage d of user u, with $t, d \in \mathbb{N}$.

For each session Π_u^t, its state contains the following information:

- sk, the long-term key of u, instance's owner.
- $(\mathbf{X}, \mathbf{Y})$, a pair of sets. $\mathbf{X}$ is the set of non-used secret ephemeral keys, and $\mathbf{Y}$ is the set of non-used public ephemeral keys.

- blkNum $\in \mathbb{N}$, the block number that references the last block containing the SC's state that u read.
- SCID, the identifier of the smart contract used to handle the group.
- status $\in \{running, accept, reject\}$, the execution status of the instance. If status $= accept$, then the session key was successfully derived.
- role $\in \{initiator, responder\}$, the role of u in the session. If role $= initiator$, then u is the user that initiated the group's creation and is assumed to be the group administrator. Else, if role $= responder$, then u is a user that joined a group.
- idx $\in \mathbb{N}$, the index/position of u in the group.
- $h \in E(\mathbb{F}_p)$, a hash of u's contribution computed using H'.
- $k \in \mathbb{Z}_n^*$, a secret value used to obtain the *pre*-group key computed by the smart contract.
- Mbrs, the list of group members.
- ssk $\in \{\perp\} \cup \{0,1\}^*$, the group (session) key to be used.
- $d \in \mathbb{N}$, the stage of the session.

4.2 Adversarial Model

In this subsection, we define the security goals we aim to achieve and the capabilities of an adversary.

Key Independence. The knowledge of a session key must not endanger the security of another session key. Specifically, for any instance Π_u^t, each session key ssk established during any stage ${}^d\Pi_u^t$ must be indistinguishable from random.

Perfect Forward Secrecy (PFS)**.** Session keys that were established prior to the corruption of a session's state must remain secure. More precisely, for all $t \in \mathbb{N}$, if for $d_0 \in \mathbb{N}$, ${}^{d_0}\Pi_u^t$ is corrupted, then the session key in ${}^d\Pi_u^t$, for $0 \leq d < d_0$, must remain secure.

Post Compromise Security (PCS)**.** If a session's state of a user u has been compromised, members in that session should regain a security guarantee if they successfully execute the key update process. Specifically, if an attacker successfully compromises the state of a session ${}^{d_0}\Pi_u^t$ but allows ${}^{d_0+i}\Pi_u^t$ to *accept* after a key update procedure, then the session key in ${}^d\Pi_u^t$ is secure, where $d \geq d_0 + i$.

Adversary Queries. We consider a PPT adversary $\mathcal{A}$ that completely controls the network with access to the following queries:

- NewSession($\{u_0, u_1, \ldots, u_m\}$, admin, SCID): It executes the *Group Creation* process between unused instances of users in the set $\{u_0, u_1, \ldots, u_m\}$, where the administrator is chosen through the admin variable and the Group Key SC (GSKC) to be used through the SCID variable. Each input of NewSession is selected by the adversary, and at the end of its execution, its output trace is sent to the adversary.
- Send(Π_u^t, δ): It sends a message δ to an instance Π_u^t. If δ is well formed, u processes it according to the protocol. Otherwise, u rejects δ. If δ leads Π_u^t to $accept$ after processing, u updates the state of Π_u^t accordingly. This query simulates the possibility for the adversary to tamper messages sent over the network or to mimic the blockchain.
- Reveal(${}^d\Pi_u^t$): If ${}^d\Pi_u^t$.ssk $\neq \perp$, it returns ${}^d\Pi_u^t$.ssk to the adversary. This query characterizes a leakage of the session key of a stage d to the adversary.
- Corrupt(${}^d\Pi_u^t$) : This query simulates a total corruption of the session state. It returns all the values stored in ${}^d\Pi_u^t$'s state except the session key.
- Test(${}^d\Pi_u^t$): If ${}^d\Pi_u^t$.status $= accept$, a bit $b \xleftarrow{\$} \{0,1\}$ is selected, and if $b = 1$, ${}^d\Pi_u^t$.ssk is returned, else if $b = 0$, a string of bits selected uniformly at random is returned.

5 Asynchronous Group Key Exchange

In this section, we provide a detailed description of our proposed scheme. First, we give a description of the SC and functionalities provided to users that are used by our scheme, and second, we give a description of the main operations of our proposed GKE scheme.

5.1 Components of the Proposed Scheme

Group Key SC. This is the SC used to manage the group. A complete description of the GKSC is provided in Fig. 3. First, it has an internal state St that holds the following elements:

- grpOP $\in \{\perp\} \cup \{$CRT, UPD, ADD, RM$\}$ is a variable that keeps track of which operation was performed on the group. CRT refers to the creation of a group, UPD is the update of the group key, ADD is the addition of a new member, and RM refers to the removal of a member.
- eKeysMap : $E(\mathbb{F}_p) \rightarrow [E(\mathbb{F}_p)]^\ell$ is a map that links a public key pk with a set of ephemeral public keys $\mathbf{Y}$, where $|\mathbf{Y}| = \ell$.
- admin $\in \{\perp\} \cup E(\mathbb{F}_p)$ is a variable that holds the public key of the group administrator.
- **gKey** is a set with cardinality $|\mathbf{gKey}| = 2$. It holds the group *pre*-key.
- **encK** is a set that holds encrypted secret value k for each group member.
- **encS** is a set that holds encrypted secret value s for each group member.
- **usedEKeys** is a set that holds the ephemeral public keys that were used during an operation.

Second, it consists of five public functions PostEKeys, CreateGrp, UpdGKey, AddMbr, and GetEKey that modify the contract state St and listen to the transactions ${}^*tx_{PostEKeys}$, ${}^*tx_{CreateGrp}$, ${}^*tx_{UpdGkey}$, ${}^*tx_{AddMbr}$, ${}^*tx_{GetEKey}$, respectively, and two public functions GetGKey, GetGKeyMat that read the contract's state stored at the most current block. Following is the further description of each function:

- PostEKeys allows a user to store a set of ephemeral public keys in St.eKeysMap. It is activated by a transaction ${}^*tx_{PostEKeys}$.
- CreateGrp allows a user (the group administrator) to *create a group* or *remove a group member*. More specifically, if St.grpOP $=$ CRT, CreateGrp computes and stores the different key materials that will be used to derive the initial group key. Else, if St.grpOP $=$ RM, CreateGrp updates different key materials to reflect the removal of members. It is activated by a transaction ${}^*tx_{CreateGrp}$.

INTERNAL STATE St :
- Variables:
  - grpOP $\in \{\perp\} \cup \{\mathsf{CRT}, \mathsf{UPD}, \mathsf{ADD}, \mathsf{RM}\}$
  - eKeysMap : $E(\mathbb{F}_q) \rightarrow [E(\mathbb{F}_q)]^\ell$
  - admin $\in \{\perp\} \cup E(\mathbb{F}_q)$
  - $\mathbf{gKey}$
  - $\mathbf{encK}, \mathbf{encS}, \mathbf{usedEKeys}$
- Events: shareGrpOP(_grpOP)

PostEKeys() :
- INPUTS:
  - Sender public key pk_s
  - Set of ephemeral public keys $\mathbf{Y}_s$

1) assert(msg.sender.pubkey $= pk_s$)
2) Set eKeysMap$[pk_s] \leftarrow \mathbf{Y}_s$

GetEKey() :
- INPUTS:
  - Target public key pk_t
- OUTPUTS: An unused ephemeral key y_{i_t} of target

1) Let $\mathbf{Y}_t \leftarrow$ eKeysMap$[pk_t]$
2) Let i equals index of unused key in $\mathbf{Y}_t$
3) Let $y_{i_t} \leftarrow \mathbf{Y}_t[i]$
4) Mark $\mathbf{Y}_t[i]$ as used
5) Output y_{i_t}

CreateGrp() :
- INPUTS:
  - Administrator public key pk_a
  - _grpOP
  - Set of secret values $\mathbf{\_encK}$
  - Set of secret values $\mathbf{\_encS}$
  - Set of members contributions $\mathbf{A}$
  - Set of used ephemeral keys $\mathbf{\_usedEKeys}$

1) assert(
   - msg.sender.pubkey $= pk_a$
   - $|\mathbf{A}| = |\mathbf{\_usedEKeys}|$
   - _grpOP $=$ CRT or _grpOP $=$ RM)
2) Let $m \leftarrow |\mathbf{A}|$
3) grpOP $\leftarrow$ _grpOP
4) If grpOP $=$ RM:
   - Check that the group exist and admin $= pk_a$

   Else: admin $\leftarrow pk_a$
5) $\mathbf{encK} \leftarrow \mathbf{\_encK}$, $\mathbf{encS} \leftarrow \mathbf{\_encS}$
6) $\mathbf{usedEKeys} \leftarrow \mathbf{\_usedEKeys}$
7) $\mathbf{gKey}[0] \leftarrow \sum_{i=1}^{m} \mathbf{A}[i]$
8) $\mathbf{gKey}[1] \leftarrow \sum_{i=1}^{m} \mathbf{usedEKeys}[i]$
9) Emmit shareGrpOP(grpOP)

GetGKey() :
- INPUTS:
  - Sender public key pk_s
- OUTPUTS: the variable $\mathbf{gKey}$

1) CheckMembership(pk_s)
2) Output $\mathbf{gKey}$

GetGKeyMat() :
- INPUTS:
  - Sender public key pk_s
- OUTPUTS: tp, a tuple of key materials

1) CheckMembership(pk_s)
2) If grpOP $\in \{\mathsf{CRT}, \mathsf{ADD}, \mathsf{RM}\}$:
   - tp $\leftarrow (\mathbf{gKey}, \mathbf{encK}[pk_s], \mathbf{encS}[pk_s], \mathbf{usedEkeys}[pk_s])$

   Else If grpOP $=$ UPD:
   - tp $\leftarrow (\mathbf{gKey}, \mathbf{encK}[pk_s], \mathbf{usedEkeys}[pk_s])$
3) Output tp

UpdGKey() :
- INPUTS:
  - Sender public key pk_s
  - Set of secret values $\mathbf{\_encK}$
  - Updating factor λ
  - Sum of used ephemeral keys _sumEKeys
  - Set of used ephemeral keys $\mathbf{\_usedEKeys}$

1) CheckMembership(pk_s)
2) $\mathbf{encK} \leftarrow \mathbf{\_encK}$
3) $\mathbf{usedEKeys} \leftarrow \mathbf{\_usedEKeys}$
4) $\mathbf{gKey}[0] \leftarrow \mathbf{gKey}[0] + \lambda$
5) $\mathbf{gKey}[1] \leftarrow$ _sumEKeys
6) grpOP $\leftarrow$ UPD
7) Emmit shareGrpOP(grpOP)

AddMbr() :
- INPUTS
  - New Member public key pk_n
  - New member's encK$_n$
  - New member contribution θ
  - Set of secret values $\mathbf{\_encS}$
  - Set of used ephemeral keys $\mathbf{\_usedEKeys}$

1) assert(admin $=$ msg.sender.pubkey)
2) $\mathbf{usedEKeys} \leftarrow \mathbf{\_usedEKeys}$
3) Add encK$_n$ to $\mathbf{encK}$
4) $\mathbf{encS} \leftarrow \mathbf{\_encS}$
5) $\mathbf{gKey}[0] \leftarrow \mathbf{gKey}[0] + \theta$
6) $\mathbf{gKey}[1] \leftarrow \mathbf{gKey}[1] + \mathbf{\_usedEKeys}[pk_n]$
7) grpOP $\leftarrow$ ADD
8) Emmit shareGrpOP(grpOP)

PRIVATE FUNCTIONS

CheckMembership() :
- INPUTS
  - Sender public key pk_s

1) assert(msg.sender.pubkey $= pk_s$)
2) Check that the group exist
3) Check that pk_s is a member of the group

Fig. 3. Full group key smart contract (GKSC).

- UpdGKey allows a group member to update the values of sets St.**encK** and St.**usedEKeys** and the group *pre*-key St.**gKey**. It is activated by a transaction ${}^{*}tx_{UpdGKey}$.
- AddMbr enables a group administrator to add new group member. More specifically, it allows the group administrator to update the key materials stored in St such that all members including the new one, can

obtain the new group key. It is activated by a transaction ${}^{*}tx_{AddMbr}$.

- GetEKey reads St at the most current block and returns an unused ephemeral public key of a target t stored in the set $\mathbf{Y}_t \leftarrow \mathsf{St.eKeysMap}[pk_t]$, with pk_t being the public key of t. It is activated by a transaction ${}^{*}tx_{GetEKey}$.
- GetGKey reads St at the most current block and returns the *pre*-group key **gKey**.
- GetGKeyMat reads St at the most current block and returns different key materials that are chosen based on the value of St.grpOP. Those key materials will be used to obtain the final group key.

Futhermore, GKSC has a *private/helper* function named CheckMembership() whose aim is to check if a user is part of the group managed by GKSC. Also, in Fig. 3, we used the following notations derived from Solidity [25]:

- msg.sender.pubkey allows to retrieve the public key of the user who sent a transaction.
- Events are *actions* that are logged in the blockchain. They are activated by the keywork *Emmit*. In Fig. 3, we have an event called shareGrpOP() which allows us to log the operations that were performed on the group.

User's Algorithms. To interact with the blockchain and the group key SC, users have access to the following algorithms:

- $\mathsf{CreateAccount}(1^\lambda) \rightarrow (sk, pk)$. CreateAccount takes as input a security parameter 1^λ and returns a secret key $sk \in \mathbb{Z}_n$ and public key $pk = sk.G \in E(\mathbb{F}_p)$. Furthermore, it creates an account on the blockchain hosting the SC such that pk can be used to reference that account.
- $\mathsf{GenEkeys}(1^\lambda, \ell) \rightarrow (\mathbf{Y}, \mathbf{X})$. GenEkeys takes as input a security parameter 1^λ and a variable $\ell \in \mathbb{N}$ and returns two sets $\mathbf{Y}, \mathbf{X}$ such that $|\mathbf{X}| = |\mathbf{Y}| = \ell$. $\mathbf{y}$ represents the set of ephemeral public keys, and $\mathbf{X}$ the set of ephemeral secret keys such that $x_i \in \mathbf{X}$ corresponds to the secret key of the public key $y_i \in \mathbf{Y}$.
- $\mathsf{UploadEKeys}(pk, \mathbf{Y}) \rightarrow {}^{*}tx_{PostEKeys}$. UploadEKeys allows a user to upload a set of fresh ephemeral public keys to GKSC. It takes as input the public key pk of that user, and a set $\mathbf{Y}$ containing the ephemeral public keys. Then, it outputs a transaction ${}^{*}tx_{PostEKeys}$.
- $\mathsf{reqEKey}(pk_t) \rightarrow y_{i_t}$. reqEKey allows a user to obtain a fresh ephemeral public key of a target t stored in GKSC's state St. It takes as input the target's public key pk_t and outputs one of its fresh ephemeral public key y_{i_t}. It generates a transaction ${}^{*}tx_{GetEKey}$ that calls GKSC's function GetEKey() which returns y_{i_t}.
- $\mathsf{reqGKey}(pk) \rightarrow \mathbf{gKey}$. reqGKey allows group members to obtain the set **gKey** stored in GKSC's state St. It takes as input pk, the public key of a group member, and generates a call to GKSC's function GetGKey().
- $\mathsf{reqGKMat}(pk) \rightarrow \mathsf{tp}$. reqGKMat allows group members to obtain different key materials that will allow them to derive the final group key. It takes as input a user's public key pk and outputs tp, a tuple of key materials. It generates a call to GKSC's function GetGKeyMat(), and if $\mathsf{St.grpOP} \in \{\mathsf{CRT}, \mathsf{ADD}, \mathsf{RM}\}$, then $\mathsf{tp} = (\mathbf{gKey}, \mathbf{encK}[pk_s], \mathbf{encS}[pk_s], \mathbf{usedEkeys}[pk_s])$. Otherwise, if $\mathsf{St.grpOP} = \mathsf{UPD}$, then $\mathsf{tp} = (\mathbf{gKey}, \mathbf{encK}[pk_s], \mathbf{usedEkeys}[pk_s])$.
- $\mathsf{initGrp}(pk_{adm}, \mathsf{grpOP}, \mathbf{encK}, \mathbf{encS}, \mathbf{A}, \mathbf{usedEKeys}) \rightarrow {}^{*}tx_{CreateGrp}$. initGrp allows a user (the administrator) to initiate the creation of a group or the removal of members from the group. It takes as input the administrator's public key pk_{adm}, the group status grpOP(CRT for the creation of the group, RM for the removal of members), two sets $\mathbf{encK}, \mathbf{encS}$ containing the encryption of the secret values k and s, respectively, for each member, a set $\mathbf{A}$ containing the contribution of group members, and a set **usedEKeys** of ephemeral public keys used by the process. It outputs a transaction ${}^{*}tx_{CreateGrp}$.
- $\mathsf{initGrpUpd}(pk_s, \mathbf{encK}, \lambda, \mathsf{sumEKeys}, \mathbf{usedEKeys}) \rightarrow {}^{*}tx_{UpdGkey}$. initGrpUpd allows a group member to initiate an update for the group key. It takes as input the group member public key pk_s, a set **encK** containing the encryption of the secret value k for each member, a value λ that will be used to update the group key, a value sumEKeys that represents the sum of ephemeral keys used, and a set **usedEKeys** that contains the ephemeral keys used. It outputs a transaction ${}^{*}tx_{UpdGkey}$.
- $\mathsf{initAddMbr}(pk_n, \mathsf{encK}_n, \theta, \mathbf{encS}, \mathbf{usedEKeys}) \rightarrow {}^{*}tx_{AddMbr}$. initAddMbr allows a group administrator to add a new member. It takes as input the new member's public key pk_n, the encryption of the secret value k for the new member encK_n, the contribution of the new member θ, two set **encS**, **usedEKeys** containing the encryption of the secret value s and the ephemeral public keys used, respectively. It outputs a transaction ${}^{*}tx_{AddMbr}$.

5.2 Description of Operations

Setup. The setup process performs the following steps:

1) It calls $\mathsf{GroupGen}(1^\nu) \rightarrow (n, G, a, b, E(\mathbb{F}_p))$. GroupGen generates the public parameters for the elliptic curve group that will be used by the system. It takes as input a security parameter 1^ν and outputs a group of points $E(\mathbb{F}_p)$ defined over $E/\mathbb{F}_p$ with coefficients $a, b \in \mathbb{F}_p$, and a point $G \in E(\mathbb{F}_p)$, the base point, with primer order n.
2) It initializes the blockchain and generates the genesis block b_0. In addition, it selects a Chosen-Ciphertext Attack (CCA)-secure cipher $F : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{C}$.
3) It deploys GKSC (Fig. 3), and initializes its internal state St. Furthermore, it publishes the parameters $n, G, a, b, E(\mathbb{F}_p)$, and F.

Once the setup process is completed, each user u_i executes $\mathsf{CreateAccount}(1^\lambda)$. The setup process can be executed once or multiple times depending on the needs of the runtime environment.

Ephemeral Key Uploading. The use of ephemeral keys is crucial to ensure that our proposed scheme achieves PFS and PCS. Hence, each participant should periodically refresh its ephemeral keys stored in GKSC. Moreover, before any execution of the proposed scheme, all group members should have fresh ephemeral keys stored in GKSC. Following is the procedure to generate and update the ephemeral keys:

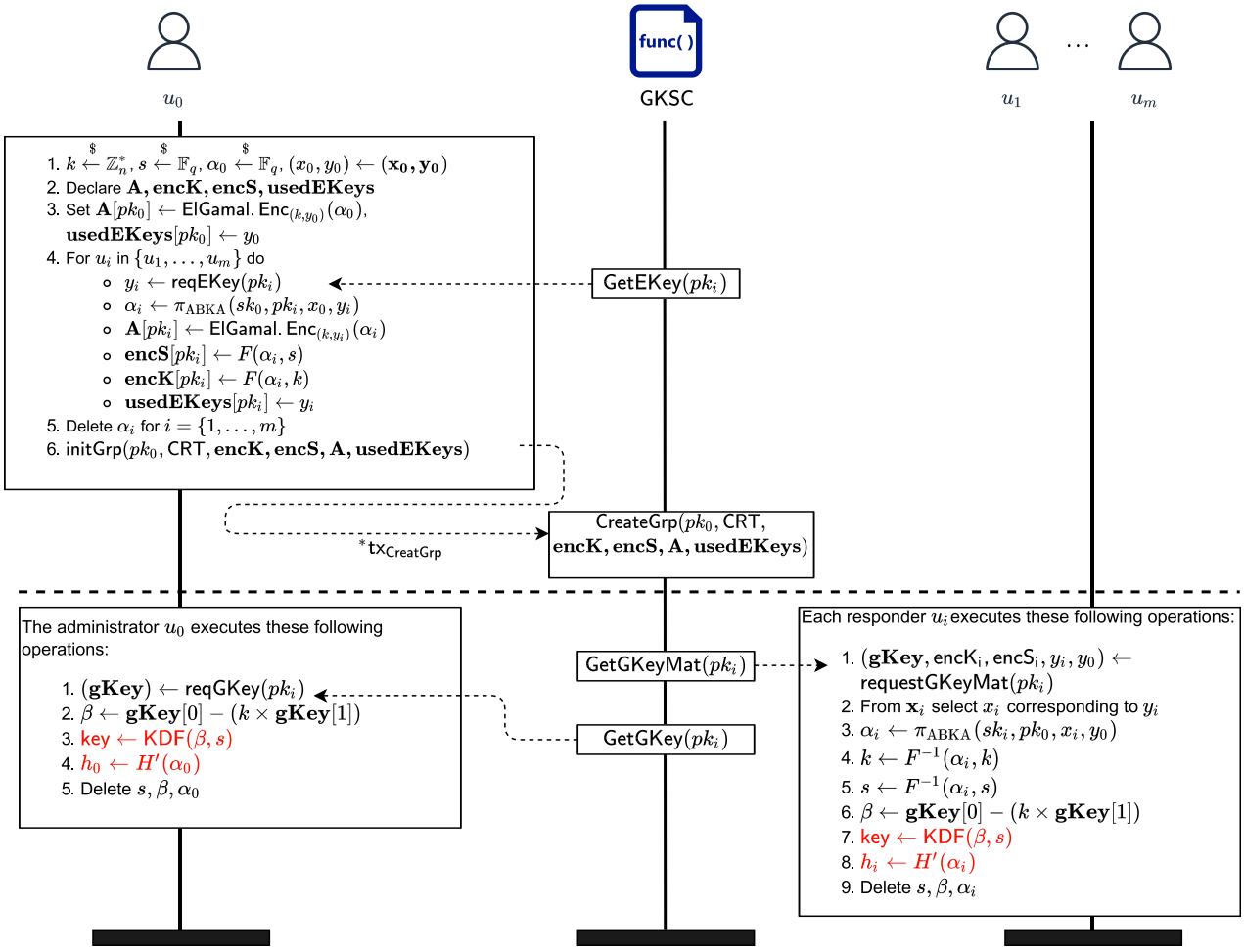


Fig. 4. Group creation process.

1) A user u_i executes $\mathsf{GenEkeys}(1^\nu, \ell) \rightarrow (\mathbf{Y}_i, \mathbf{X}_i)$.
2) u_i securely stores $\mathbf{X}_i$ and executes $\mathsf{UploadEKeys}(pk_i, \mathbf{Y}_i)$.
3) Once GKSC receives $^*tx_{PostEKeys}$, the function PostEKeys is executed. This allows to store $\mathbf{Y}_i$ in St.

If u_i re-executes that procedure, its ephemeral keys will be updated. It is worth noting that an ephmeral key pair must be discarded after it is used.

Group Creation. The group creation process allows an initiator/group administrator u_0 and a set of responders $\{u_1, \ldots, u_m\}$ to initialize the overall state of the protocol and establish the group key. Fig. 4 provides a graphical representation of this operation. When u_0 and $\{u_1, \ldots, u_m\}$ wish to create a group, they proceed as follows:

- *Phase 1.* (we assume that the initiator is the only user online) The initiator u_0 executes the following operations:
  1) Sample at random two secret values $k \in \mathbb{Z}_n^*, s \in \mathbb{F}_p$, and its contribution $\alpha_0 \in \mathbb{F}_p$. Then, select a fresh pair of ephemeral keys $(x_0, y_0) \leftarrow (\mathbf{X}_0, \mathbf{Y}_0)$.
  2) Declare four empty sets **A**, **encK**, **encS**, **usedE** *Keys*. Then, set $\mathbf{A}[pk_0] \leftarrow \mathsf{ElGamal.Enc}_{(k,y_0)}(\alpha_0)$ and $\mathbf{usedEKeys}[pk_0] \leftarrow y_0$.
  3) For each responder $u_i \in \{u_1, \ldots, u_m\}$:
     a) Get $y_i \leftarrow \mathsf{reqEKey}(pk_i)$, and then compute $\alpha_i \leftarrow \pi_{\mathsf{ABKE}}(s_0, pk_i, x_0, y_i)$, the contribution of u_i.
     b) Initialize $\mathbf{A}[pk_i] \leftarrow \mathsf{ElGamal.Enc}_{(k,y_i)}(\alpha_i)$, $\mathbf{encK}[pk_i] \leftarrow F(\alpha_i, k)$, $\mathbf{encS}[pk_i] \leftarrow F(\alpha_i, s)$, and $\mathbf{usedEKeys}[pk_i] \leftarrow y_i$.
     c) Discard α_i.
  4) Execute the function

$$\mathsf{initGrp}(pk_0, \mathsf{CRT}, \mathbf{encK}, \mathbf{encS}, \mathbf{A}, \mathbf{usedEKeys}).$$

- *Phase 2.* Each responder $u_i \in \{u_1, \ldots, u_m\}$ comes online and executes the following operations to obtain the group key:
  1) Get $(\mathbf{gKey}, \mathsf{encK}_i, encS_i, y_i, y_0) \leftarrow \mathsf{reqGKMat}(pk_i)$.
  2) Select $x_i \in \mathbf{X}_i$ the secret ephemeral key corresponding to y_i and compute $\alpha_i \leftarrow \pi_{\mathsf{ABKE}}(s_i, pk_0, x_i, y_0)$, its contribution that was used by u_0.
  3) Compute $k \leftarrow F^{-1}(\alpha_i, \mathsf{encK}_i)$ and $s \leftarrow F^{-1}(\alpha_i, \mathsf{encS}_i)$ and then compute $\beta \leftarrow \mathbf{gKey}[0] - (k \times \mathbf{gKey}[1])$.
  4) Compute the group key $\mathsf{ssk} \leftarrow \mathsf{KDF}(\beta, s)$ and $h_i \leftarrow H'(\alpha_i)$ (h_i *will be used to update the group key*).
  5) Discard the tuple of key materials (s, β, α_i).

In case of u_0, to obtain the group key, it performs the following operations:

1) Get $\mathbf{gKey} \leftarrow \mathsf{reqGKey}(pk_0)$ and then compute $\beta \leftarrow \mathbf{gKey}[0] - (k \times \mathbf{gKey}[1])$ (u_0 *has* s and k *in memory*).
2) Compute the group key $\mathsf{ssk} \leftarrow \mathsf{KDF}(\beta, s)$ and $h_0 \leftarrow H'(\alpha_0)$.

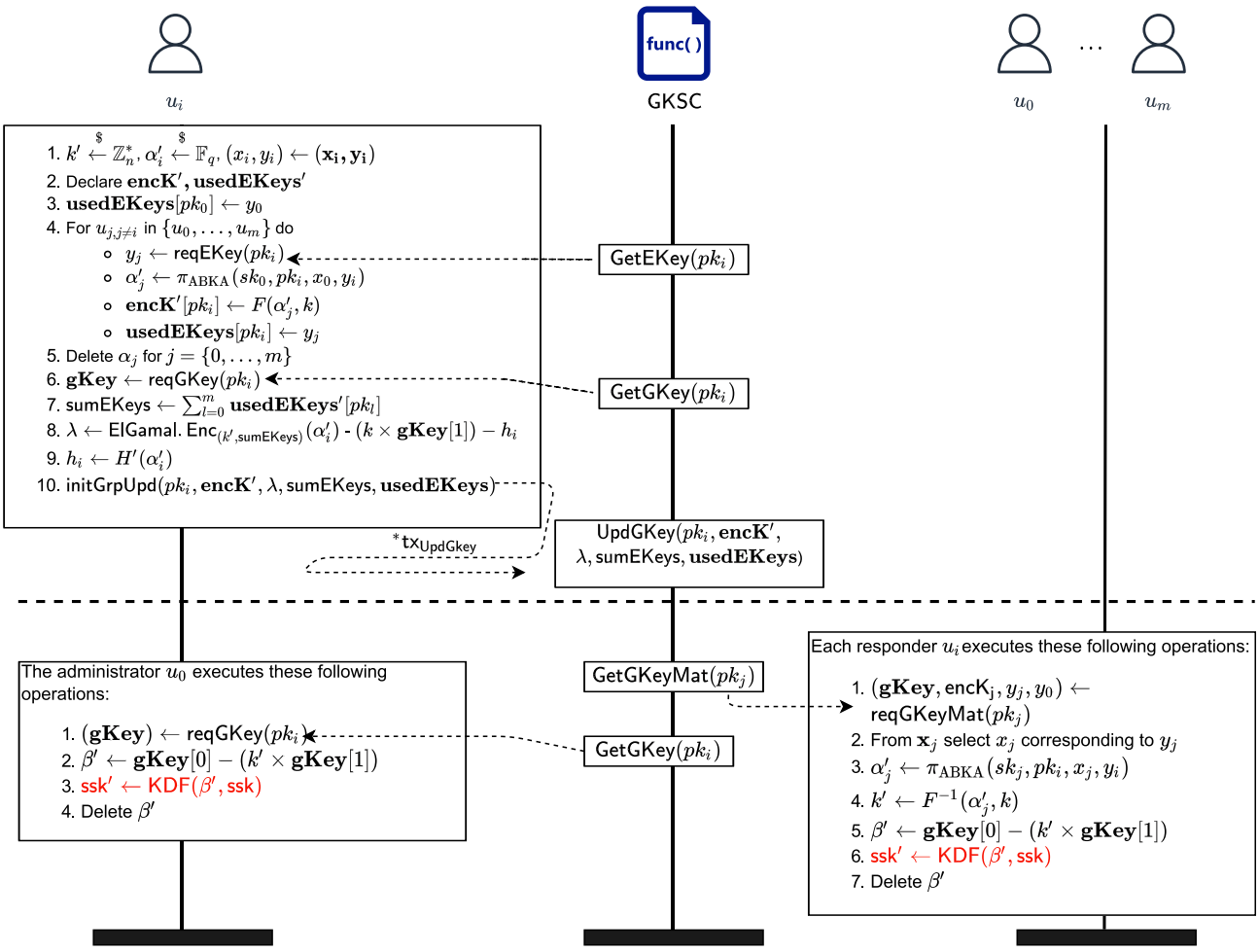


Fig. 5. Group update process.

3) Discard the tuple of key materials (s, β, α_0).

We show that all group members derive the same group key ssk $\leftarrow$ KDF(s, β).

Update Group. Using the update process, a group member (whose local state was possibly leaked) can update its local state and the group state, which will re-establishing the security of the group key. Fig. 5 provides a graphical representation of this operation. Following is a description of the update group process:

- *Phase 1*. A group member $u_i \in \{u_0, \ldots, u_m\}$ that wants to initiate the process, executes the following operations:
  1) Sample at random a secret value $k' \in \mathbb{Z}_n^*$ and a new contribution $\alpha_i' \in \mathbb{F}_p$. Then, select a fresh pair of ephemeral keys $(x_i, y_i) \leftarrow (\mathbf{X}_i, \mathbf{Y}_i)$.
  2) Declare two empty sets $\mathbf{encK'}, \mathbf{usedEKeys'}$ and initialize $\mathbf{usedEKeys'}[pk_i]$ with y_i.
  3) For each user $u_{j,j\neq i} \in \{u_0, \ldots, u_m\}$:
     a) Get $y_j \leftarrow$ reqEKey(pk_j) and then compute $\alpha_j' \leftarrow \pi_{\text{ABKE}}(s_i, pk_j, x_i, y_j)$.
     b) Initialize $\mathbf{encK'}[pk_j] \leftarrow F(\alpha_j', k')$ and $\mathbf{usedE}$ $Keys'[pk_j] \leftarrow y_j$.
     c) Delete α_j'.
  4) Get $\mathbf{gKey} \leftarrow$ reqGKey(pk_i).
  5) Compute sumEKeys $\leftarrow \sum_{l=0}^{m} \mathbf{usedEKeys'}[pk_l]$ and the updating factor:

$$\lambda \leftarrow \text{ElGamal.Enc}_{(k', \text{sumEKeys})}(\alpha_i') - (k \times \mathbf{gKey}[1]) - h_i,$$

     where h_i is the hash of its contribution that was computed during *Phase 2* of the *Group Creation* process.
  6) Compute $h_i' \leftarrow H'(\alpha_i')$, the update of h_i.
  7) Execute

$$\text{initGrpUpd}(pk_i, \mathbf{encK'}, \lambda, \text{sumEKeys}, \mathbf{usedEKeys}).$$

- *Phase 2*. To get the new group key, each group member $u_{j,j\neq i} \in \{u_0, \ldots, u_m\}$ comes online and executes the following operations:
  1) Get $(\mathbf{gKey}, \text{encK}_j, y_j, y_i) \leftarrow$ reqGKMat(pk_j).
  2) Select $x_j \in \mathbf{X}_j$ the secret ephemeral key corresponding to y_j and compute $\alpha_j' \leftarrow \pi_{\text{ABKE}}(s_j, pk_i, x_j, y_i)$, its contribution that was used by u_i.
  3) Compute $k' \leftarrow F^{-1}(\alpha_j', \text{encK}_j)$ and then compute $\beta' \leftarrow \mathbf{gKey}[0] - (k' \times \mathbf{gKey}[1])$.
  4) Compute the new group key ssk$' \leftarrow$ KDF$(\beta',$ ssk$)$ then delete β'.

The initiator of the update group process u_i obtains the new group key as follows:

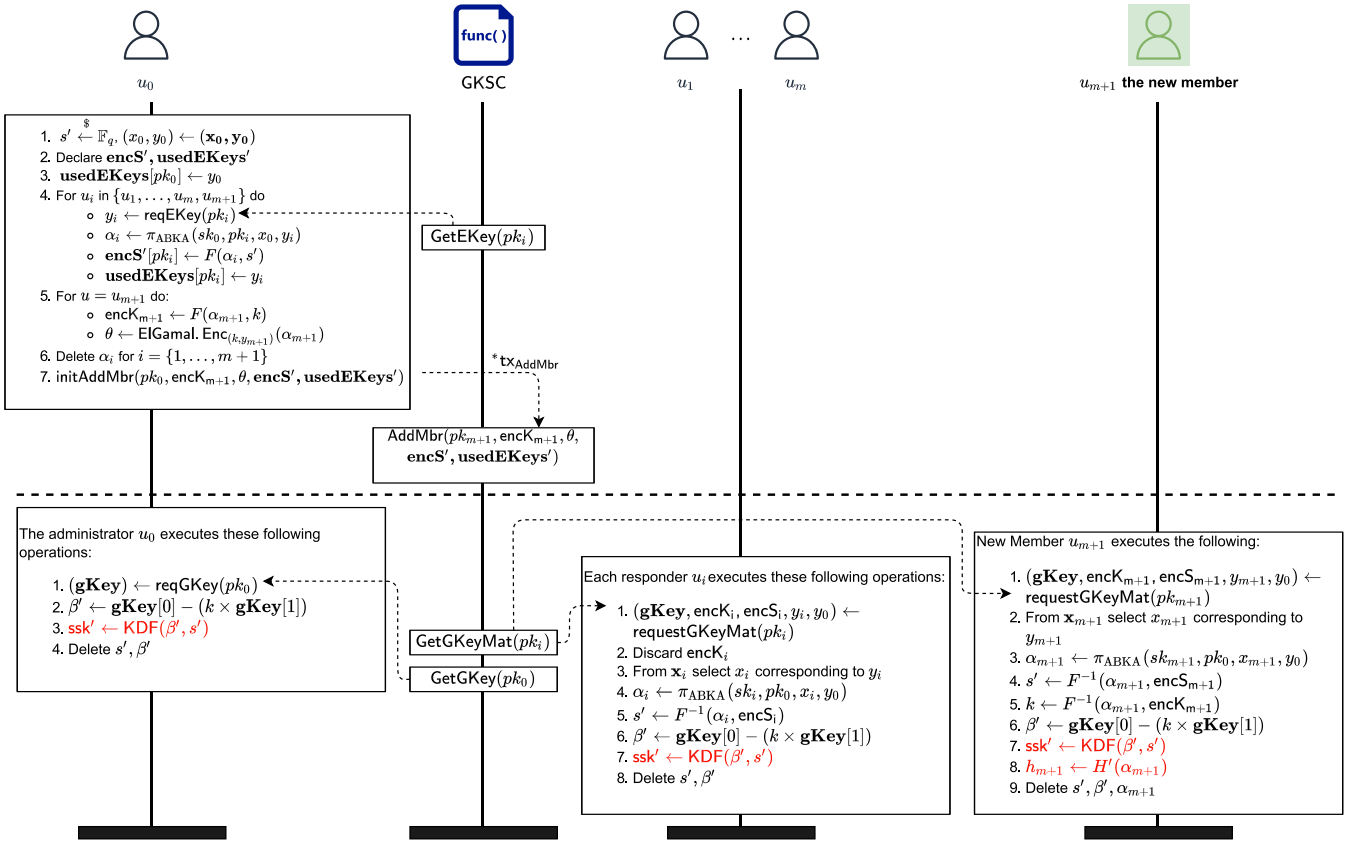


Fig. 6. Add member process.

1) Get the *pre*-group key $\mathbf{gKey} \leftarrow \text{reqGKey}(pk_i)$ and then compute $\beta' \leftarrow \mathbf{gKey}[0] - (k' \times \mathbf{gKey}[1])$, ($u_i$ *has* k' in memory).
2) Compute the new group key $\text{ssk}' \leftarrow \text{KDF}(\beta', \text{ssk})$ and then delete β'.

Add Member. A group administrator u_0 can add a new member u_{m+1} to its group $\{u_0, \ldots, u_m\}$. The addition of a new member will cause an update of the current group key, which will prevent the new group member from accessing messages exchanged prior to its addition. Fig. 6 provides a graphical representation of this operation. Following is a description of process to add new group member:

- *Phase 1.* The group administrator u_0 executes the following operations:
  1) Sample at random a secret value $s' \in \mathbb{F}_p$ and select a fresh pair of ephemeral keys $(x_0, y_0) \leftarrow (\mathbf{X}_0, \mathbf{Y}_0)$.
  2) Declare two sets $\mathbf{encS'}$, $\mathbf{usedEKeys'}$ and initialize $\mathbf{usedEKeys'}[pk_0]$ with y_0.
  3) For each user $u_i \in \{u_1, \ldots, u_m, u_{m+1}\}$, do:
     a) Get $y_i \leftarrow \text{reqEKey}(pk_i)$ and then initialize $\mathbf{usedEKeys'}[pk_i] \leftarrow y_i$.
     b) Compute $\alpha_i \leftarrow \pi_{\text{ABKE}}(s_0, pk_i, x_0, y_i)$ then initialize $\mathbf{encS'}[pk_i] \leftarrow F(\alpha_i, s')$.
     c) For the new member u_{m+1}, do:
        i) compute $\theta \leftarrow \text{ElGamal.Enc}_{(k, y_{m+1})}(\alpha_{m+1})$ and $\text{encK}_{m+1} \leftarrow F(\alpha_{m+1}, k)$.
     d) Delete α_i
  4) Execute the function

     initAddMbr(pk_0, encK_{m+1}, θ, $\mathbf{encS'}$, $\mathbf{usedEKeys'}$).

- *Phase 2.* To get the new group key, each previous member $u_i \in \{u_1, \ldots, u_m\}$ executes the following operations:
  1) Get $(\mathbf{gKey}, \text{encK}_i, \text{encS}_i, y_i, y_0) \leftarrow \text{reqGKMat}(pk_i)$ and then discard encK_i.
  2) Select $x_i \in \mathbf{X}_i$ the secret key associated with y_i and then compute $\alpha_i \leftarrow \pi_{\text{ABKE}}(s_i, pk_0, x_i, y_0)$.
  3) Compute $s' \leftarrow F^{-1}(\alpha_i, \text{encS}_i)$ and $\beta' \leftarrow \mathbf{gKey}[0] - (k \times \mathbf{gKey}[1])$.
  4) Compute the new group key $\text{ssk}' \leftarrow \text{KDF}(\beta', s')$ and then delete the tuple (s', β').

In case of the new group member u_{m+1}, to obtain the new group key, it performs the following operations:

1) Execute the steps 1, 2, and 3 that the previous members $\{u_1, \ldots, u_m\}$ did, but at step 1, do not discard encK_i and at step 3, do not compute β'.
2) Compute $k \leftarrow F^{-1}(\alpha_{m+1}, \text{encK}_{m+1})$ and $\beta' \leftarrow \mathbf{gKey}[0] - (k \times \mathbf{gKey}[1])$.
3) Compute the new group key $\text{ssk}' \leftarrow \text{KDF}(\beta', s')$ and $h_{m+1} \leftarrow H'(\alpha_{m+1})$.
4) Discard the tuple $(s', \beta', \alpha_{m+1})$.

In case of the administrator u_0, it obtains the new group key by executing the following operations:

1) Get $\mathbf{gKey} \leftarrow \text{reqGKey}(pk_0)$.

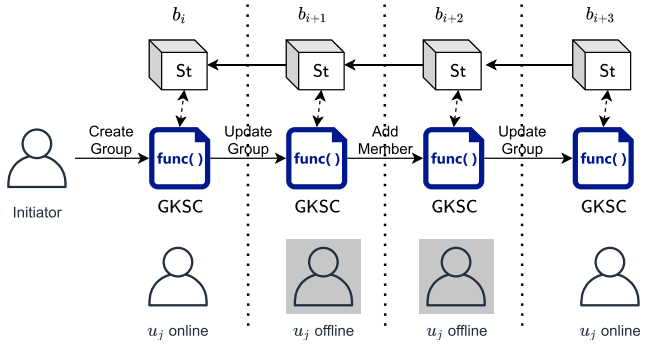


Fig. 7. u_j missed two group operations.

2) Compute $\beta' \leftarrow \mathbf{gKey}[0] - (k \times \mathbf{gKey}[1])$.
3) Compute the new group key $\mathsf{ssk}' \leftarrow \mathsf{KDF}(\beta', s')$, *(from phase 1, u_0 has s' in the memory)* and then delete the tuple (s', β').

Remove Member. A group administrator can remove one or multiple group members from its group. The remove member process will initiate an update of the current group key, and hence, will prevent the removed members from accessing future messages. For a group administrator u_0 to remove a set of users $\{u_i, \ldots, u_j\}$ from the group $\{u_0, u_1, \ldots, u_m\}$, it needs to re-execute the *Group Creation* process with the group $\{u_0, u_1, \ldots, u_m\}/\{u_i, \ldots, u_j\}$. However, at the end of *Phase 1*, instead of executing

$$\mathsf{initGrp}(pk_0, \mathsf{CRT}, \mathbf{encK}, \mathbf{encS}, \mathbf{A}, \mathbf{usedEKeys}),$$

u_0 executes

$$\mathsf{initGrp}(pk_0, \mathsf{RM}, \mathbf{encK}, \mathbf{encS}, \mathbf{A}, \mathbf{usedEKeys}).$$

Recovering Group Key. It is possible for a group member to recover the group key after missing multiple group operations. For instance, let us consider the scenario depicted by Fig. 7. A user u_j having an instance $\Pi^r_{u_j}$ with a group successfully derives the group key during the *Group Creation* process, which sets $\Pi^r_{u_j}.\mathsf{blkNum} = b_i$. However, while u_j is offline, three operations are performed on the group associated with $\Pi^r_{u_j}$ in the following order: *Update Group* $\rightarrow$ *Add Member* $\rightarrow$ *Update Group*. When u_j comes back online, $\Pi^r_{u_j}.\mathsf{ssk}$ does not match the group key anymore. To recover the current group Key, u_j simply starts by reading GKSC's St at block $\Pi^r_{u_j}.\mathsf{blkNum} + 1$, which is equal to b_{i+1} in this case, and performs different instructions to derive the group key based on the group operations that were performed until $\Pi^r_{u_j}.\mathsf{blkNum}$ is equal to the most recent block.

Handling Simultaneous Update Requests. The *Update Group* operation can be initiated by any group member at anytime. Therefore, it is possible for two or more group members to perform that operation at the same time. Other group operations might not be in such situation because they can only be triggered by the group administrator. We propose two ideas on how to handle such situation, but we would like to emphasize that they do not have a detrimental effect on the security of the proposed scheme:

- Allow operations to proceed: suppose two group members u_i and u_j simultaneously perform the *Update Group* operation. At the end of the *Phase 1*, u_i outputs a transaction $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_i$ and u_j outputs $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_j$. These transactions will trigger the GKSC's function UpdGKey and modify the GKSC's internal variables **encK**, **usedEKeys**, **gKey**. Therefore, miners will have to execute these transactions in sequence. However, modifying **encK**, **usedEKeys** will not have an adverse effect on the proposed scheme since they are not directly influencing the computation of the group key. Thus, let us focus on **gKey**. Suppose $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_i$ is executed before $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_j$. At the end of $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_i$'s execution, we have

$$\mathbf{gKey}[0] = H'(\alpha_0) + \cdots + H'(\alpha'_i) + \cdots + H'(\alpha_m) + k' \cdot \sum_{l=0}^{m} y'_l,$$

where, α'_i is the new contribution of u_i, k' is the new secret value and $\sum_{l=0}^{m} y'_l$ is the sum of fresh ephemeral keys used by u_i. Next, after the execution of $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_j$, we have

$$\mathbf{gKey}[0] = H'(\alpha_0) + \cdots + H'(\alpha'_i) + H'(\alpha'_j) + \cdots + H'(\alpha_m) + k' \cdot \sum_{l=0}^{m} y'_l + k'' \cdot \sum_{l=0}^{m} y''_l - k \cdot \sum_{l=0}^{m} y_l,$$

where α'_j is the new contribution of u_j, k is the old secret variable and $\sum_{l=0}^{m} y_l$ is the sum of ephemeral keys used before u_i and u_j execute the *Update Group* operation, and k'' is the new secret variable and $\sum_{l=0}^{m} y''_l$ is the sum of fresh ephemeral public keys that were used by u_j during its execution of *Update Group*.

Next, during *Phase 2* of *Update Group* member, groups members will compute:

$$\begin{aligned}\beta &= \mathbf{gKey}[0] - k'' \cdot \sum_{l=0}^{m} y''_l \\ &= H'(\alpha_0) + \cdots + H'(\alpha'_i) + H'(\alpha'_j) + \cdots + H'(\alpha_m) \\ &\quad + k' \cdot \sum_{l=0}^{m} y'_l - k \cdot \sum_{l=0}^{m} y_l,\end{aligned}$$

instead of

$$\beta = H'(\alpha_0) + \cdots + H'(\alpha'_i) + H'(\alpha'_j) + \cdots + H'(\alpha_m).$$

Therefore, allowing all operations to proceed will increase the size of the *pre*-group key by adding the term $k' \cdot \sum_{l=0}^{m} y'_l - k \cdot \sum_{l=0}^{m} y_l$, but this will not reduce the security of the key update process since a potential attacker will not be able to derive the value $k' \cdot \sum_{l=0}^{m} y'_l$.

- Use a time-lock: we can also augment the GKSC's function UpdGKey with a time-lock that will prevent the execution of UpdGKey for a given period once it is activated. Given a sequence of calls to UpdGKey, the goal is for the first call in the sequence to activate the time-lock, which will cause the subsequent calls to abort by preventing them from triggering the execution of UpdGKey. For instance, suppose two users u_i, u_j try to execute the *Update Group* operation simultaneously and issue the transactions $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_i$, $\{^*\mathsf{tx}_{\mathsf{UpdGkey}}\}_j$,

respectively. However, suppose $\{^{*}\mathsf{tx}_{\mathsf{UpdGkey}}\}_i$ is executed first, and after its execution, the time-lock is activated. This will force $\{^{*}\mathsf{tx}_{\mathsf{UpdGkey}}\}_j$ to abort and allow u_j to complete the execution of *Update Group* that was started by u_i before re-initiating an *Update Group* process.

6 Security Analysis

In this Section, we analyze the proposed scheme and prove that the proposed asynchronous GKE is secure against the adversarial model defined in Section 4. To do so, first we introduce the following two important definitions.

Definition 4 (Partnering). *Partnering allows to capture the fact that two or more instances that are part of the same group have derived the same group key. We say that the instances in a set* $\{\Pi^{t_0}_{u_0}, \Pi^{t_1}_{u_1}, \ldots\}$ *are partnered when the following condition are satisfied:*

- *The status of each instance is set to accept.*
- *All instances are at the same state d.*
- *The list of members* Mbrs *is the same for each instance.*
- *All instances use the same smart contract ID (*SCID*).*
- *The session variable* blkNum *is the same for all instances.*

Definition 5 (Freshness). *Freshness defines the scope in which an adversary cannot trivially obtain the session key of an instance* ${}^{d}\Pi^{t}_{u}$*. Hence, an instance* ${}^{d}\Pi^{t}_{u}$ *is fresh if the following conditions are satisfied:*

- ${}^{d}\Pi^{t}_{u}.\mathsf{status} = accept.$
- *An adversary has not issued a query* $\mathsf{Reveal}({}^{d}\Pi^{t}_{u})$*, and for all instance* ${}^{d}\Pi^{r}_{v}$ *partenered to* ${}^{d}\Pi^{t}_{u}$*, the adversary has not issue a query* $\mathsf{Reveal}({}^{d}\Pi^{r}_{v})$ *or* $\mathsf{Test}({}^{d}\Pi^{r}_{v})$*.*
- *An adversary has not issued a query* $\mathsf{Corrupt}({}^{d}\Pi^{t}_{u})$ *before* ${}^{d}\Pi^{t}_{u}$ accept.
- *If* $d > 0$*, then* ${}^{d-1}\Pi^{t}_{u}$ *is also fresh.*

Definition 6 (Semantic Security). *This captures the fact that it should be hard for a PPT adversary to distinguish the key produced by a fresh instance of the protocol from a random key. Given a challenger* Ch*, an instance* Π*, and a PPT adversary* $\mathcal{A}$ *that has access to the queries defined in section 4.2 with the restriction that* Test *queries can only be performed on fresh instances, a GKE is secure is* $\mathcal{A}$*'s advantage* $\mathsf{Adv}_{\mathcal{A}}$ *at correctly guessing the bit used during the response of the* Test *query is negligible, where*

$$\mathsf{Adv}_{\mathcal{A}} = \left|\Pr[b' \leftarrow \mathcal{A} : b = b'] - \frac{1}{2}\right|.$$

6.1 Smart Contract Leakage Analysis

In this subsection, we analyse the information that an adversary can obtain from GKSC. We show that under the Random Oracle (RO) model, the adversary cannot obtain the session key from the data stored in GKSC's state St.

First, by reading St.eKeysMap, an adversary can learn which users decided to form a group because to be part of a group, one needs to upload a set of ephemeral keys. Also, by reading St.**usedEKeys**, an adversary can learn the actual size and participants of a group since St.**usedEKeys** contains the list of ephemeral keys that were used by the group's initiator. However, such information can also be inferred by an adversary that has complete control over the network. In addition, just getting the used public ephemeral keys does not leak the session key. Next, an adversary can determine the group administrator/initiator by reading St.admin. Replacing GKSC by a TTP will not prevent an attacker from obtaining such information since with total control over the network, the adversary can get the size of a group and its members. Furthermore, if the TTP is corrupted, then an attacker can get the identity of the group's initiator.

Theorem 1. *Assume* KDF *is a RO. Consider* GSKC *to be the* SC *that is used in an instance* Π^{t}_{u}*. If* F *is* $(t_{\mathsf{CCA}}, \epsilon_{\mathsf{CCA}})$*-CCA secure and* π_{ABKE} *is a* $(t_{\mathsf{ABKE}}, \epsilon_{\mathsf{ABKE}})$*-secure asynchronous key exchange, then for any stage* $d \in \mathbb{N}$ *of* Π^{t}_{u}*, the probability that a PPT adversary* $\mathcal{A}$ *obtains* ssk *by only accessing* GSKC*'s state* St *is negligible in the values* $\epsilon_{\mathsf{CCA}}, \epsilon_{\mathsf{ABKE}}$ *and the security parameter* 1^{ν}*, where its probability is defined as follows:*

$$\Pr[\mathsf{ssk}_{\mathcal{A}} \leftarrow \mathcal{A}(1^{\nu}, \mathsf{St}) : \mathsf{ssk}_{\mathcal{A}} = \mathsf{ssk}].$$

Proof. Based on the type of operation performed by group members, the session key is computed as $\mathsf{ssk} \leftarrow \mathsf{KDF}(\beta, s)$ or $\mathsf{ssk} \leftarrow \mathsf{KDF}(\beta, \mathsf{ssk}')$, where $\beta = \sum_{i=0}^{m} H'(\alpha_i)$, α_i is the contribution of u_i, and ssk′ is the session key at the previous stage of Π^{t}_{u}. Since KDF is a RO, the only way for $\mathcal{A}$ to obtain ssk is to query KDF at (β, s) or (β, ssk') depending on the cases. In St we have four variables that can help an adversary to obtain the session key:

- the set **usedEKeys**.
- the set **gKey** in which $\mathbf{gKey}[0] = \beta + k \times \sum_{i=0}^{m} y_i$ and $\mathbf{gKey}[1] = \sum_{i=0}^{m} y_i$.
- The sets **enck** and **encS**. Let encK_i and encS_i be elements of **enck** and **encS** at position i respectively. We have $\mathsf{encK}_i = F(\alpha_i, k)$ and $\mathsf{encS}_i = F(\alpha_i, s)$, where α_i is the result of π_{ABKE} between the group initiator and a group member at position i, i.e., the contribution of u_i.

It is impossible for $\mathcal{A}$ to extract β from $\mathbf{gKey}[0]$ unless it is able to find the value k.

Let us consider the following cases:

Case ssk. This case happens when *Group Creation*, *Add Member* or *Remove Member* is executed. To show that $\mathcal{A}$ cannot compute ssk using only St, we use a series of games:

Game 0. In this game, the proposed protocol is executed normally with the restriction that only the following operations can be executed: *Group Creation, Add Member* and *Remove Member*. At the end of the protocol, GKSC's state St and the security parameter 1^{ν} is given to $\mathcal{A}$. Let p_0 denotes the success's probability of $\mathcal{A}$ in this game, i.e., the probability that $\mathcal{A}$ obtains ssk.

Game 1. This game is similar to *Game 0* with the exception that π_{ABKE} always returns a random string. Let p_1 denotes the success's probability of $\mathcal{A}$ in this game. We show that

$$|p_0 - p_1| \leq \epsilon_{\mathsf{ABKE}}. \tag{1}$$

Let $\mathcal{A}'$ be an adversary that tries to distinguish the output of π_{ABKE} from random. Generally, in a security model of

π_{ABKE}, $\mathcal{A}'$ has access to a set of queries among which there is a test query that when executed, it either returns the session key computed by π_{ABKE} or an output sampled at random [22]. Let us consider the following experiment ($\mathsf{EXP}^b{}_{\mathsf{ABKE}}$):

1) $\mathcal{A}'$ selects an instance of π_{ABKE} between the group initiator u_0 and a responder u_i.
2) $\mathcal{A}'$ executes different queries in the security model of π_{ABKE} except the test query.
3) $\mathcal{A}'$ executes the test query. If $b = 0$, the test query sets skey to be the computed session key. Otherwise, it sets skey to be a random string. Next, $\mathcal{A}'$ receives skey.
4) $\mathcal{A}'$ generates $\widetilde{\mathsf{St}}$ which is a collection of variables that mimics St, a state of GKSC between u_0 and u_i as follows:
   a) Randomly sample $k \in \mathbb{Z}_n^*, s \in \mathbb{F}_p, \beta \in E(\mathbb{F}_p)$, and set $\mathsf{ssk} \leftarrow \mathsf{KDF}(\beta, s)$.
   b) Set $\widetilde{\mathsf{St}}.\mathbf{gKey}[0] = \beta + k \times (y_0 + y_i)$, $\widetilde{\mathsf{St}}.\mathbf{gKey}[1] = y_0 + y_i$, and add y_i, y_0 to $\widetilde{\mathsf{St}}.\mathbf{usedEKeys}$, where y_0, y_i are the ephemeral keys of u_0 and u_i, respectively.
   c) Compute $\mathsf{encS}_i \leftarrow F(\mathsf{skey}, s)$, $\mathsf{encK}_i \leftarrow F(\mathsf{skey}, k)$ and add them to $\widetilde{\mathsf{St}}.\mathbf{encS}$ and $\widetilde{\mathsf{St}}.\mathbf{encK}$ respectively.
5) $\mathcal{A}'$ computes $\mathsf{ssk}_{\mathcal{A}} \leftarrow A(1^\nu, \widetilde{\mathsf{St}})$. If $\mathsf{ssk}_{\mathcal{A}} = \mathsf{ssk}$, $\mathcal{A}'$ returns 1. Else, it returns 0.
6) $\mathcal{A}'$ wins if it outputs 1.

$\mathcal{A}'$ is PPT since $\mathcal{A}$ is also PPT.

Remark, we limited $\mathcal{A}'$ simulation to two parties (u_0, u_i) for simplicity, but this can be extended to a polynomial number of parties by allowing $\mathcal{A}'$ to attack multiple π_{ABKE} instances in parallel.

In $\mathsf{EXP}^0{}_{\mathsf{ABKE}}$, the distribution of $\widetilde{\mathsf{St}}$ is similar to the distribution of St in *Game 0*, and in $\mathsf{EXP}^1{}_{\mathsf{ABKE}}$, it is similar to the distribution of St in *Game 1*. Therefore

$$\left|\Pr[\mathcal{A}' \text{ wins} \,|\, \mathsf{EXP}^0{}_{\mathsf{ABKE}}] - \Pr[\mathcal{A}' \text{ wins} \,|\, \mathsf{EXP}^1{}_{\mathsf{ABKE}}]\right| = |p_0 - p_1|,$$

but

$$\left|\Pr[\mathcal{A}' \text{ wins} \,|\, \mathsf{EXP}^0{}_{\mathsf{ABKE}}] - \Pr[\mathcal{A}' \text{ wins} \,|\, \mathsf{EXP}^1{}_{\mathsf{ABKE}}]\right| \leq \epsilon_{\mathsf{ABKE}}.$$

Combining the results, we reach equation 1.

Game 2. This game is similar to *Game 1* with the exception that evaluating F in encryption mode returns a random value in $\mathcal{C}$ independent of its inputs. Let p_2 be the success probability of $\mathcal{A}$ in this game. Unless $\mathcal{A}$ is able to break the security of F, $\mathcal{A}$ should not be able to see the difference between *Game 1* and *Game 2*. However, such an event can happen with a probability of at most ϵ_{CCA}. Hence,

$$|p_1 - p_2| \leq \epsilon_{\mathsf{CCA}}.$$

Furthermore, in *Game 2*, the output of π_{ABKE} is independent from pk_0, pk_i, y_0, y_i, and the output of F is independent from its inputs. Hence, the best $\mathcal{A}$ can do to extract k and s from St to compute ssk is to randomly guess their values. Therefore, for the case $\mathsf{ssk} \leftarrow \mathsf{KDF}(\beta, s)$, the probability for $\mathcal{A}$ to extract the session key using St is negligible.

Case ssk This case happens when the operation *Update Group* is executed. Let us assume that $\mathcal{A}$ is in possession of ssk′ since the execution of the *Update Group* operation may indicate the corruption of a group member. Thus the objective of $\mathcal{A}$ is to find β given St. As stated earlier, $\mathcal{A}$ cannot extracts β from $\mathsf{St}.\mathbf{gKey}[0]$ without knowing k. Therefore, either $\mathcal{A}$ randomly guesses the value k or $\mathcal{A}$ tries to extract k from encK_i. For the former option, the success probability of $\mathcal{A}$ is bounded above by $q \cdot |\mathbb{Z}_n^*|^{-1}$, where q is the maximum number of guesses that $\mathcal{A}$ can make, and for the later option, its success probability is bounded above by ϵ_{CCA}.

Since the above cases are exhaustive, we conclude that the success probability of $\mathcal{A}$ to obtain ssk from St is negligible in the values $\epsilon_{\mathsf{CCA}}, \epsilon_{\mathsf{ABKE}}$ and the security parameter 1^ν This proves Theorem 1. □

6.2 Perfect Forward Secrecy

In this subsection, we show that our proposed scheme supports PFS.

Given an instance Π_u^t, PFS stipulates that the execution of a query $\mathsf{Corrupt}({}^{d_0}\Pi_u^t)$ by an adversary should not reveal the session keys derived in a previous stage d, where $0 \leq d \leq d_0$, or derived in another instance Π_u^s, where $s \neq t$, that terminated before Π_u^t starts.

Assume that KDF is a RO and the discrete logarithm assumption holds in $E(\mathbb{F}_q)$ and π_{ABKE} is a strong one-round authenticated key exchange protocol. Consider that for an instance Π_u^t, a PPT adversary $\mathcal{A}$ executes the query $\mathsf{Corrupt}({}^{d_0}\Pi_u^t)$. This reveals the state of Π_u^t at stage d_0, which includes the long-term private key. Depending on when the adversary executes the query, we have two cases:

- *Case 1:* $\mathcal{A}$ executes the query before ${}^{d_0}\Pi_u^t$ *accepts*. In this case, the adversary can easily obtain the session key that was derived at stage d_0.
- *Case 2:* $\mathcal{A}$ executes the query after ${}^{d_0}\Pi_u^t$ *accepts*, i.e., ${}^{d_0}\Pi_u^t$ is *fresh*. In this case, the adversary cannot easily obtain the session key that was derived at stage d_0 unless it executes the query $\mathsf{Reveal}({}^{d_0}\Pi_u^t)$ because the secret ephemeral keys and either the secret variable s (in case *Group Creation*, *Add Member* or *Remove Member* is executed), or the previous session key ssk′ (in case *Update Group* is executed) that were used during the computation of the session key were erased once ${}^{d_0}\Pi_u^t$ *accepted*.

In both cases, the adversary cannot obtain session keys that were derived in the previous stages or instances for reasons that we explain in the following lines. First, since KDF is a RO, each derived session key is random and uniform, and therefore it does not reveal anything about other session keys. Second, each stage requires fresh ephemeral keys that are discarded after their usage. Though, the adversary can obtain the public counterpart of each ephemeral key and the long-term private key that were used at a stage, this is not sufficient to derive a session key unless the adversary is able to extract the output of π_{ABKE} using only the long-term private key and the ephemeral public keys, which means that it can breaks the security of π_{ABKE}, or is able to

breaks the discrete logarithm assumption. Therefore, our proposed scheme satisfies PFS.

6.3 Post-Compromise Security

In this subsection, we demonstrate that our proposed scheme supports PCS.

Given an instance Π_u^t, PCS stipulates that if an adversary executes a query $\mathsf{Corrupt}({}^{d_0}\Pi_u^t)$, instances partnered to Π_u^t should be able to re-establish security guarantees [12]. In the case of our proposed scheme, this is ensured by a successful execution of the *Update Group* process.

After the adversary has executed the query $\mathsf{Corrupt}({}^{d_0}\Pi_u^t)$, it has the state of Π_u^t at stage d_0 in its possession and can possibly obtain the session key that was derived at that stage. Since it has access to u's long-term key, it can easily prevent the future stages from *accepting* by using queries defined in subsection 4.2. Now, let us assume that during the stage $d_0 + i$ of instance Π_u^t, the adversary goes passive, i.e., it does not issue queries anymore, and user u successfully initiates the *Update Group* process. Once the execution of the *Update Group* process is completed, the instance ${}^{d_0+i}\Pi_u^t$ is now *fresh* and so are the instances $\{{}^{d_0+i+1}\Pi_u^t, {}^{d_0+i+2}\Pi_u^t, \ldots\}$. Hence, our proposed scheme satisfies PCS under a passive adversary

6.4 Semantic Security

Security Experiment. We define a security game between a challenger Ch and a PPT adversary $\mathcal{A}$. At the beginning of the game, Ch runs the *Setup* process (5.2) and initializes a set $\{u_0, u_1, \ldots, u_m\}$ of users. Then, it sends GKSC's ID to $\mathcal{A}$. Next, $\mathcal{A}$ can execute any queries defined in section 4.2 except for the Test query that can only be executed once and on a *fresh* instance (Definition 5). Once $\mathcal{A}$ executes the Test query, it must output a guess bit b'. $\mathcal{A}$ wins if its guessed bit b' is correct.

Theorem 2. *Assume* KDF *is a RO, and let* $\mathcal{D}(\mathsf{KDF})$ *be the domain space of the* KDF. *If* F *is* $(t_{\mathsf{CCA}}, \epsilon_{\mathsf{CCA}})$*-CCA secure,* π_{ABKE} *is a* $(t_{\mathsf{ABKE}}, \epsilon_{\mathsf{ABKE}})$*-secure asynchronous key exchange, and* Φ *is a* $(t_{\mathsf{CMA}}, \epsilon_{\mathsf{CMA}})$*-CMA secure digital signature,* $\mathcal{A}$*'s advantage against our proposed game is bounded by the following inequality*

$$\mathsf{Adv}_{\mathcal{A}} \leq \epsilon_{\mathsf{CCA}} + \epsilon_{\mathsf{ABKE}} + \epsilon_{\mathsf{St}} + \eta \cdot \epsilon_{\mathsf{CMA}} + \frac{\binom{\eta \cdot T \cdot D}{2}}{n} + \frac{q_r}{2^{|\mathcal{D}(\mathsf{KDF})|}},$$

where η *is the group size,* T *is the maximum number of instances for a given user,* D *is the maximum number of stages for a given instance,* q_r *is the maximum number of RO queries that* $\mathcal{A}$ *can perform and* $\epsilon_{\mathsf{St}} = \Pr[\mathsf{ssk}_{\mathcal{A}} \leftarrow \mathcal{A}(1^v, \mathsf{St}) : \mathsf{ssk}_{\mathcal{A}} = \mathsf{ssk}]$.

Proof. We present our proof as a sequence of related games between a challenger Ch and an adversary $\mathcal{A}$. We use *Game i* to denote the ith game and p_i to denote the success probability of $\mathcal{A}$ in *Game i*.

Game 0. This is the *security experiment* defined above. The success probability of $\mathcal{A}$ is p_0.

Game 1. This game is identical to *Game 0* with the exception that π_{ABKE} always returns a random string. Therefore

$$|p_0 - p_1| \leq \epsilon_{\mathsf{ABKE}}. \tag{2}$$

Game 2. This game is identical to *Game 1* with the exception that F always returns a random element from $\mathcal{C}$. Therefore

$$|p_1 - p_2| \leq \epsilon_{\mathsf{CCA}}. \tag{3}$$

Game 3. This game is identical to *Game 2* with the exception that $\mathcal{A}$ queries GKSC's state St and successfully derives the session key from St before outputting its guess. If that event happens, Ch halts the game and $\mathcal{A}$ loses. Let us consider that event to be BrkSt. As we can see, *Game 0* = *Game 1* unless BrkSt occurs. Hence,

$$|p_2 - p_3| \leq \Pr[\mathsf{BrkSt}].$$

We have $\Pr[\mathsf{BrkSt}] = \Pr[\mathsf{ssk}_{\mathcal{A}} \leftarrow \mathcal{A}(1^v, \mathsf{St}) : \mathsf{ssk}_{\mathcal{A}} = \mathsf{ssk}]$. Let $\epsilon_{\mathsf{St}} = \Pr[\mathsf{ssk}_{\mathcal{A}} \leftarrow \mathcal{A}(1^v, \mathsf{St}) : \mathsf{ssk}_{\mathcal{A}} = \mathsf{ssk}]$ Therefore,

$$|p_2 - p_3| \leq \epsilon_{\mathsf{St}}. \tag{4}$$

However, from Theorem 1, we know that ϵ_{St} is negligible.

Game 4. This game is identical to *Game 3*, with the exception that $\mathcal{A}$ can produce multiple queries $\mathsf{Send}(\Pi_u^t, \delta)$, where $u \in \{u_0, u_1, \ldots, u_m\}$, that can lead Π_u^t to *accept*. More specifically, δ is not a message that was previously generated by a user $v \in \{u_0, u_1, \ldots, u_m\}$. Let us consider that event to be Forge. If Ch detects that the event Forge happened, it halts the game and $\mathcal{A}$ loses.

Therefore

$$|p_3 - p_4| \leq \Pr[\mathsf{Forge}].$$

In the worst case, $\mathcal{A}$ only needs to produce one query $\mathsf{Send}(\Pi_u^t, \delta)$ that makes Π_u^t *accepts*. Because, in the proposed scheme, participants communicate through signed messages/transactions, the message δ produced by $\mathcal{A}$ will also need to be signed. Since $\mathcal{A}$ can issue a query $\mathsf{Send}(\Pi_u^t, \delta)$ to any member $u \in \{u_0, \ldots, u_m\}$, we have

$$\Pr[\mathsf{Forge}] \leq \eta.\epsilon_{\mathsf{CMA}}.$$

Therefore

$$|p_3 - p_4| \leq \eta.\epsilon_{\mathsf{CMA}}, \tag{5}$$

where $\eta = |\{u_0, \ldots, u_m\}|$.

Game 5. This game is identical to *Game 4*, with the exception that Ch halts the game and $\mathcal{A}$ loses if two different stages ${}^{d}\Pi_u^t$ and ${}^{d'}\Pi_u^t$ generate the same secret random values. Let us consider that event to be $\mathsf{Collision}$. We have

$$|p_4 - p_5| \leq \Pr[\mathsf{Collision}].$$

However, we have $\Pr[\mathsf{Collision}] \leq \binom{\eta \cdot T \cdot D}{2} \cdot n^{-1}$, where T is the maximum number of instance, D the maximum number of stages in an instance, and n is the order of $E(\mathbb{F}_q)$. Therefore

$$|p_4 - p_5| \leq \binom{\eta \cdot T \cdot D}{2} \cdot n^{-1}. \tag{6}$$

Furthermore, the best $\mathcal{A}$ can do to win *Game 5* is to query the KDF at the correct entries. Let q_r be the maximum number of RO queries that $\mathcal{A}$ can make. we have

$$\left| p_5 - \frac{1}{2} \right| \leq \frac{q_r}{2^{|\mathcal{D}(\mathsf{KDF})|}}. \quad (7)$$

Combining inequalities (2), (3), (4), (5), (6), (7), we obtain

$$\mathsf{Adv}_{\mathcal{A}} \leq \epsilon_{\mathsf{CCA}} + \epsilon_{\mathsf{ABKE}} + \epsilon_{\mathsf{St}} + \eta \cdot \epsilon_{\mathsf{CMA}} + \frac{\binom{\eta \cdot T \cdot D}{2}}{n} + \frac{q_r}{2^{|\mathcal{D}(\mathsf{KDF})|}}.$$

Hence, the proof is completed. □

7 Implementation and Evaluation

In this section, we describe and evaluate two implementations of our protocol using Ethereum [17]: a version in which all key materials are stored in the GKSC's state (this is the SC defined in Fig. 3), and a version in which only the key materials necessary for the execution of the core GKSC functionalities (CreateGrp(), UpdGKey(), AddMbr()) are stored on GKSC's state and the others on a distributed storage system (presented in Appendix as a light version, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2022.3189977).

7.1 Cryptographic Primitives

In the construction of our protocol, we used different cryptographic primitives as black-boxes. Besides simplifying the complexity of the protocol, this approach allows anyone to swap them with concrete instances based on its environment needs. Following is a description of the instantiations we used in our implementation:

- *Elliptic Curve Group* $E(\mathbb{F}_p)$. We used a group based on Barreto-Naehrig (BN) curve over 256-bit prime field [26]. This choice was motivated by the fact that Ethereum supports elliptic curve points addition as built-in operation, which offers the prospect of a reduce gas cost [27].
- *Digital Signature* Φ. We used Elliptic-Curve Digital Signature Algorithm (ECDSA) [28] since it is the signature scheme used by Ethereum [17].
- *Asynchronous Biparty KE* π_{ABKE}. We used the Extended Triple Diffie-Hellman (X3DH) protocol [23]. Given an initiator's private keys $sk_I, x_I \in \mathbb{Z}_n$ and a responder's public keys $pk_R, y_R \in E(\mathbb{F}_p)$, we have

$$\mathsf{X3DH}(sk_I, pk_R, x_I, y_R) = \mathsf{KDF}(sk_I.y_R || x_I.pk_R || x_I.y_R)$$

- *Cipher* F. We used Chacha20-Poly1305 from the PyCryptodome library [29]. This choice was motivated by its computational efficiency [30].
- *Key Derivation Function* KDF. We used the HKDF scheme [31] from the Py_ECC library [32].
- *Hash to curve* H'. We have relaxed the CHR property of H'. Given a base point $G \in E(\mathbb{F}_p)$ with order n and a message $m \in \{0,1\}^*$, $H'(m) = [m \bmod n] \times G$.

7.2 Simulation Environment

We implemented the *User's algorithms* (Section 5.1) with Python (version 3.6.7) and used Web3.Py [33] to interface the Ethereum blockchain. We used Truffle Suite [34] to simulate a local instance of the Ethereum blockchain and implemented GKSC. Our implementation was written and complied using Solidity version 0.5.16. It has successfully passed the formal verification process by means of the automated tools OYENTE and OYENTE IPFS[1] [35]. A non-complied version of our implementation and testing results are available from our website.[2]

Each participant had a long-term key pair defined over BN curve and a signing key defined over SECP256K1 curve [36] (this is the curve used by Ethereum to define the address of an account and sign transactions [17]). The simulation was performed on a Windows 10 computer with a 4.00 GHz Intel core i7 and 32 GB of RAM.

7.3 Performance Evaluation

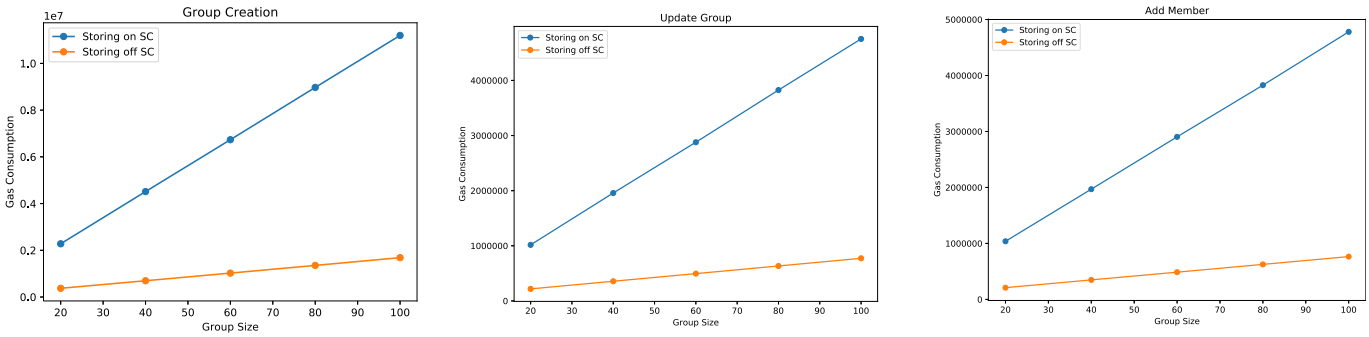
We measured the gas consumption of the functions CreateGrp(), UpdGKey(), and AddMbr() for different group size in two configurations: (1) all key materials are stored in St; and (2) only the pre-group key computed by GKSC is stored in St and the rest in a distributed storage such as IPFS [37] (we refer to this version as Lite GKSC). Fig. 8 shows the difference in average gas consumption between both implementations for each function. In both cases, CreateGrp() is the operation that consumes the most gas, whereas UpdGKey() and AddMbr() consume approximately the same amount of gas. This is because variables stored in St are initialized during the execution of CreateGrp() while they are updated during the execution of UpdGKey() and AddMbr(). It is worth noting that the group administrator is the one paying for CreateGrp() and AddMbr() gas consumption, whereas the group member initiating UpdGKey() is the one paying for its gas consumption. Overall using the lite GKSC consumes approximately 83% less gas than the full GKSC for each operation. This is because storing data on Ethereum is expensive (the store operation cost 20,000 gas [17]). By combining lite GKSC with a distributed storage, the group administrator pays approximately 1685263 gas for the execution of CreateGrp() and 772963 gas for the execution of AddMbr() for a group size of 100 members. In the case of UpdGKey(), a group member pays 764771 gas.

8 Related Works

The existing works in the literature focused on GKA schemes that reduce the computational burden of the group members by delegating a part of the process to a TTP [38], [39], [40]. In these approaches, each group member sends key materials to the TTP. Then, the TTP computes and broadcasts a pre-group key to group members which will be used to derive the final group key. However, all group members need to be online and send their contributions during a given time-frame. In addition, these schemes do not provide post-compromised security and suffer from the drawbacks of using a TTP.

In GKA, the main role of a TTP is to provide Public Key Infrastructure (PKI)-related operations. PKIs are used to store and manage public encryption keys used by nodes in a network. To mitigate the issues of traditional PKIs, e.g.,

1. https://oyente.tech/
2. http://i2s.kennesaw.edu/resources



(a) Group Creation (b) Update Group (c) Add Member

Fig. 8. Average gas consumption based on group size.

high centralization, decentralized PKIs based on blockchain have been proposed [16], [41], [42]. Since blockchain is immutable, blockchain-based solutions improve the integrity of public keys. However, using decentralized PKIs do not completely shield GKAs, and thus TTPs are still required to perform some operations such as prekey computation. To use blockchain in conjunction with GKA, Schindler *et al.* proposed a distributed key generation system that leverages Ethereum's smart contract for communication [43]. In their scheme, TTP is no longer needed to handle keys and some operations are executed on Ethereum through smart contracts for devices' efficiency. However, all group members must be online during the initial operation, and a subset of group members must cooperate to obtain the secret key. This is not practical in an environment where some members have an intermittent connection.

A TreeKEM, asynchronous decentralized key management for large dynamic groups, was proposed by IETF Message-Layer Security (MLS) working group [44]. After then, many schemes were proposed to enhance the security and efficiency of it. Alwen *et al.* pointed out that the TreeKEM does not provide an adequate form of the forward secrecy and proposed an extended version, named RTreeKEM, in order to address the insecurity [45]. A rigorous security proof which concluded that the basic instantiation of MLS is a Secure Group Messaging (SGM) is provided [46]. It also provides a basic construction of SGM protocl based on several primitives. A key tree grafting scheme was proposed to make it possible to efficiently deal with users belong to multiple groups [47].

Most importantly, none of the existing works offer post-compromise security and asynchronism concurrently without a TTP. Moreover, most schemes were developed without considering smart contract environments, and therefore, additional efforts are required in order to make them adequately work on such environments. Some of the existing schemes provide those requirements for large group [7], [48]. However, they rely on a TTP and do not support the delegation of computation, which makes them impractical for resource-constrained IoT devices.

9 Conclusion

In this paper, we presented an asynchronous group key exchange scheme based on blockchain and smart contracts that is resistant to common attacks targeted at trusted third parties. The use of smart contracts allows us to reduce the memory and computational load of the participants. The scheme provides perfect forward secrecy and post-compromise security. Furthermore, it allows the addition of the participants to the group and removal of the participants from the group. Our simulation results show that the scheme can easily support 100 members when paired with a distributed storage system.

As a future work, we plan to explore the use of Randomness Beacon [49] to directly generate random secret values on the smart contract, and therefore, aim at further reducing the computational load of the participants.

References

[1] V. Y. Kemmoe, Y. Kwon, S. Shin, R. Hussain, S. Cho, and J. Son, "Leveraging smart contracts for asynchronous group key agreement in Internet of Things," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 2020, pp. 70–75.

[2] T. Brecher, E. Bresson, and M. Manulis, "Fully robust tree-Diffie-Hellman group key exchange," in *Proc. Int. Conf. Cryptol. Netw. Secur.*, 2009, pp. 478–497.

[3] M. F. Zanjani, S. M. A. Abhari, and A. G. Chefranov, "Group key exchange protocol based on Diffie-Hellman technique in ad-hoc network," in *Proc. 7th Int. Conf. Secur. Inf. Netw.*, 2014, pp. 166–169.

[4] P. Rahimi and C. Chrysostomou, "Improving the network lifetime and performance of wireless sensor networks for iot applications based on fuzzy logic," in *Proc. 15th Int. Conf. Distrib. Comput. Sensor Syst.*, 2019, pp. 667–674.

[5] C. Cachin and R. Strobl, "Asynchronous group key exchange with failures," in *Proc. 23rd Annu. ACM Symp. Princ. Distrib. Comput.*, 2004, pp. 357–366.

[6] C. Boyd, G. T. Davies, K. Gjøsteen, and Y. Jiang, "Offline assisted group key exchange," in *Proc. 7th Int. Conf. Secur. Inf. Netw.*, 2018, pp. 268–285.

[7] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner, "On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1802–1819.

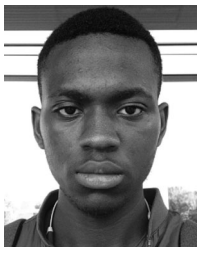
[8] Z. Dong, K. Kane, and L. J. Camp, "Detection of rogue certificates from trusted certificate authorities using deep neural networks," *ACM Trans. Privacy Secur.*, vol. 19, no. 2, pp. 1–31, Sep. 2016.

[9] C. Hlauschek, M. Gruber, F. Fankhauser, and C. Schanes, "Prying open pandora's box: KCI attacks against TLS," in *Proc. 9th USENIX Workshop Offensive Technol.*, 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/hlauschek

[10] Y. Meidan *et al.*, "N-baIoT—Network-based detection of IoT botnet attacks using deep autoencoders," *IEEE Pervasive Comput.*, vol. 17, no. 3, pp. 12–22, Jul.–Sep. 2018.

[11] W. Diffie, P. C. Van Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges," *Des., Codes Cryptogr.*, vol. 2, pp. 107–125, 1992.

[12] K. Cohn-Gorden, C. Cremers, and L. Garratt, "On post-compromise security," in *Proc. IEEE 29th Comput. Secur. Found. Symp.*, 2016, pp. 164–178.
[13] C. Cremers, J. Fairoze, B. Kiesl, and A. Naska, "Clone detection in secure messaging: Improving post-compromise security in practice," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 1481–1495.
[14] B. Rodrigues, T. Bocek, A. Lareida, D. Hausheer, S. Rafati, and B. Stiller, "A blockchain-based architecture for collaborative DDoS mitigation with smart contracts," in *Proc. IFIP Int. Conf. Auton. Infrastructure, Manage. Secur.*, 2017, pp. 16–29.
[15] S. Wani, M. Imthiyas, H. Almohamedh, K. M. Alhamed, S. Almotairi, and Y. Gulzar, "Distributed denial of service (ddos) mitigation using blockchain—A comprehensive insight," *Symmetry*, vol. 13, no. 2, 2021. [Online]. Available: https://www.mdpi.com/2073–8994/13/2/227
[16] M. Al-Bassam, "SCPKI: A smart contract-based PKI and identity system," in *Proc. ACM Workshop Blockchain, Cryptocurrencies Contracts*, 2017, pp. 35–40.
[17] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger petersburg version 41c1837 – 2021–02–1," pp. 1–39, 2019. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf
[18] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Proc. Annu. Int. Cryptol. Conf.*, 2017, pp. 357–388.
[19] K. Wüst and A. Gervais, "Do you need a blockchain?," in *Proc. Crypto Valley Conf. Blockchain Technol.*, 2018, pp. 45–54.
[20] Block.One, Eos.io technical white paper v2, Accessed: Mar. 10, 2021, 2018. [Online]. Available: https://github.com/EOSIO/Documentation
[21] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proc. Adv. Cryptol.*, 1985, pp. 10–18.
[22] H. Krawczyk, "HMQV: A high-performance secure Diffie-Hellman protocol," *Cryptol. ePrint Arch.*, Tech. Rep. 2005/176, 2005. [Online]. Available: https://eprint.iacr.org/2005/176
[23] M. Marlinspike and T. Perrin, "The X3DH key agreement protocol," 2016, pp. 1–11. [Online]. Available: https://www.signal.org/docs/specifications/x3dh/x3dh.pdf
[24] M. Fischlin and F. Günther, "Multi-stage key exchange and the case of google's quic protocol," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1193–1204.
[25] Ethereum.org, Solidity, Accessed: Apr. 18, 2021. [Online]. Available: https://docs.soliditylang.org
[26] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *Proc. Int. Workshop Sel. Areas Cryptogr.*, 2006, pp. 319–331.
[27] A. S. Cardozo and Z. Williamson, Eip-1108: Reduce alt_bn128 precompile gas costs, *Ethereum Improvement Proposals*, 2018. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1108
[28] D. R. L. Brown, "Generic groups, collision resistance, and ECDSA," *Des., Codes Cryptogr.*, vol. 35, pp. 119–152, 2005.
[29] Pycryptodome, 2021. [Online]. Available: https://github.com/Legrandin/pycryptodome
[30] N. Sullivan, "Do the chacha: Better mobile performance with cryptography," Accessed: May 4, 2021. [Online]. Available: https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/
[31] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Proc. Adv. Cryptol.*, 2010, pp. 631–648.
[32] Py_ecc, 2021. [Online]. Available: https://github.com/ethereum/py_ecc
[33] Ethereum.org, Web3.py, 2021. [Online]. Available: https://github.com/ethereum/web3.py
[34] ConsenSys, Truffle suite, 2021. [Online]. Available: https://www.trufflesuite.com/
[35] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," in *Proc. Australas. Comput. Sci. Week Multiconference*, pp. 1–10, 2021.
[36] D. R. L. Brown, "Sec 2: Recommended elliptic curve domain parameters," 2010, pp. 1–37. [Online]. Available: https://www.secg.org/sec2-v2.pdf
[37] J. Benet, "IPFS - content addressed, versioned, P2P file system (draft 3)," pp. 1–11, 2014. [Online]. Available: https://raw.githubusercontent.com/ipfs/papers/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf
[38] L. Veltri, S. Cirani, S. Busanelli, and G. Ferrari, "A novel batch-based group key management protocol applied to the Internet of Things," *Ad Hoc Netw.*, vol. 11, no. 8, pp. 2724–2737, 2013.
[39] S. H. Islam, M. S. Obaidat, P. Vijayakumar, E. Abdulhay, F. Li, and M. Reddy, "A robust and efficient password-based conditional privacy preserving authentication and group-key agreement protocol for vanets," *Future Gener. Comput. Syst.*, vol. 84, pp. 216–227, 2018.
[40] Q. Zhang, Y. Gan, L. Liu, X. Wang, X. Luo, and Y. Li, "An authenticated asymmetric group key agreement based on attribute encryption," *J. Netw. Comput. Appl.*, vol. 123, pp. 1–10, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804518302704
[41] H. Yao and C. Wang, "A novel blockchain-based authenticated key exchange protocol and its applications," in *Proc. IEEE 3rd Int. Conf. Data Sci. Cyberspace*, 2018, pp. 609–614.
[42] Y. Hu, Y. Xiong, W. Huang, and X. Bao, "Keychain: Blockchain-based key distribution," in *Proc. 4th Int. Conf. Big Data Comput. Commun.*, 2018, pp. 126–131.
[43] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "Ethdkg: Distributed key generation with ethereum smart contracts," *Cryptol. ePrint Arch.*, Tech. Rep. 2019/985, 2019, [Online]. Available: https://eprint.iacr.org/2019/985
[44] K. Bhargavan, R. Barnes, and E. Rescorla., "TreeKEM: Asynchronous decentralized key management for large dynamic groups a protocol proposal for messaging layer security (MLS)," *[Res. Rep.] Inria Paris⟨hal-02425247⟩*, pp. 1–20, 2018.
[45] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security analysis and improvements for the ietf mls standard for group messaging," in *Proc. 40th Annu. Int. Cryptol. Conf.*, 2020, pp. 248–277.
[46] J. Alwen, S. Coretti, Y. Dodis, and Yiannis, "Modular design of secure group messaging protocols and the security of MLS," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 1463–1483.
[47] J. Alwen *et al.*, "Grafting key trees: Efficient key management for overlapping groups," in *Proc. Theory Cryptogr. Conf.*, 2021, pp. 222–253.
[48] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert, "The messaging layer security (MLS) protocol," Internet Engineering Task Force, Internet-Draft draft-ietf-mls-protocol-09, Mar. 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-09
[49] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in ethereum," *IEEE Secur. Privacy Blockchain*, pp. 1–11, 2017. [Online]. Available: https://jbonneau.com/doc/BGB17-IEEESB-proof_of_delay_ethereum.pdf



Victor Youdom Kemmoe received the BS and MS degrees in computer science from Kennesaw State University, GA, USA, in 2018 and 2020, respectively. Currently, he is working toward the PhD degree in computer science with Brown University. From August 2020 to July 2021, he was a research assistant with the Information and Intelligent Security Laboratory at Kennesaw State University. His areas of research are cryptography and distributed computing.

Yongseok Kwon received the BE degree in computer science and engineering from Hanyang University, South Korea, in 2019. He is currently working toward the MS-leading-to-PhD degree in computer science and engineering with Hanyang University, South Korea. Since 2019, he has been with the Computer Science and Engineering, Hanyang University of Engineering, South Korea. His research interests include AI security, applied cryptography, and information security and privacy.

Rasheed Hussain (Senior Member, IEEE) received the BS engineering degree in computer software engineering from the University of Engineering and Technology, Peshawar, Pakistan, in 2007, and the MS and PhD degrees in computer science and engineering from Hanyang University, South Korea, in 2010 and 2015, respectively. He works as a senior lecturer with the Smart Internet Lab and Bristol Digital Futures Institute (BDFI), University of Bristol, U.K.. He worked as a postdoctoral fellow with Hanyang University, South Korea from March 2015 to August 2015, and as a guest researcher and consultant with the University of Amsterdam (UvA) from September 2015 till May 2016. He also worked as assistant professor and associate professor, head of MS program in Security and Network Engineering (SNE), and the head of the Networks and Blockchain Lab, Innopolis University, Russia from June 2016 till December 2021. He serves as an editorial board member for various journals including *IEEE Communications Surveys & Tutorials*, *IEEE Access*, *IEEE Internet Initiative, Internet Technology Letters*, *Wiley*, and serves as reviewer for most of the IEEE transactions, Springer and Elsevier Journals. He also serves as a technical program committee member of various conferences such as IEEE VTC, IEEE VNC, IEEE Globecom, IEEE ICCVE, and so on. He is a certified trainer for Instructional Skills Workshop (ISW). Furthermore, he is a ACM distinguished speaker. His research interests include information security and privacy and particularly IoT security, digital twins security, role of AI in cybersecurity, eXplainable AI, fairness in AI, future internet architecture, and blockchain.

Sunghyun Cho received the BS, MS, and PhD degrees in computer science and engineering from Hanyang University, Korea, in 1995, 1997, and 2001, respectively. From 2001 to 2006, he was with Samsung Advanced Institute of Technology, and with Telecommunication R&D Center of Samsung Electronics, where he has been engaged in the design and standardization of MAC and network layers of WiBro/WiMAX and 4G-LTE systems. From 2006 to 2008, he was a postdoctoral visiting scholar with the Department of Electrical Engineering, Stanford University. He is currently a professor with the Department of Computer Science and Engineering, Hanyang University.

Junggab Son (Senior Member, IEEE) received the BSE degree in computer science and engineering from Hanyang University, Ansan, South Korea in 2009, and the PhD degree in computer science and engineering from Hanyang University, Seoul, South Korea in 2014. From 2014 to 2016, he was a postdoctoral research associate with the Department of Math and Physics, North Carolina Central University. From 2016 to 2018, he was a research fellow and a limited-term assistant professor with Kennesaw State University. From 2018 to 2022, he was an assistant professor of computer science and a director of Information and Intelligent Security (IIS) Laboratory, Kennesaw State University. Currently, he is an assistant professor of computer science with the University of Nevada, Las Vegas. His research interests include applied cryptography, privacy preservation, blockchain and smart contract, malware detection, and security/privacy issues in artificial intelligent algorithms.