A BLOCKCHAIN BASED GROUP KEY AGREEMENT PROTOCOL (B-GKAP)

by

Yaşar Berkay Taçyıldız

B.S., Computer Engineering, Yıldız Technical University, 2015

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2020

ACKNOWLEDGEMENTS

For their supports and motivations during my thesis work, I would like to thank first to my family and friends. Especially to Meltem Saman, who has consistently encouraged me to complete this work with a success.

I would like to express my great appreciation to Dr. Orhan Ermiş and Dr. Gürkan Gür for their valuable and constructive suggestions during the development of this work. They have spent a vast amount of time to guide me to produce a better study.

Moreover, I would like to offer my deep gratitude to my thesis supervisor Prof. Fatih Alagöz, who has given me the chance for starting to this program and guided me to be prepared for the scientific research.

Finally, I would like to thank my friends at SATLAB, for their inspirational talks and supports during this period.

ABSTRACT

A BLOCKCHAIN BASED GROUP KEY AGREEMENT PROTOCOL (B-GKAP)

Group key agreement protocols are crucial in the case of multiple parties agreeing on a common key without a centralized entity. However, the decentralized characteristic of these protocols causes performance challenges where parties need to communicate and verify other participants in the group. To overcome this issue, we propose a new approach to the group key agreement protocols by utilizing Hyperledger Fabric framework as a blockchain platform. To this end, we migrate the communication and verification overhead of the group key agreement participants to the blockchain network in our developed scheme. This paradigm allows a flexible group key agreement protocol that considers resource-constrained entities and trade-offs regarding distributed computation. According to our performance analysis, participants with low computing resources can efficiently utilize our protocol. In addition, the secret parameters of the participants are distributed among the isolated participants that constitute the blockchain network. Thus, the only way for the network participants to compute group keys is to collude maliciously. Furthermore, we have demonstrated that our protocol has the same security features as other comparable protocols in the literature.

ÖZET

BLOK ZİNCİRİ TABANLI BİR GRUP ANAHTAR ANLAŞMASI PROTOKOLÜ

Birden fazla tarafın merkezi bir varlık olmadan ortak bir anahtar üzerinde anlaştığı durumlarda, grup anahtar anlaşması protokolleri çok önemlidir. Bu protokollerin merkezi bir sisteme ihtiyaç duymama özelliği, katılımcıların gruptaki diğer katılımcılarla iletişim kurması ve doğrulaması gereken yerlerde performans sorunlarına neden olur. Bu sorunun üstesinden gelebilmek için, bir block zinciri platformu olan Hyperledger Fabric çözümünü kullanarak grup anahtar anlaşması protokollerine yeni bir yaklaşım öneriyoruz. Bu amaçla, çalışmamızda grup anahtar anlaşması katılımcılarının iletişim ve doğrulama yükünü blok zincir ağına taşıyoruz. Performans analizimize göre, limitli bilgi işlem kaynaklarına sahip katılımcılar protokolümüzü verimli bir şekilde kullanabilirler. Ayrıca, katılımcıların gizli parametreleri, blokzincir ağı üzerinde oluşturmuş olduğumuz birbirinden ayrık katılımcılar arasında dağıtılmaktadır. Böylece, ağ üzerindeki katılımcıların grup anahtarlarını üretmesinin önüne geçmekteyiz. Ek olarak, önerdiğimiz protokolün literatürde sunulan diğer protokoller ile aynı güvenlik özelliklerini sağladığını çalışmamızda gösterdik.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

CK_i	Secret Key of i_{th} participant
M	Network Participant Count
N	Participant Count
s, e	Schnorr Signature products
t_i	Temporary secret key of i_{th} participant
U_i	i_{th} Participant
$\mathcal{U}$	Participant list
$\mathcal{U}_{participant}$	Participant group
$\mathcal{U}_{network}$	Network Participant group
x_i, y_i	Respectively private and public keys of participant U_i
ω_i	i_{th} participant temporary public key

LIST OF ACRONYMS/ABBREVIATIONS

BFT	Byzantine Fault Tolerant
BAN	Body Area Networks
B-GKAP	A Blockchain Based Group Key Agreement Protocol
CA	Certification Authority
CFT	Crash Fault Tolerant
DDHP	Decisional Diffie-Hellman Problem
DoS	Denial of Service
GKA	Group Key Agreement
GKAP	Group Key Agreement Protocol
HBC	Honest But Curious
HDD	Hard Disk
HF	Hyperledger Fabric
HTTP	Hyper Text Transfer Protocol
IoT	Internet of Things
KAP-PBC	A Key Agreement Protocol with Partial Backward Confidentiality
KB	Kilobyte
MB	Megabyte
ms	Millisecond
MSP	Membership Service Provider
PKI	Public Key Infrastructure
PoS	Proof of Stake
PoW	Proof of Work
SDK	Software Development Kit
SDN	Software Defined Networking
sec	Second
TB	Terrabyte
TEE	Trusted Execution Environment
TLS	Transport Layer Security

VANET	Vehicular Ad-hoc Networks
WSN	Wireless Sensor Network

1. INTRODUCTION

The digitalization of human activities and daily life has become a reality via the emergence of more prevalent and high-performance communications and networking. With the help of more capable network technologies/standards such as 5G and Software-Defined Networking (SDN), innovative and collective solutions which consist of different type of machines are now much more feasible. Although some of these systems require ultra-reliable and real-time connectivity such as telesurgery or industrial networks, several systems require a dynamic environment (e.g. connected vehicles) where interaction between the network entities changes frequently. Moreover, there are some requirements where devices with low computing power and limited energy resources should operate seamlessly and efficiently such as wireless sensor networks (WSN). In such systems, while the number of connected devices increases rapidly, decentralized and efficient secure communication frameworks are necessary to meet the demand.

As a secure communication facilitator, group key agreement (GKA) protocols where participants can agree on a common secret key in the insecure channel have gained significant importance. Starting with Diffie-Hellman [3], where two parties can agree on a secret key, several protocols have been developed which enable multiple parties to agree on a common key [4–6]. In these protocols, participants share key agreement parameters with each other without a single trusted entity. There are several factors which affect the performance of these protocols, the first one is the way of broadcasting these key agreement parameters and the second one is validating the identity of participants via verification of received parameters. To perform better in parameter distribution and verification stages, cluster-based approaches [6–8] and tree-based methods [5, 9] are introduced.

As a decentralized computing platform, Blockchain technology has recently emerged starting with ‘Bitcoin’ [1] as a monetary system based on cryptocurrency. The technology is later decoupled from cryptocurrencies first in Hyperledger Fabric [10]. Hyperledger Fabric is a generic decentralized application development platform, where transactions history is shared among peers as computation nodes in the network.

In our study, we propose a group key agreement protocol called B-GKAP which is an improved version of KAP-PBC [11] protocol based on the Hyperledger Fabric platform to improve key computation performance while keeping important security properties of known group key agreement protocols.

Our main contributions in this thesis are as below:

- To reduce the number of parameter transmission in the protocol, in our proposal, group key agreement participants communicate with the blockchain network instead of communicating with each other. In this way, the number of network transmissions is decreased significantly.
- We employ a blockchain application in Hyperledger Fabric, to perform verification of group key agreement parameters. Therefore, instead of each participant performs verification of every other participant, we migrate verification operations to the blockchain network.
- We propose two B-GKAP models which are B-GKAP1 and B-GKAP2. In B-GKAP1 we propose a single network entity as a network participant which follows the protocol except for the key computation. All the B-GKAP parameters are stored and verified in this entity. Additional to B-GKAP1, in B-GKAP2, we propose an organization entity as a network participant, in which we distribute sensitive GKA parameters of the participants between these entities. Therefore, even if a network participant has malicious behavior, all of the organizations need to collude to compute the group key.

The outline of this thesis is as follows: in Chapter 2, we discuss the group key agreement solutions and blockchain platforms in literature. Chapter 3 explains our proposed model B-GKAP in terms of its system model, protocol flow and functions. Later on, in Chapter 4, we prove that our protocol has the same security features with known group key agreement protocols. Additionally, we discuss the security of B-GKAP1 and B-GKAP2 models. Last but not least, in Chapter 5, we discuss performance analysis of our models in terms of communication and computation complexity analysis. Moreover, in this chapter, we present our simulation results in graphics. Finally, Chapter 6 composes up our findings in this study, and we discuss possible extensions.

2. LITERATURE OVERVIEW

In this chapter, we present an overview of the contributions in the literature which are related to our work. First of all, we go through the relevant group key agreement protocols. Then, we concisely describe blockchain technology and known applications to render a technical background. Lastly, we elaborate on the Hyperledger Fabric (HF) [12] platform which we have selected as the specific blockchain platform to implement and evaluate our protocol.

2.1. Group Key Agreement Protocols

In this section, first, we provide a mathematical background that is common in group key agreement protocols. Then, we describe the protocols in the literature according to their capabilities, security features, and usage areas.

2.1.1. Mathematical Background

Definition (Decisional Diffie-Hellman Problem (DDHP)): As stated in [13], p and q are large prime numbers, where $p = 2q+1$. Let $g \in Z_p^*$ be a generator of some cyclic group of quadratic residues G_p in Z_p^*. The two statements below are computationally indifferent from each other.

- $\langle g^a, g^b, g^{ab} \rangle$, where a and b are integers randomly and independently selected from Z_p.
- $\langle g^a, g^b, g^c \rangle$, where a, b and c are integers randomly and independently selected from Z_p.

Definition (Dynamic Groups): At time t_0, let the participant group be $\mathcal{U} = \{U_1, U_2, \ldots, U_N\}$. In time t_1, the group be $\mathcal{U}'$. In the case of where $t_0 \neq t_i$ and $\mathcal{U} \neq \mathcal{U}'$ conditions present, the group can be considered as dynamic group [11]. The list of dynamic group key agreement functions are defined as below:

(i) *Join:* Let k be number of participants that joins the group. From the group $\mathcal{U}$, participants $U_{n+1}, U_{n+2}, \ldots, U_{n+k}, \ldots, U_{n+k}$ joins $\mathcal{U}$ where $k \geq 1$.

(ii) *Leave:* Let k be number of participants that leaves the group. From the group $\mathcal{U}$, participants $U_i, U_{i+1}, \ldots, U_{i+k}$ leaves $\mathcal{U}$ where $0 < k < n$.

Definition (Backward Confidentiality): Previously computed group keys cannot be computed by the participants who joined the group [11].

Definition (Forward Confidentiality): Subsequently computed group keys cannot be computed by the participants who left the group [11].

Definition (Asymmetric Group Key Agreement): A group key agreement protocol has asymmetric property if decryption dk and encryption sk keys of the participants $U_i, U_j \in \mathcal{U}$ meet the equation $dk_i \neq dk_j \neq sk$ [14].

2.1.2. Static Group Key Agreement Protocols

Group key agreement (GKA) protocols allow multiple participants to agree on a secret key in an insecure environment [15]. The first key exchange protocol is Diffie-Hellman [3] where two participants agree on a common key. The security of Diffie-Hellman is based on discrete logarithm problem [16] which is still the main security structure of the modern GKA protocols.

The Diffie-Hellman protocol is represented in Figure 2.1 with two participants Alice and Bob:

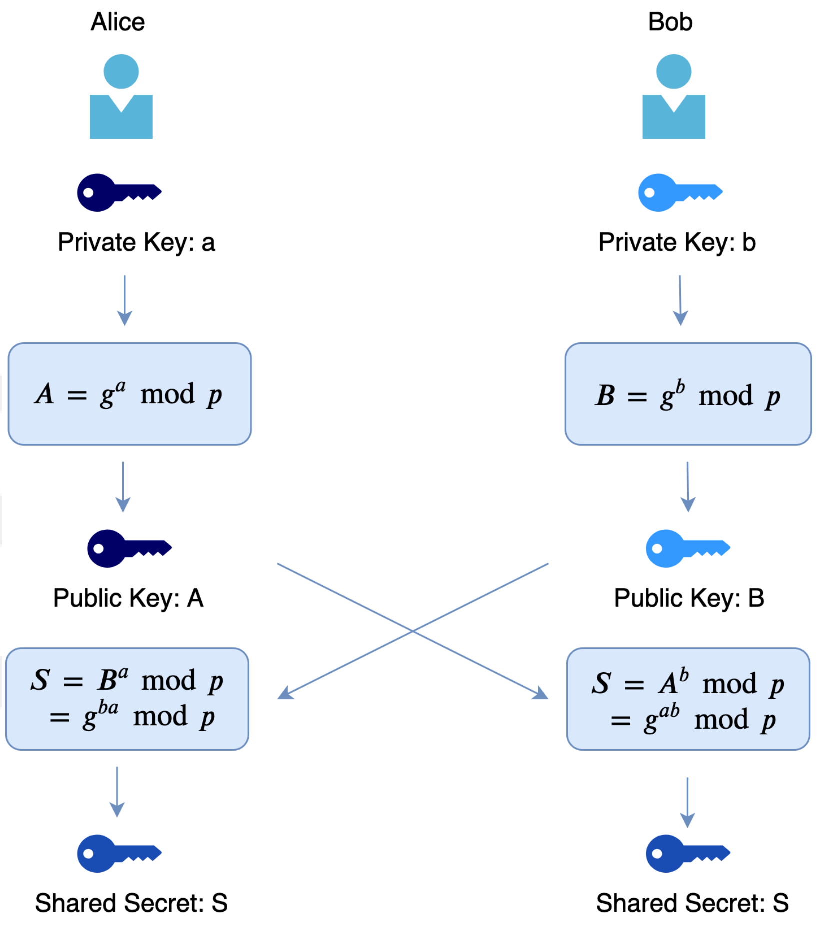


Figure 2.1. Diffie-Hellman Key Exchange

Although Diffie-Hellman protocol is a simple but powerful protocol for key agreement, the protocol is vulnerable to man-in-the-middle attacks. This issue is later solved by [17] by utilizing digital certificates to identify participants.

Ingemarsson et al. propose a conference key distribution system [18] where conference participants are arranged as a logical ring. Each participant computes a temporary variable and shares it with the next participant in the ring. For N participants, $N-1$ round is required for the key computation.

Burmester et al. have introduced an efficient group key agreement protocol [19] where only two rounds are required to compute the group key. In the first round, each participant produces a temporary parameter and broadcasts it. Next, each participant produces a second parameter based on the next and previous parameters of participants and broadcasts it. Finally, with the last parameter, participants can compute the group key.

2.1.3. Dynamic Group Key Agreement Protocols

The protocols mentioned before are all static group key agreement protocols. In static GKA protocols, when participants in the group change, the protocol needs to be started from the beginning. On the other hand, dynamic GKA protocols provide auxiliary operations (defined in Section 2.1.1) that require much less computation effort such as in [5, 6, 11, 20–23].

To increase efficiency in group key computation, tree-based group key agreement protocols are proposed in [5, 9]. Dutta et al. [5] propose a dynamic tree-based group key agreement protocol where participants are hierarchically positioned as a tree structure. In the lower leaves, subgroup of participants computes group key, and forwards to the participants at the upper leaves. And this process goes until the common group key is computed in the root of the tree.

Alternative approaches are introduced by utilizing cluster structures in [6, 8]. Ermis et al. have proposed a cluster-based dynamic group key protocol called GKAP-MANET [6]. In this protocol, the participants are grouped as a set of a clusters and each cluster has a cluster head. Two types of communication occur between participants; first, between participants in the same cluster and second, between the cluster heads. Therefore, the cluster key is firstly agreed with the participants in the same cluster while, later on, cluster heads agree on the common group key.

An asymmetric GKA protocol is proposed by Wu et al. in [14]. In this protocol, instead of participants agreeing on a symmetric group key, a public key is shared among the participants. This key is used to verify signatures, and encrypt transmitted messages. The shared public key corresponds to the private keys of the participants, which can only be generated by the owner. The participants use their private keys for decrypting and signing the messages. This protocol ensures key agreement in a single round.

2.1.4. Security Properties

In GKA protocols, the fault tolerance property is very crucial since it is necessary to detect and eliminate malicious participants from the key agreement group. In other words, even if there are malicious participants in the group, they should not be able to affect the key computation of honest participants. Early protocol examples with this property are [24–26].

In this regard, in Tseng's protocol [24], every participant keeps a verification matrix V_{ij}. After the secret key distribution step, each participant checks the signature of other participants. According to the result, the verification list is marked and submitted to other participants. Afterwards, in the fault detection step, participants re-validate the verification matrix and remove the faulty participants from the key agreement group. Finally, GKA protocol is started from scratch with the remaining participants.

Forward secrecy (also stated as Perfect forward secrecy) is also a substantial property that protects against the computation of group keys by malicious actors even if private keys are compromised. Forward secrecy is utilized in protocols presented in [4, 27, 28].

Dynamic group key operations in group key agreement protocols must provide forward and backward confidentiality properties defined in Section 2.1.1. Introduced by Ermis et al., KAP-PBC [11] protocol provides these properties within its dynamic operations. In join and leave operations, last participants in the group re-compute the GKA parameters. Therefore, joined participants cannot compute the former group keys, and leaving participants cannot generate the subsequent keys. Moreover, KAP-PBC provides 'Partial Backward Confidentiality' property, which enables the participants to compute the group keys just before joining the group.

2.1.5. Usage Areas

There is a correlation with usage areas of IoT systems and group key agreement protocols. For instance, ad hoc networks are good candidates to utilize efficient GKA protocols. In this area there are several proposals [6–8, 29], which are based on cluster structure. Since participants are grouped as clusters, the required group key computation effort is minimized in these smaller groups.

Wireless sensor networks (WSN) is also another convenient area to implement GKA protocols. In [30], Lu et al. propose a three-factor GKA protocol which utilizes mutual authentication with Elliptic Curve Cryptography. Moreover, Challa et al. introduce a GKA protocol [31] for Wireless Healthcare sensor networks. This protocol provides high security, low computation and communication costs which are suitable for healthcare applications.

Another GKA protocol is introduced by Tang et al. called PBAKA [32]. This protocol provides a solution for Body Area Networks (BANs) where multiple bio-sensors collect health-related information and provide intelligent health-care services. To solve authenticity, a control unit is used to initiate authentication via collected physiological features from the BAN sensors. These features are later utilized to negotiate session keys for the sensors. Another protocol introduced in this area is [33].

Vehicular Ad-hoc Networks (VANET) platform is another promising usage area for the group key agreement protocols. Islam et al. introduced a GKA protocol [34], which focuses on controlling city traffic via an efficient protocol. To provide efficiency, password-based authentication and group key generation are utilized instead of elliptic curve and bilinear-pairing which have significant computational cost.

2.2. Hyperledger Fabric Platform

In this section, we first briefly explain the blockchain technology and the most common applications. Subsequently, we account for the Hyperledger Fabric platform components and its transaction flow.

2.2.1. Background

The idea of the cryptographically secured chain of blocks was first introduced by Haber et al. [35]. Later on, a pseudonym called ‘Nakamoto Satoshi’ [1] introduced a new monetary system, namely ‘Bitcoin’, using blockchain technology in 2008. This technology has enabled trusted decentralized applications by removing the center of trust from the network. Blockchain networks consist of multiple peers that maintain synced transaction history called ledgers. In the ledger, transactions are chained and collected in transaction blocks. Additionally, each transaction block is bounded together as a hash chain.

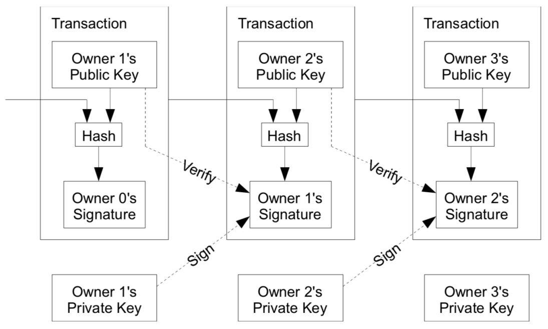


Figure 2.2. Chain of Transactions [1]

Figure 2.2 shows how transactions are stored in a transaction block. In this depiction, each transaction contains the owner’s public key, hash, and signature which is signed by the owner of the previous transaction. Hence, the public key in the transaction can verify the following transaction.

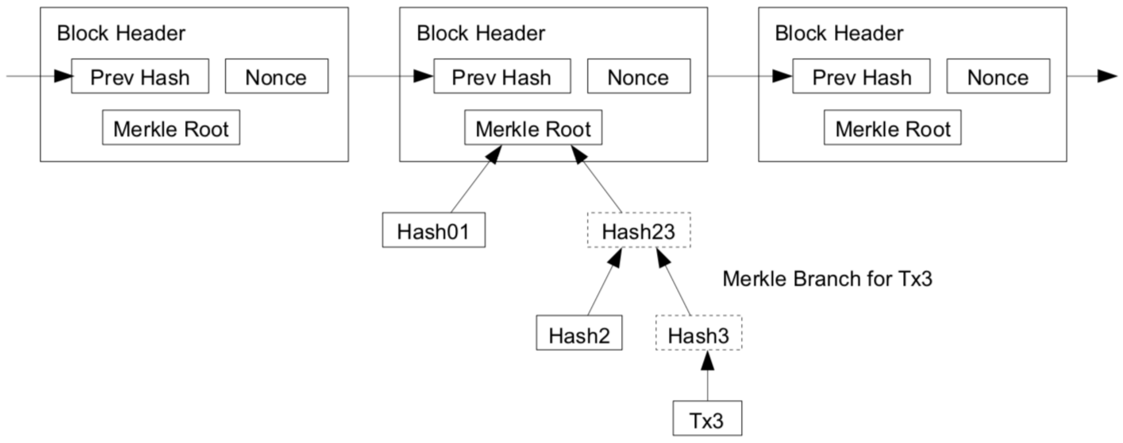


Figure 2.3. Chain of Transaction Blocks [1]

In Figure 2.3, the transaction block structure is represented. Each block contains the hash of the previous block, nonce, and Merkle root. To keep track of the integrity of stored transactions in the block, Merkle tree structure in [36] is utilized. In this way, any transaction in the block can be validated efficiently. Peers in blockchain need to solve proof of work (PoW) challenge to validate and add new transaction blocks. PoW is a consensus protocol [37] which ensures the immutability of the ledger. Since each block is bounded to each other by hashes, proof of work should be performed for all following blocks to modify a transaction in a block. Therefore, altering a transaction becomes harder when new blocks are added to the ledger.

After Bitcoin, blockchain technology has been used in many other projects. Among these projects, Ethereum [38] is worth mentioning. Founded by Vitalik Buterin in 2013, Ethereum introduced a novel method for running applications in its network which is called smart contract. Smart contracts allow us to run business logic in a decentralized way such that if any node in the Ethereum network fails the application will still be operational. To run a smart contract in the network, the owner of the smart contract should pay ether (the currency of Ethereum) as a fee. In this way, only deterministic smart contracts can run in the network. As a consensus protocol, Ethereum utilizes proof of work, but with Ethereum 2.0 [39], proof of stake (PoS) [40] protocol will be utilized due to security vulnerabilities [41] and efficiency problems of PoW.

2.2.2. The Platform

Hyperledger Fabric (HF) platform [42] is originally contributed by IBM and Digital Asset, later made an open-source project under Linux Foundation Projects [43]. Unlike Bitcoin and Ethereum, Hyperledger Fabric does not rely on cryptocurrencies. This feature enables HF to be a truly decentralized application development environment. Additionally, both Bitcoin and Ethereum are public blockchain platforms where everyone can interact with the network. On the other hand, Hyperledger Fabric is a permissioned blockchain platform that only allows identified participants. Thus, with the identification of the network modules, Byzantine Fault Tolerant (BFT) [44–46] or Crash Fault Tolerant (CFT) [47] consensus protocols can be utilized.

Another important feature of HF is that most of the HF components are designed to be modular such as Membership Service Provider (MSP) and consensus protocol. The modular design of the HF platform is made possible by its novel execute-order-validate architecture as shown in Figure 2.5. In other applications [1, 38], order-execute architecture is utilized where transactions first ordered via a consensus protocol, later on, they are executed by all peers sequentially as represented in Figure 2.4. On the other hand, in Hyperledger Fabric, execution of the transactions is performed first to allow running non-deterministic applications and the ordering phase is separated from the validation step to isolate consensus logic from the peers. Therefore, the transactions can run in parallel without the necessity to keep the order. After the consensus is provided by ordering state, the final state of the transaction can be applied by all nodes individually.

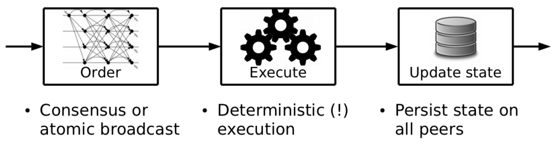

Order
Execute
Update state
• Consensus or atomic broadcast
• Deterministic (!) execution
• Persist state on all peers

Figure 2.4. Order-Execute Architecture [2]

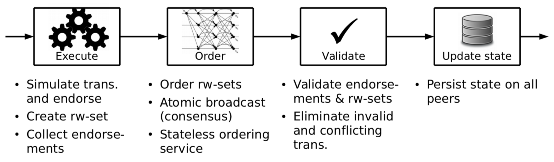

Execute
Order
Validate
Update state
• Simulate trans. and endorse
• Create rw-set
• Collect endorsements
• Order rw-sets
• Atomic broadcast (consensus)
• Stateless ordering service
• Validate endorsements & rw-sets
• Eliminate invalid and conflicting trans.
• Persist state on all peers

Figure 2.5. Execute-Order-Validate Architecture [2]

2.2.3. Fabric Components

Hyperledger Fabric Platform consists of several entities which are explained as below:

- *Membership Service Provider (MSP):* Membership Service Provider module is what makes Hyperledger Fabric is a permissioned blockchain platform. In Fabric, MSP maintains the identities of fabric nodes in the system including clients, peers and orderers. The interactions among the fabric nodes are performed by gRPC [48] and authenticated via mutual Transport Layer Security (TLS). MSP is an abstraction layer for a modular entity, which can be replaced with other identity providers. Hyperledger Fabric comes with Fabric CA component as default MSP based on Public Key Infrastructure (PKI) with digital signatures.
- *Fabric Peer:* Peers in Hyperledger Fabric hold and maintain the blockchain ledger. Prior to the commitment of transactions, Fabric peers simulate these transactions on its copy of the ledger. In addition to the blockchain ledger, peers also keep the latest status of the ledger as either in Go LevelDB [49] or in Apache CouchDB [50]. In Fabric, all peers validate the transaction, but only limited number of peers are chosen as endorser peers. Endorser peers manage the chaincode where transactions are simulated.
- *Fabric Chaincode (Smart Contract):* Chaincode is the replacement name of the smart contract in Hyperledger Fabric. Chaincode should be installed on peers, and instantiated to start the network. In Hyperledger Fabric, only the chaincode is permitted to modify ledger.
- *Fabric Orderer:* Orderer maintains a consensus mechanism such as PBFT and Kafka [51]. This entity decides if the endorsement policy requirements are completed based on simulation results of the peers. After the simulation step is completed, transactions

are collected as blockchain blocks. Finally the block sent back to peers and appended to their ledger. Orderer solves concurrency, double spending and many security related issues [41].

- *Organization:* In Hyperledger Fabric, organizations work together to set up common functionality into the platform. They agree on an identical chaincode to run their collective logic. Organizations in Fabric are partially trusted into each other, hence the platform enforces to authorize common functionality by utilizing endorsement policies.
- *Channels:* Channels are isolated instances of Hyperledger Fabric where ledger and transactions are separated from each other.
- *Endorsement Policy:* Endorsement policy defines a set of requirements that should be performed in the transaction simulation (execution) phase to proceed ordering state. For instance, after the transaction proposal is sent to multiple peers, a client collects the endorsement results of the peers. Afterward, a client packs the endorsement results as invocation request and submits to the Fabric Orderer. The orderer than verifies the endorsement policy. If a specific set of endorsers are not satisfied, the transaction proposal is denied. Endorsement policy can be a logical collection of organization entities, for instance, $AND(Org1, OR(Org2, Org3))$ where Org1 and one of Org2 and Org3 should verify the transaction request.

2.2.4. Fabric Transaction Flow

Figure 2.6 illustrates a transaction flow in Hyperledger Fabric platform.

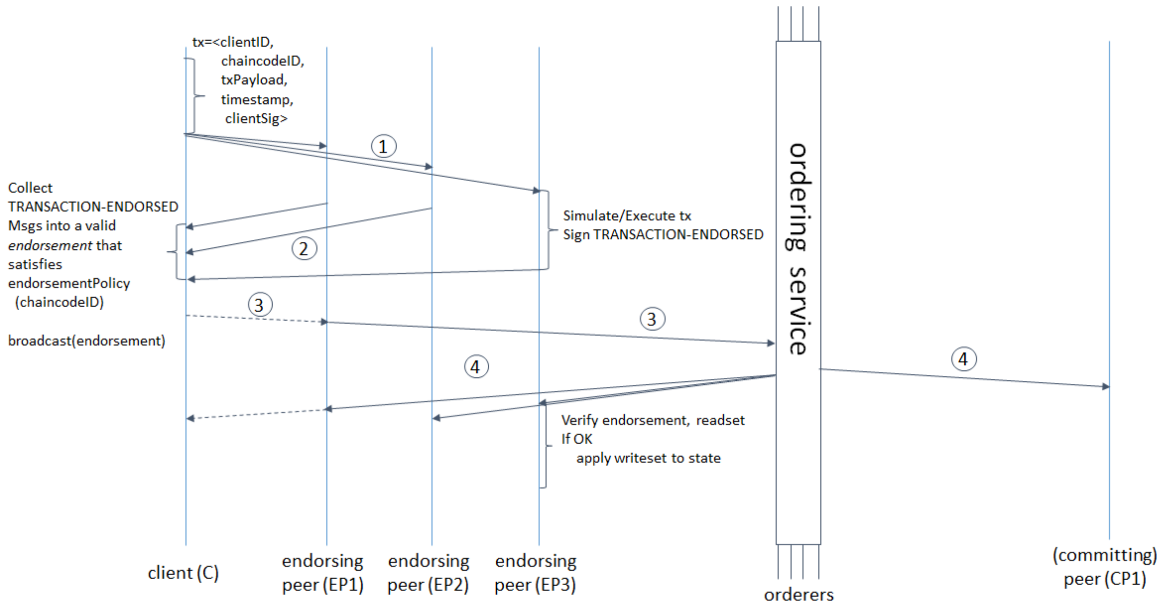

tx=<clientID, chaincodeID, txPayload, timestamp, clientSig>
1
Collect TRANSACTION-ENDORSED Msgs into a valid endorsement that satisfies endorsementPolicy (chaincodeID)
2
Simulate/Execute tx
Sign TRANSACTION-ENDORSED
3
broadcast(endorsement)
3
ordering service
4
4
Verify endorsement, readset
If OK
apply writeset to state
client (C)
endorsing peer (EP1)
endorsing peer (EP2)
endorsing peer (EP3)
orderers
(committing) peer (CP1)

Figure 2.6. Hyperledger Fabric Transaction Flow [2]

According to the figure, first, a client submits the transaction request to the endorser peers (EP1, EP2, and EP3). Later on, the peers simulate the transaction request in parallel via deployed chaincodes. After the simulation step, endorser peers generate the read-write set and signs the endorsed transaction. Next, the client collects signed endorsements, create an actual transaction and sends it to the ordering service. The ordering service validates the endorsements via endorsement policy, orders collected transactions in a batch, signs the batch and disseminates to all peers via gossip protocol [52]. Finally, the peers validate endorsements together with the signature of the orderer and apply transaction output to their ledgers.

3. A BLOCKCHAIN BASED GROUP KEY AGREEMENT PROTOCOL (B-GKAP)

In this chapter, we propose Blockchain Based Group Key Agreement Protocol (B-GKAP) which is deployed on Hyperledger Fabric (HF) [12] as a blockchain platform. In the first section, we provide a system overview that explains the positioning of the HF components. Afterward, first, we present data flow between B-GKAP network and participants, and then we give more details about B-GKAP.

3.1. System Overview

B-GKAP is based on the Key Agreement Protocol with Partial Backward Confidentiality, called KAP-PBC [11] but extends and improves it with blockchain aspect. B-GKAP utilizes Hyperledger Fabric platform in order to increase the key computation performance. Moreover, in B-GKAP, we migrate the communication among participants to communication between participants and the network which, therefore, reduces the communication cost during the key computation in terms of the length of the transmitted messages. Additionally, to verify the variables of the participants, we utilize HF chaincodes. When a variable is received as an invoke request by the network, the chaincode first performs the verification operation depending on the variable type. Then if the verification succeeds, the chaincode approves the operation.

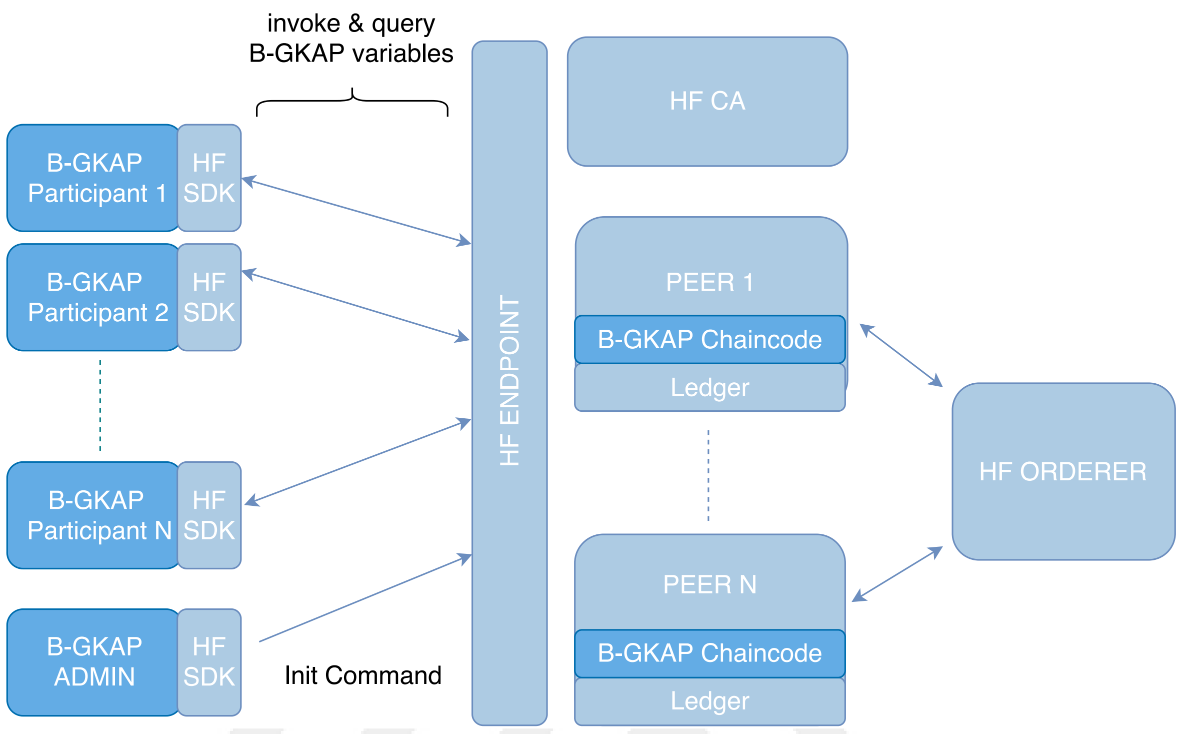


Figure 3.1. B-GKAP System Diagram.

The overview of B-GKAP is shown in Figure 3.1, which consists of the following main components:

(i) B-GKAP participants are the entities which compute the group key.

(ii) B-GKAP admin sends initialization command to the network to start up the Hyperledger Fabric platform and setup initial variables. Both B-GKAP participants and admin use HF Software Development Kit (SDK) which enables them to communicate with the network.

(iii) The peers are responsible for simulating incoming transactions by utilizing B-GKAP chaincode. Additionally, each peer maintains a blockchain ledger and latest ledger state. We have utilized HF ledger to store B-GKAP parameters.

(iv) HF Endpoint is the interaction point for B-GKAP participants. It can represent more than one peer.

(v) B-GKAP Chaincode handles all ledger read-write requests of the participants. The chaincode performs all the necessary verification operations, and if the request is valid, it produces read-write set.

(vi) HF Orderer performs the ordering of the produced transaction outputs sets as a block of transactions and it disseminates to all HF peers, then peers update their ledger states.

(vii) HF Certification Authority (CA) maintains the identities of the HF components and the B-GKAP participants.

3.2. General Definitions

In this section, we provide definitions that are used in B-GKAP based on [11, 13, 53].

Definition (Participants): Participant entity definition:

- Each participant is an entity and is represented as U_i.
- The participant list is represented as $\mathcal{U} = \langle U_1, U_2, \ldots, U_{N+M} \rangle$ which consist of two subgroups, $|\mathcal{U}_{network}| = M$, and $|\mathcal{U}_{participant}| = N$.

$$\mathcal{U} = \mathcal{U}_{participant} \cup \mathcal{U}_{network}$$

- The participant group $\mathcal{U}_{participant}$ is circular so that $U_{N+i} = U_i$ for some positive $1 \leqslant i \leqslant N$. The order of the participants is known by each participant.

Definition (Public Parameters): B-GKAP uses the following public parameters based on the definitions in [53]:

- $p = 2q + 1$, where both p and q are large prime numbers.
- g is a generator for $G_q = \{i^2 | i \in Z_p^*\}$, where G_q is a cyclic subgroup of quadratic residues in Z_p^*.
- T is the time-stamp against replay attack.

Definition (Long-term Public Private Key Pair): The protocol uses the following long-term key definitions based on [11]. Each entity in B-GKAP holds this key pair.

- $x_i \in Z_q^*$ is the private key and only the entity that holds the key knows it. This key is never shared with other entities in the network.
- y_i is the public key where $y_i = g^{x_i} mod p$.

Since our solution is based on KAP-PBC in [11], we assume that long-term public keys of each participant are issued via Certification Authority (CA). Before transmission, each variable is signed with long-term private key. Thus, during signature verification stage, identities of the participants are verified. For the signature method, Schnorr Signature scheme is utilized [54].

Definition (Schnorr Signature Scheme): Using long term key pairs x_i and y_i: A message M can be signed as $e,s = SS(x_i, y_i, M)$ and the signature products e,s can be verified using $SV(y_i,(e,s),M) \overset{?}{=} True$. In these equations, SS stands for 'Schnorr Sign' and SV stands for 'Schnorr Verify'.

Definition (Ledger Functions): Each network participant U_i in $\mathcal{U}_{network}$ maintains its own blockchain ledger and has two functionalities called $readLedger(\cdot)$ and $writeLedger(\cdot)$. A variable x can be written to the ledger via $writeLedger(x)$, and read from the ledger via $x = readLedger()$.

3.3. B-GKAP Protocol

In this section, B-GKAP protocol is expressed briefly via the activity diagram. Later on, B-GKAP1 and B-GKAP2 models are accounted in detail.

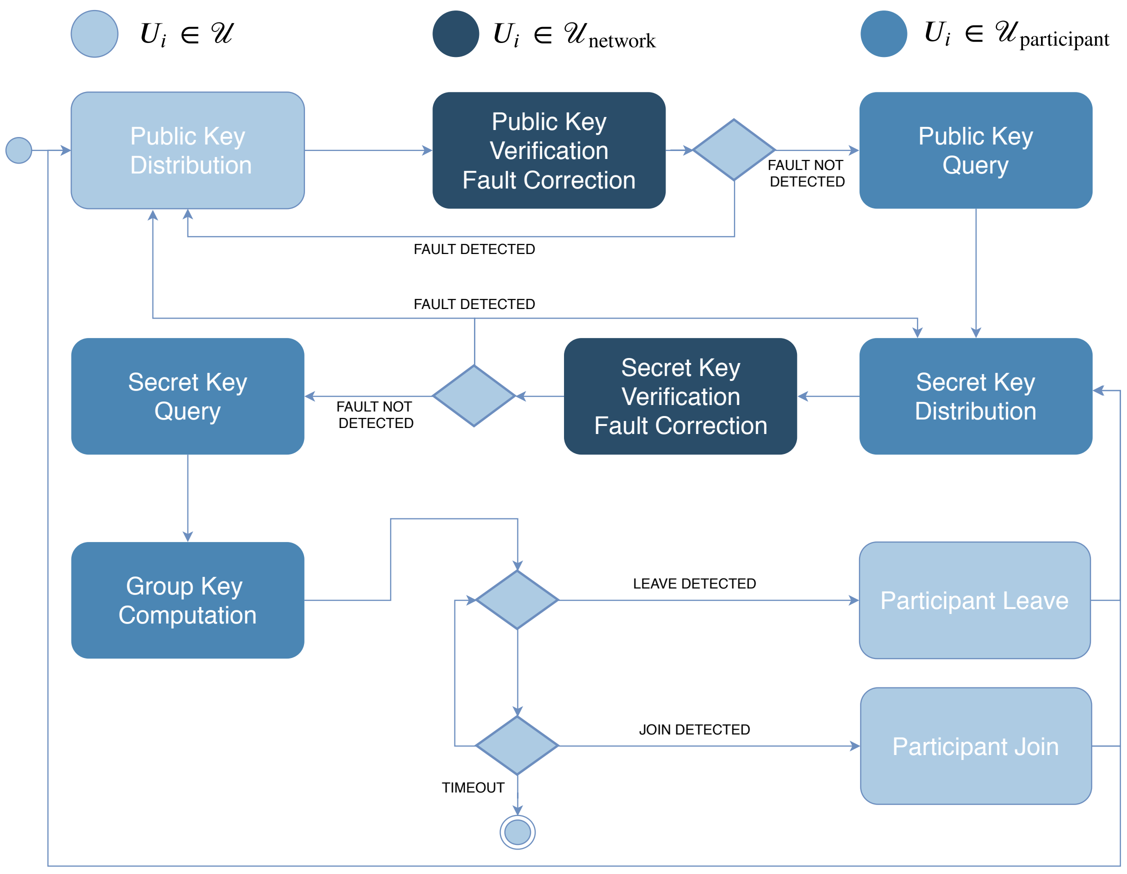


Figure 3.2. UML Activity Diagram of B-GKAP

According to the Figure 3.2, first, each participant $U_i \in \mathcal{U}$ executes Public Key Distribution Step to distribute temporary public keys. Then, each network participant executes Public Key Verification and Fault Correction Steps to remove dishonest participants from the group. Later on, remaining honest participants execute Public Key Query to fetch the temporary public key of the next participant in the group. Once this step is completed, each participant executes the Secret Key Distribution Step to send the secret keys to the network participants. Afterward, network participants perform Secret Key Verification and Fault Correction steps to exclude malicious participants from the group. Finally, each participant performs Secret Key Query and Group Key Computation Steps respectively to compute the common group key. Besides, when a new participant joins the group or leaves the group, Participant Join or Participant Leave Steps can be executed.

3.3.1. B-GKAP1 Model

As represented in Figure 3.3, this model is the basis of B-GKAP implementation. It has multiple peers that are bounded to a single organization and a single channel.

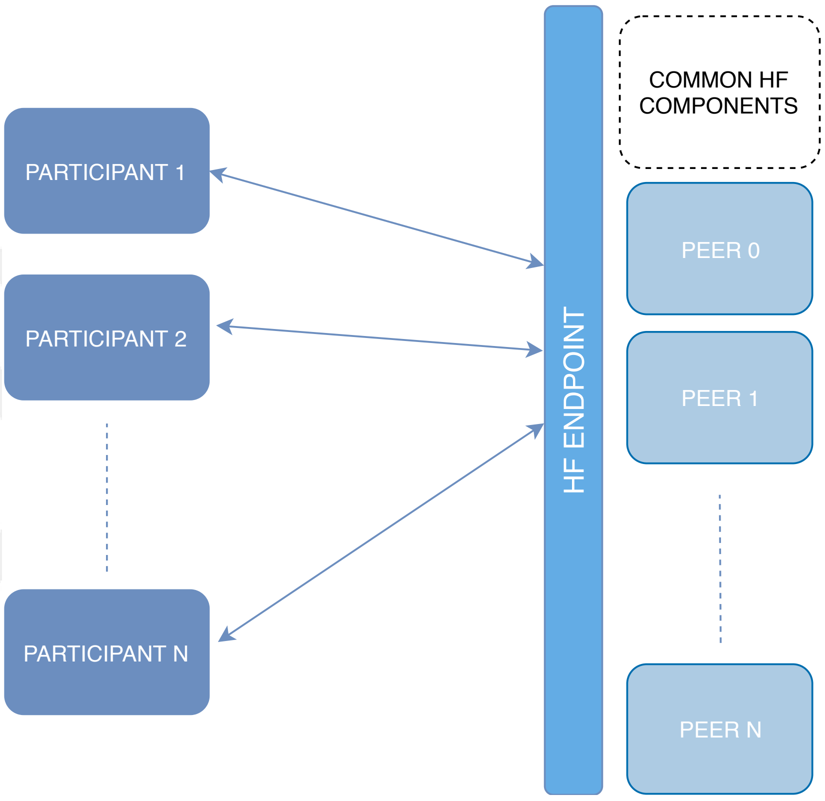


Figure 3.3. B-GKAP1 System Model.

In this model, there is only one network participant $U_{network}$ that does not involve directly into the group key computation stage. Instead, the network produces its B-GKAP participant variables except for the secret key. Therefore, the network can verify the temporary public and secret keys of the participants.

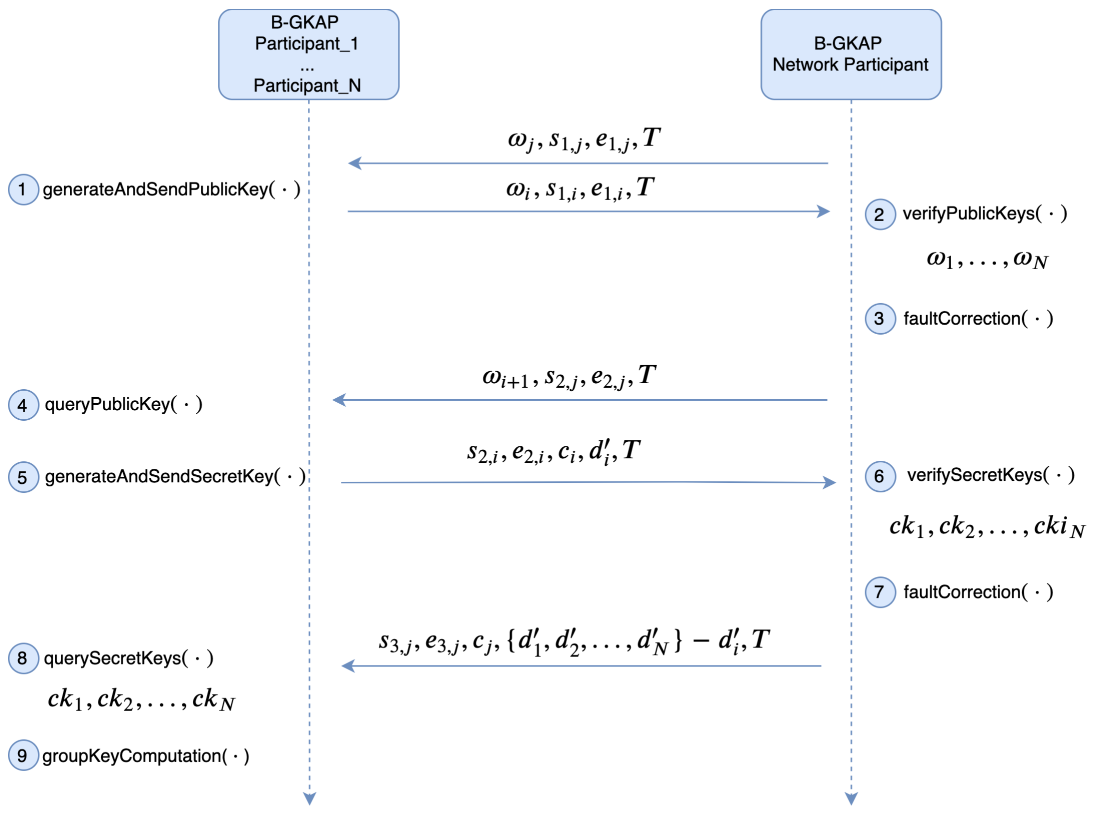

B-GKAP Participant_1 ... Participant_N
B-GKAP Network Participant
$\omega_j, s_{1,j}, e_{1,j}, T$
1 generateAndSendPublicKey(·)
$\omega_i, s_{1,i}, e_{1,i}, T$
2 verifyPublicKeys(·)
$\omega_1, \ldots, \omega_N$
3 faultCorrection(·)
$\omega_{i+1}, s_{2,j}, e_{2,j}, T$
4 queryPublicKey(·)
$s_{2,i}, e_{2,i}, c_i, d'_i, T$
5 generateAndSendSecretKey(·)
6 verifySecretKeys(·)
$ck_1, ck_2, \ldots, cki_N$
7 faultCorrection(·)
$s_{3,j}, e_{3,j}, c_j, \{d'_1, d'_2, \ldots, d'_N\} - d'_i, T$
8 querySecretKeys(·)
$ck_1, ck_2, \ldots, ck_N$
9 groupKeyComputation(·)

Figure 3.4. B-GKAP1 Protocol

Figure 3.4 indicates protocol steps and transmitted variables between B-GKAP network participants and participants in each step.

Public Key Distribution: In this step, each participant $U_i \in \mathcal{U}$ executes following function in Figure 3.5 to distribute temporary public keys.

```
$generateAndSendPublicKey(\cdot)$:
  1: randomly select $t \in Z_q^*$
  2: $\omega = g^t mod p$
  3: Sign $\omega$, $e, s = SS(y, x, \omega)$
  4: Send the message $M = \{\omega, e, s, T\}$
```

Figure 3.5. Public Key Distribution Function.

Public Key Verification: The network participant $U_{network}$ executes following function in Figure 3.6 to verify temporary public key of each participant $U_i \in \mathcal{U}$. If the participant is verified the key is written to the network ledger.

```
$verifyPublicKeys(\cdot)$:
1: for all $U_i \in \mathcal{U}_{participant}$ do
2:     if $SV(y_i, e_{1,i}, s_{1,i}, \omega_i))$ holds then
3:         $writeLedger(\omega_i)$
4:     end if
5: end for
```

Figure 3.6. Public Key Verification Function.

Fault Correction: The network participant $U_{network}$ removes any participant $U_i \in \mathcal{U}_{participant}$ whose temporary public key or secret key verification fails from the participant list with the function in Figure 3.7.

```
$faultCorrection(\cdot)$:
1: for all $U_i \in \mathcal{U}_{participant}$ do
2:     if $U_i$ is faulty then
3:         $U_i$ is removed from the participant group, $\mathcal{U}' = \mathcal{U} - U_i$
4:         For the participant $U_i$, $participantLeave(\cdot)$ is executed
5:     end if
6: end for
```

Figure 3.7. Fault Correction Function.

Public Key Query: After the fault correction step, each participant $U_i \in \mathcal{U}_{participant}$ executes the following function in Figure 3.8 to query temporary public key of the next participant (U_{i+1}).

```
$queryPublicKey(\cdot)$:
1: if $U_{network} \in \mathcal{U}_{network}$ then
2:     $\omega_{i+1} = readLedger()$
3:     Sign temporary public key of $U_{i+1}$, $e_n, s_n = SS(y_n, x_n, \omega_{i+1})$
4:     Send the message $M = (\omega_{i+1}, e_n, s_n, T)$ to the participant $U_i$
5: end if
6: if $U_i \in \mathcal{U}_{participant}$ then
7:     Receive the message $M$
8:     Check timestamp $T$
9:     Verify signature of the network participant, $SV(y_n, e_n, s_n, \omega_{i+1}))$
10: end if
```

Figure 3.8. Public Key Query Function.

Secret Key Distribution: Based on the received temporary public key, each participant U_i in $\mathcal{U}_{participant}$ executes function in Figure 3.9 to generate and send secret key (CK_i). The secret key is encrypted with public key of the network (ω_n) and signed with long term private key of the participant ($SS(x_i, y_i, CK_i)$). Generated values are sent to the network participant.

```
$generateAndSendSecretKey(\cdot)$:
1: $CK_i = \omega_{(i+1)}^{t_i} \mod p = g^{t_i t_{i+1}} \mod p$
2: Randomly select an integer $a \in Z_q^*$
3: $k = (\omega_n^a mod p) \mod q$
4: Randomly selects a line $L(x)$;
   $L(x) = xc_i + CK_i$ mod $q$,
   $c_i = g^a \mod p$.
5: $d_i = L(k) \mod q$
6: $d_i' = k \oplus d_i$
7: $e_{2,i}, s_{2,i} = SS(y_i, x_i, CK_i)$
8: $U_i$ sends the $M$ to $U_{network}$, $M = \{s_{2,i}, e_{2,i}, c_i, d_i', T\}$
```

Figure 3.9. Secret Key Distribution Function.

Secret Key Verification: The network participant executes function in Figure 3.10 to verify secret key of each participant $U_i \in \mathcal{U}_{participant}$. If the participant is verified, the key is written to the network ledger.

```
$verifySecretKeys(\cdot)$:
1: for all $U_i \in \mathcal{U}_{participant}$ do
2:     Receive message $M = \{T, e_{2,i}, s_{2,i}, c_i, d'_i\}$
3:     Recover the sub-key $CK_i$ and checks time-stamp $T$.
4:     $k = (c_i^{t_n} \mod p) \mod q$
5:     $d_i = d'_i \oplus k$
6:     $CK_i = d_i - c_i * k \mod q$
7:     Check the signature of $U_i$.
8:     if $SV(y_i, e_{2,i}, s_{2,i}, CK_i)$ holds then
9:         $writeLedger(CK_i)$
10:    end if
11: end for
```

Figure 3.10. Secret Key Verification Function.

Secret Key Query: After the network participant $U_{network}$ performs fault correction, each participant $U_i \in \mathcal{U}_{participant}$ queries for secret keys as in Figure 3.11.

```
$querySecretKeys(\cdot)$:
1:  if $U_{network} \in \mathcal{U}_{network}$ then
2:      Randomly select an integer $a \in Z_q^*$
3:      $k_i = (\omega_i^a mod p) mod q$
4:      for all $U_j \in \mathcal{U}_{participant} - U_i$ do
5:          Randomly select a line $L(x)$;
            $L(x) = xc_n + CK_j \mod q$,
            $c_n = g^a \mod p$.
6:          $d_j = L(k_i) \mod q$
7:          $d'_j = k_i \oplus d$
8:      end for
9:      Sign $CK_{1...N}$, $e_{2,n}, s_{2,n} = SS(y_n, x_n, CK_{1...N})$
10:     Send message to $U_i$: $M = \{s_{2,n}, e_{2,n}, c_n, \{d'_1, d'_2, \ldots, d'_N\}, T\}$
11: end if
12: if $U_i \in \mathcal{U}_{participant}$ then
13:     $k_i = (c_n^{t_i} \mod p) \mod q$
14:     for all $U_j \in \mathcal{U} - U_i$ do
15:         $d_j = d'_j \oplus k_i$
16:         $CK_j = d_j - c_n * k_i \mod q$
17:     end for
18:     Check timestamp $T$
19:     Check the signature of $U_{network}$: $SV(y_n, s_{2,n}, e_{2,n}, CK_{1...N})$
20: end if
```

Figure 3.11. Secret Key Query Function.

Group Key Computation: Each participant by $U_i \in \mathcal{U}_{participant}$ computes the group key as in Figure 3.12.

```
$groupKeyComputation(\cdot)$:
1: for all $U_k \in \mathcal{U}$ do
2:     $CK = ((CK_1' CK_2' \cdots CK_n')mod p)mod q = (g^{t_1 t_2 + t_2 t_3 + \ldots + t_{n-1} t_n + t_n t_1} mod p)mod q$
3: end for
```

Figure 3.12. Group Key Computation Function.

Participant Join: When participant U_i is joined to key agreement group $U_{participant} = U_1, U_2$-$, ..., U_N$, the function in Figure 3.13 is called.

```
$participantJoin(\cdot)$:
1: if $U_i \in \{U_1, U_2, \ldots, U_{N-1}\}$ then
2:     $U_i$ re-distributes $generateAndSendPublicKey(\cdot)$, $generateAndSendSecretKey(\cdot)$
       function outputs.
3: end if
4: if $U_i \in \{U_N, U_{N+1}, \ldots, U_{N+K}\}$ then
5:     $U_i$ performs $generateAndSendPublicKey(\cdot)$ function.
6:     Network participants perform $faultCorrection(\cdot)$
7:     $U_{N-1}$ performs $generateAndSendSecretKey(\cdot)$ function.
8:     Network participants perform $faultCorrection(\cdot)$
9: end if
10: for all $U_i \in \{U_1, U_2, \ldots, U_{N+K}\}$ do
11:     $U_i$ performs $querySecretKeys(\cdot)$ and $groupKeyComputation(\cdot)$ functions.
12: end for
```

Figure 3.13. Participant Join Function.

Participant Leave: When a set of participants $U_i, U_{i+1}, ..., U_{i+N}$ leaves the group $U_{participant}$, the function in Figure 3.14 is executed. Let the group with leaving participants be $U'_{participant}$.

```
$participantLeave(\cdot)$:
1:  if $|\mathcal{U}_{participant}| - |\mathcal{U}'_{participant}| < 2$ then
2:      The group key computation is terminated
3:  end if
4:  for each leaving participant $U_j \in \mathcal{U}'_{participant}$ do
5:      non-leaving participant(s) $U_{j-1} \in \mathcal{U}_{participant} - \mathcal{U}'_{participant}$, performs
        $generateAndSendPublicKey(\cdot)$.
6:      Network participants perform $faultCorrection(\cdot)$
7:      $U_{j-1}$ and $U_{j-2} \in \mathcal{U}_{participant} - \mathcal{U}'_{participant}$ perform $generateAndSendSecretKey(\cdot)$.
8:      Network participants perform $faultCorrection(\cdot)$
9:  end for
10: for all $U_i \in \mathcal{U}_{participant} - \mathcal{U}'_{participant}$ do
11:     $U_i$ performs $querySecretKeys(\cdot)$ and $groupKeyComputation(\cdot)$ functions.
12: end for
```

Figure 3.14. Participant Leave Function.

3.3.2. B-GKAP2 Model

In this model, we propose a new entity called the organization as a network participant. As presented in Figure 3.15, each organization have multiple peers and an isolated ledger. In this way, secret key variables can be stored and validated separately by each organization.

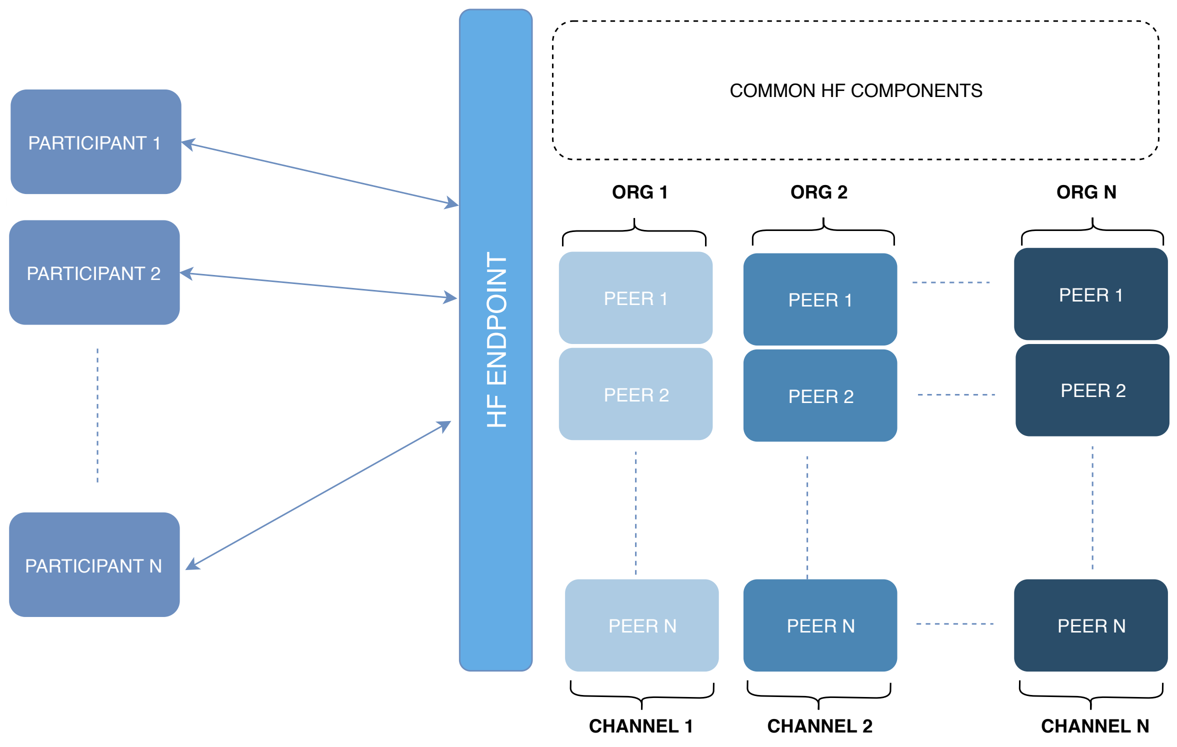
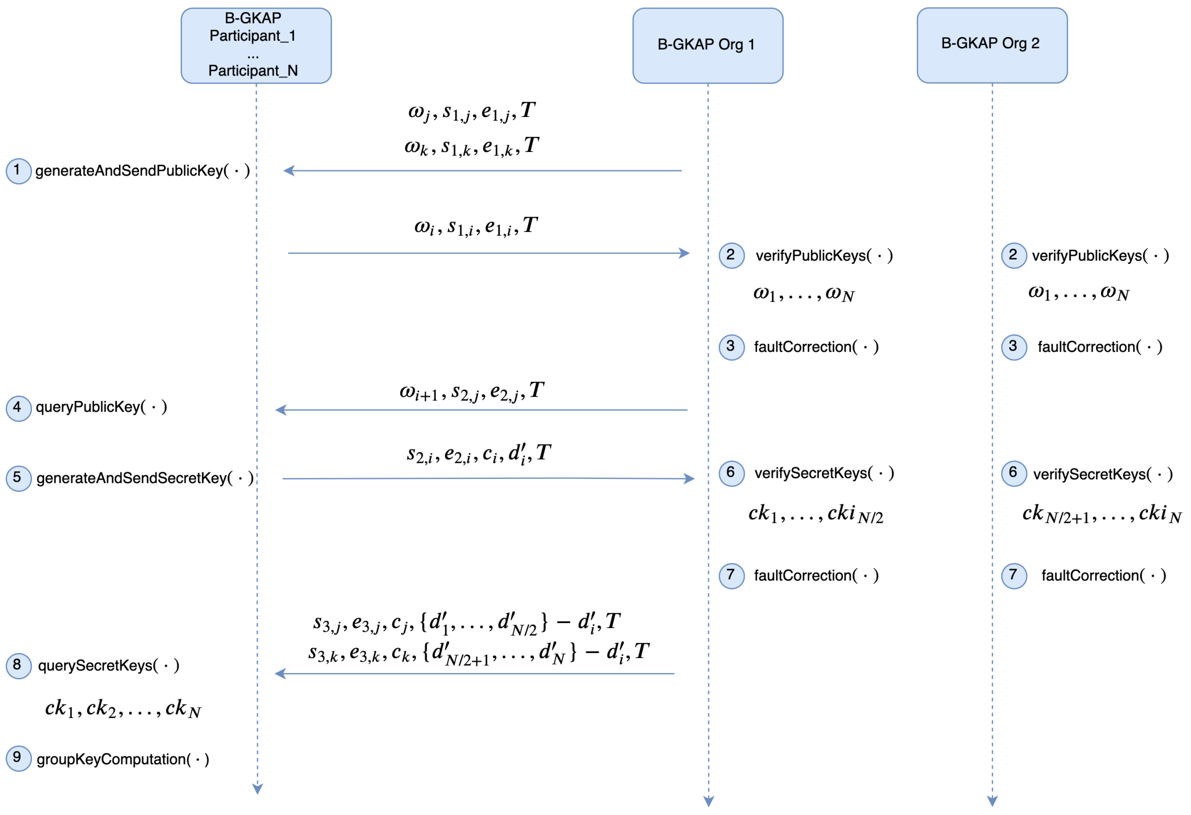


Figure 3.15. B-GKAP2 Model

As shown in Figure 3.16, B-GKAP2 protocol steps and transmitted variables between the participant and organization entities are presented in each step.



B-GKAP Participant_1 ... Participant_N
B-GKAP Org 1
B-GKAP Org 2
$\omega_j, s_{1,j}, e_{1,j}, T$
$\omega_k, s_{1,k}, e_{1,k}, T$
1 generateAndSendPublicKey(·)
$\omega_i, s_{1,i}, e_{1,i}, T$
2 verifyPublicKeys(·)
$\omega_1, \ldots, \omega_N$
3 faultCorrection(·)
2 verifyPublicKeys(·)
$\omega_1, \ldots, \omega_N$
3 faultCorrection(·)
$\omega_{i+1}, s_{2,j}, e_{2,j}, T$
4 queryPublicKey(·)
$s_{2,i}, e_{2,i}, c_i, d'_i, T$
5 generateAndSendSecretKey(·)
6 verifySecretKeys(·)
$cki_1, \ldots, cki_{N/2}$
7 faultCorrection(·)
6 verifySecretKeys(·)
$ck_{N/2+1}, \ldots, cki_N$
7 faultCorrection(·)
$s_{3,j}, e_{3,j}, c_j, \{d'_1, \ldots, d'_{N/2}\} - d'_i, T$
$s_{3,k}, e_{3,k}, c_k, \{d'_{N/2+1}, \ldots, d'_N\} - d'_i, T$
8 querySecretKeys(·)
$ck_1, ck_2, \ldots, ck_N$
9 groupKeyComputation(·)

Figure 3.16. B-GKAP2 Protocol

Public Key Distribution: Each organization $U_j \in U_{network}$ and each participant $U_i \in U_{participant}$ execute the function in Figure 3.17 to distribute temporary public keys.

```
generateAndSendPublicKeyv2(·):
 1: randomly select $t \in Z_q^*$
 2: $\omega = g^t mod p$
 3: Sign $\omega$, $e, s = SS(y, x, \omega)$
 4: Send the message $M = \{\omega, e, s, T\}$
```

Figure 3.17. Public Key Distribution Function v2.

Public Key Verification: Each organization $U_j \in U_{network}$ execute the function in Figure 3.18 to verify temporary public key of each participant $U_i \in \mathcal{U}$. According to verification result, the key is written to the ledger of U_j.

```
$verifyPublicKeysv2(\cdot)$:
1: for all $U_i \in \mathcal{U}_{participant}$ do
2:     Check the timestamp $T$
3:     if $SV(y_i, e_{1,i}, s_{1,i}, \omega_i)$ holds then
4:         $writeLedger(\omega_i)$
5:     end if
6: end for
```

Figure 3.18. Public Key Verification Function v2.

Fault Correction: Each organization $U_j \in U_{network}$ perform the function in Figure 3.19 to remove any participant $U_i \in \mathcal{U}_{participant}$ whose temporary public key or secret key verification fails.

```
$faultCorrectionv2(\cdot)$:
1: for all $U_i \in \mathcal{U}_{participant}$ do
2:     if $U_i$ is faulty then
3:         $U_i$ is removed from the participant group, $\mathcal{U}' = \mathcal{U} - U_i$
4:         For the participant $U_i$, $participantLeave(\cdot)$ is executed
5:     end if
6: end for
```

Figure 3.19. Fault Correction Function v2.

Public Key Query: In this step, each participant $U_i \in \mathcal{U}_{participant}$ requests for temporary public key of next participant U_{i+1} in the group from the target organization $U_j \in \mathcal{U}_{network}$ as in Figure 3.20.

```
$queryPublicKeyv2(\cdot)$:
1:  if $U_j \in \mathcal{U}_{network}$ then
2:      $\omega_{i+1} = readLedger()$
3:      Sign temporary public key of $U_{i+1}$, $e_j, s_j = SS(y_j, x_j, \omega_{i+1})$
4:      Send the message $M$ to the participant $U_i$,
                        $M = (\omega_{i+1}, e_j, s_j, T)$
5:  end if
6:  if $U_i \in \mathcal{U}_{participant}$ then
7:      Receive the message $M$
8:      Check timestamp $T$
9:      Verify signature of $U_j$, $SV(y_j, e_j, s_j, \omega_{i+1}))$
10: end if
```

Figure 3.20. Public Key Query Function v2.

Secret Key Distribution: Each participant $U_i \in \mathcal{U}_{participant}$ performs the function in Figure 3.21 to generate and distribute secret key (CK_i) to target organization $U_j \in \mathcal{U}_{network}$.

```
$generateAndSendSecretKeyv2(\cdot)$:
1: Generate $CK_i$: $CK_i = \omega_{(i+1)}^{t_i} \mod p = g^{t_i t_{i+1}} \mod p$
2: Select target organization $U_j$ where $j = i/(N/M) + 1$
3: Randomly select an integer $a \in Z_q^*$
4: $k = (\omega_j^a mod p) \mod q$
5: Randomly selects a line $L(x)$;
   $L(x) = xc_i + CK_i$ mod $q$,
   $c_i = g^a \mod p$.
6: $d_i = L(k) \mod q$
7: $d_i' = k \oplus d_i$
8: $e_{2,i}, s_{2,i} = SS(y_i, x_i, CK_i)$
9: Sends the message $M$ to $U_j$: $M = \{s_{2,i}, e_{2,i}, c_i, d_i', T\}$
```

Figure 3.21. Secret Key Distribution Function v2.

Secret Key Verification: Each organization $U_j \in \mathcal{U}_{network}$ performs the function in Figure 3.22 to verify secret keys of participants $U_i \in \mathcal{U}_{participant}$.

```
$verifySecretKeysv2(\cdot)$:
1: for all $U_i \in \mathcal{U}_{participant}$ do
2:     Receives message $M = \{s_{2,i}, e_{2,i}, c_i, d'_i, T\}$
3:     Recover the sub-key $CK_i$ and checks time-stamp $T$.
4:     $k = (c_i^{t_j} \mod p) \mod q$
5:     $d = d'_i \oplus k$
6:     $CK_i = d - c_i * k \mod q$
7:     Checks the signature of $U_i$.
8:     if $SV(y_i, e_{2,i}, s_{2,i}, CK_i)$ holds then
9:         $writeLedger(CK_i)$
10:    end if
11: end for
```

Figure 3.22. Secret Key Verification Function v2.

Secret Key Query: After the fault correction step, each participant $U_i \in \mathcal{U}_{participant}$ performs the function in Figure 3.23 to query secret keys from all organizations $U_j \in \mathcal{U}_{network}$.

```
$querySecretKeysv2(\cdot)$:
1: if $U_j \in \mathcal{U}_{network}$ then
2:     Randomly select an integer $a \in Z_q^*$
3:     $c_j = g^a \mod p$
4:     $k_i = (\omega_i^a mod p) mod q$
5:     for all $U_k \in \mathcal{U}_{participant} - U_i$ do
6:         Randomly select a line $L(x)$;
           $L(x) = xc_j + CK_k \mod q$.
7:         $d_k = L(k_i) \mod q$
8:         $d'_k = k_i \oplus d_k$
9:     end for
10:    Sign $CK_{1...N}$, $e_{2,j}, s_{2,j} = SS(y_j, x_j, CK_{1...N})$
11:    Send message $M$ to $U_i$: $M = \{s_{2,j}, e_{2,j}, c_j, \{d'_1, d'_2, \ldots, d'_N\}, T\}$
12: end if
13: if $U_i \in \mathcal{U}_{participant}$ then
14:    $k_i = (c_j^{t_i} \mod p) \mod q$
15:    for all $U_k \in \mathcal{U}_{participant} - U_i$ do
16:        $d_k = d'_k \oplus k_i$
17:        $CK_k = d_k - c_j * k_i \mod q$
18:    end for
19:    Check timestamp $T$
20:    Check the signature of $U_j$: $SV(y_j, s_{2,j}, e_{2,j}, CK_{1...N})$
21: end if
```

Figure 3.23. Secret Key Query Function v2.

Group Key Computation: Each participant by $U_i \in \mathcal{U}_{participant}$, computes the group key as in Figure 3.24.

```
$groupKeyComputationv2(\cdot)$:
1: for all $U_k \in \mathcal{U}_{participant}$ do
2:     $CK = ((CK'_1 CK'_2 \cdots CK'_N) mod p) mod q = (g^{t_1 t_2 + t_2 t_3 + \ldots + t_{n-1} t_n + t_n t_1} mod p) mod q$
3: end for
```

Figure 3.24. Group Key Computation Function v2.

4. SECURITY ANALYSIS

In this chapter, we go trough the security features that our base protocol in [11] provides. Then we provide proofs that B-GKAP also assures the same security level for backward confidentiality and forward secrecy features of dynamic group operations as in the base protocol. Finally, we discuss security models of B-GKAP1 and B-GKAP2.

4.1. Security Properties of Group Key Agreement Protocols

In this section, we discuss about security properties that recent group key agreement protocols provide [11,28,53].

4.1.1. Authentication

In group key agreement protocols, the authentication feature ensures that identifications of all involved participants are validated before proceeding to the further protocol steps. For B-GKAP protocol, as a first level of authentication, all participants are pre-identified with Hyperledger Fabric Certification Authority (CA) [10] component. As a requirement of interacting with Hyperledger Fabric Network, all participants must use Transport Layer Security (TLS) certificate to provide identification. The TLS certificate is created via HF Admin prior to the network initialization. In B-GKAP, we use a common TLS certificate for all the participants. For the second level of authentication, long-term key pairs of the participants are used. All long-term public keys of the participants must be singed via trusted CA. During variable transmissions between the participants and the network, payload is signed via long-term private key of the sender entity. Eventually, receiving entity verifies the signature of the payload by sender's long-term public key. As the signature scheme, we utilize Schnorr's method [54].

4.1.2. Fault Tolerance

In the course of group key agreement processes, malicious participants should be removed from the group without affecting the operations of remaining honest participants. In B-GKAP, detection and elimination of faulty participants occurs the during execution of $verifyPublicKeys(\cdot)$, $verifySecretKeys(\cdot)$ and $faultCorrection(\cdot)$ functions. If a malicious participant is detected in $verifyPublicKeys(\cdot)$ or $verifySecretKeys(\cdot)$ functions, related key of the participant is not written to the ledger. Therefore, $faultCorrection(\cdot)$ detects the missing key, and removes related participant from the group.

4.1.3. Forward Secrecy

In order to provide forward secrecy property, our protocol must provide a security mechanism to protect group keys even if produced private keys are compromised. In case of a long-term private key of any B-GKAP entity (participant, network, organization) is compromised, computed group keys will not be affected. Because in B-GKAP, long-term keys are used only for signing the payloads of the B-GKAP functions. Additionally, each entity in B-GKAP generates their temporary public and private keys in $generateAndSendPublicKey(\cdot)$ function for every group key computation process. In case the temporary private key of a participant is compromised, only the corresponding group key is compromised. Given the statements above, B-GKAP ensures forward secrecy feature.

4.2. Protection Against Security Attacks

In this section, we provide that B-GKAP has no vulnerability against following security attacks.

4.2.1. Impersonation Attack

Motivation of impersonation attacks is to take place any of B-GKAP entity during the protocol execution. To do that, an attacker needs to be able to generate the signature of an entity. Since our models are based on [11], we also utilize Schnorr signature scheme [54] for outputs of B-GKAP functions. As stated in [55] and [56], Schnorr is secure against impersonation and related key attacks, respectively.

4.2.2. Eavesdropping Attack

The purpose of the eavesdropping attack in B-GKAP is to capture the computed group key by sniffing the communication between the entities. In order to generate a group key, the attacker must grab secret keys (CK_i) of all participants ($CK_1, CK_2, \ldots, CK_N$). In B-GKAP, extraction of secret keys occurs in $verifySecretKeys(\cdot)$ and $querySecretKeys(\cdot)$ functions. In those functions, secret keys are extracted using c, d' variables of the sender and temporary private key (t) of the receiver. Since only the participants in the key agreement group U knows their temporary private key (t), to compute $k = (c^t mod p)$ equation, attacker should try to extract t_j from $\omega_j = g^{t_j} mod p$ for each participant U_j in U. Eventually, because of solving this equation is infeasible due to discrete logarithm problem, B-GKAP is secure against eavesdropping attack.

4.2.3. Replay Attack

During the communication between the B-GKAP entities, an attacker might sniff messages in the network and re-transmit sniffed messages for malicious reasons such as Denial of Service (DoS) attacks. To protect against replay attacks, timestamp variable (T) is added into the protocol messages. Hence, the receiver entity can check T value against replay attack attempts.

4.3. Security of Join and Leave Operations

Dynamic group operations enables group key agreement protocols to be more efficient during re-generation of group keys. In order to overcome security weaknesses stated in [57], a protocol must ensure forward and backward confidentiality features. Forward confidentiality feature assures that further group keys cannot be generated by a participant who lefts the group. And backward confidentiality feature warrants that previous group keys cannot be produced by recently joined participants. In the following sections, we prove that Leave and

Join operations in B-GKAP provide the same security features as [11] assures.

Prior to applying dynamic group operations, let the participant group be $\mathcal{U}_{participant} = \{U_1, U_2, U_3, \ldots, U_N\}$ and the group key be $CK = ((g^{t_1t_2 + \ldots + t_{i-1}t_i + t_it_{i+1} + \ldots + t_Nt_1}) mod p) mod q)$.

4.3.1. Join Operation Security

Lemma 1 Under the difficulty of discrete logarithm problem, join operation provides backward confidentiality.

Proof. For the join operation, last participant in the group U_N and joining participants should re-create temporary public and secret keys as stated in $participantJoin(\cdot)$ and in Section 3.2. In this case, let the joining participants be $U' = U_{N+1}, U_{N+2}, \ldots, U_{N+k}$, thus new group key is $CK' = ((g^{t_1t_2 + \ldots + t_{N-1}t'_N + t'_Nt_{N+1} + \ldots + t_{N+k}t_1}) mod p) mod q$. Since the difference between CK and CK' are $(g^{N-1t_N + t_Nt_{N+1}} mod p) mod q$ and $(g^{N-1t'_N + t'_Nt_{N+1}} mod p) mod q$, joined participants should find out t_N from $(g^{t_{N-1}t_N} mod p) mod q$ or $(g^{t_Nt_{N+1}} mod p) mod q$ to compute the previous group key. Since this state enforces joined participants to solve discrete logarithm problem, join operation of B-GKAP ensures backward confidentiality. □

4.3.2. Leave Operation Security

Lemma 2 Under the difficulty of discrete logarithm problem, leave operation provides forward confidentiality.

Proof. For the group U, let $U' = U_i, U_{i+1}, U_{i+2}, \ldots U_{i+k}$ be leaving participants where $U' \subseteq U$. As stated in function $participantLeave(\cdot)$ and in Section 3.2, the group key after the leave operation is $CK' = ((g^{t_1t_2 + \ldots + t_{i-2}t'_{i-1} + t'_{i-1}t_{i+k+1} \ldots + t_Nt_1}) mod p) mod q$. Since the difference between CK and CK' are $g^{t_{i-2}t_{i-1} + t_{i-1}t_i + t_it_{i+1}}$ and $g^{t_{i-2}t'_{i-1} + t'_{i-1}t_{i+k+1}}$, leaving participants should figure out t'_{i-1} from $(g^{t_{i-2}t'_{i-1}} mod p) mod q$ to compute subsequent group key. Eventually, this state enforces leaving participants to solve the discrete logarithm problem. Therefore, the leave operation of B-GKAP ensures forward confidentiality. □

4.4. Security of B-GKAP Models

In B-GKAP1, all the protocol variables of the participants are stored in the network entity. The network entity behaves as a participant except for the secret key generation and key computation. During the network initialization step ($generateAndSendPublicKey(\cdot)$), the network generates its own temporary key pairs (t_n, ω_n). Participants use the temporary public key of the network (ω_n) to encrypt their secret keys before the submission. Thus, the network can unveil the secret keys of the participants using its temporary private key (t_n). Additionally, the network also holds its own long-term key pair (x_n, y_n). These keys are used for signing the protocol variables that are sent to the participants. Hence, participants can ensure that they are receiving variables from a trusted entity.

Different than the model B-GKAP1, in B-GKAP2, secret key verification is distributed among the organizations. Since the ledger of each organization is isolated via the Fabric channels, the secret keys of the participants are kept isolated as well [58]. Therefore, even if an organization needs to compute the group key, all the organizations should cooperate together in a malicious manner. Hence, instead of trusting a single entity, we have distributed the trust among the organizations.

Moreover, Hyperledger Fabric platform provides storage immutability via blockchain ledgers. The ledger is distributed among peers, and to change the ledger data, a subset of peers needs to generate the same output. This feature is forced by endorsement policies. Additionally, the stored data is decentralized, hence, when a peer is compromised or failed, the system remains operational.

Definition (Honest-But-Curious Adversary): The honest-but-curious (HBC) adversary is a lawful participant in a communication protocol who has no other choice than following the defined protocol, on the other hand, it will attempt to learn all possible information as an intended recipient [59, 60].

Lemma 3 The existence of network participant identification and agreement on the common chaincode implementation, leads to the existence of ‘Honest-But-Curious Adversary’ model.

Proof. In B-GKAP protocol, prior to secret key distribution, identities of network participants are verified via $U_j \in \mathcal{U}_{network}$, $SV(y_j, e_{2,j}, s_{2,j}, \omega_j) \neq false$, and secret key CK_i of each participant $U_i \in \mathcal{U}_{participant}$ is encrypted via temporary public key ω_j of the network participant $U_j \in \mathcal{U}_{network}$. Thus, network participants can only receive intended parameters which are $\omega_i, s_i, e_i, d'_i, c_i$ for $U_i \in \mathcal{U}_{participant}$. Moreover, in Hyperledger Fabric, the only way to interact with the ledger is via Fabric chaincodes. Since implemented chaincode has defined a set of functionality and is shared among the peers, the network has no other choice but to follow the B-GKAP protocol. Given the properties, B-GKAP fits the HBC adversary model. $\square$

5. PERFORMANCE ANALYSIS

In this chapter, the performance analysis of B-GKAP models is presented. First, we start with the complexity analysis of the system. This analysis includes two components: communication complexity and computational complexity. Key computation and dynamic group operations of two B-GKAP models are also explained. Later on, our simulation environment is described. In the simulation environment section, hardware and software components of our testing environment are detailed. In the simulation cases and results section, the performance results of the system are presented. This section includes the effects of Hyperledger Fabric Orderer parameters, performance comparison of B-GKAP versus conventional implementation, and finally performance comparison of B-GKAP1 and B-GKAP2 models.

5.1. Complexity Analysis

Before discussing the simulation-based results, it is crucial to perform the complexity analysis of the system. Therefore, we can better understand how the system behaves under specific conditions.

5.1.1. Communication Complexity Analysis

In this section, we account communication complexity analysis of B-GKAP. The communication complexity is represented as $\mathcal{C}_t$.

5.1.1.1. Hyperledger Fabric (HF) Transaction.

In HF, each transaction should travel between different components of HF. Firstly, a transaction request is generated by a B-GKAP participant. This transaction should first reach to the endorser peers. After the endorser peers simulate and validate the transactions, endorsements are collected by the participant. Thereafter, the actual transaction is sent to the orderer. Lastly, the orderer broadcasts the transaction to all of the peers.

Given the information above;

- Let the number of endorser peers be N_{ep}.
- Let the total number of peers be N_p.

In order to be a HF transaction considered finished, it first goes backs and forth to the endorser peers $2 \times N_{ep}$, it then goes to the orderer. Finally, the transaction is disseminated to all of the peers, N_{ep}. In total, the communication complexity of a HF transaction is $2 \times O(N_{ep} + 1 + N_p)$ which can be simplified as: $C_t = O(2 \times N_{ep} + N_p)$.

5.1.1.2. Analysis of B-GKAP Functions. In B-GKAP, variable transmission occurs between participants ($U_i \in \mathcal{U}_{participant}$) and network participants ($U_j \in \mathcal{U}_{network}$). Since all transmitted variables are modular base of p and q, length of a variable is equal to its modular base.

Table 5.1. Transmission length of each B-GKAP function in bits.

Function	**Variables**	**Transmission length**
$generateAndSendPublicKey(\cdot)$	$\lvert\omega\rvert + \lvert e\rvert + \lvert s\rvert$	$(2q + p)$
$queryPublicKey(\cdot)$	$\lvert\omega\rvert + \lvert e\rvert + \lvert s\rvert$	$(2q + p)$
$generateAndSendSecretKey(\cdot)$	$\lvert c\rvert + \lvert d'\rvert + \lvert e\rvert + \lvert s\rvert$	$(3q + p)$
$querySecretKeys(\cdot)$	$\lvert\{d'_1, \ldots, d'_N\} - d'_i\rvert +$ $\lvert c\rvert + \lvert e\rvert + \lvert s\rvert$	$(N+1)q + p$

Table 5.1 provides network transmission length of each function in B-GKAP.

5.1.1.3. B-GKAP1 Key Computation. During key computation in B-GKAP1, several variable transmissions between participants and the network occur. While in protocol function $generateAndSendPublicKey(\cdot)$, variable transmission occurs two times for each participant, and for $queryPublicKey(\cdot)$, $generateAndSendSecretKey(\cdot)$, $querySecretKeys(\cdot)$ functions, variable transmission occurs for each participant $U_i \in \mathcal{U}_{participant}$.

Table 5.2. Communication cost of B-GKAP1 key computation for each participant.

Function	C_t
$generateAndSendPublicKey(\cdot)$	$\|4q+2p\|$
$queryPublicKey(\cdot)$	$\|2q+p\|$
$generateAndSendSecretKey(\cdot)$	$\|3q+p\|$
$querySecretKeys(\cdot)$	$\|(N+1)q+p\|$

According to Table 5.2, communication cost for each participant is $|(N+10)q+5p|$ and the total communication complexity is: $C_t = O(N)|(N+10)q+5p|$.

5.1.1.4. B-GKAP2 Key Computation. Let M be organization count for $U_i \in \mathcal{U}_{network}$. In B-GKAP2, for the functions $generateAndSendPublicKeyv2(\cdot)$ and $querySecretKeysv2(\cdot)$, network transmission occurs for each organization.

Table 5.3. Communication cost of B-GKAP2 key computation for each participant.

Function	C_t
$generateAndSendPublicKeyv2(\cdot)$	$\|4Mq+2Mp\|$
$queryPublicKeyv2(\cdot)$	$\|2q+p\|$
$generateAndSendSecretKeyv2(\cdot)$	$\|3q+p\|$
$querySecretKeysv2(\cdot)$	$\|(N+2M-1)q+Mp\|$

According to Table 5.3, communication cost for each participant $|(N+7M+4)q+(3M+2)p|$ bits and communication complexity is: $C_t = O(N)|(N+7M+4)q+(3M+2)p|$.

5.1.1.5. B-GKAP Join. For the join operation of B-GKAP, when a new participant joins the group, the group key should be re-computed. Therefore, for joining K participants, K participants should receive temporary public key of the network, $K+1$ participants should perform $generateAndSendPublicKey(\cdot)$ function, $K+2$ participants execute $queryPublicKey(\cdot)$

function, and $K+2$ participants should perform $generateAndSendSecretKey(\cdot)$. Finally, all participants should perform $querySecretKeys(\cdot)$.

Table 5.4. Communication cost of B-GKAP join operation for K joining participants.

Function	**B-GKAP1 C_t**	**B-GKAP2 C_t**
$generateAndSendPublicKey(\cdot)$	$(2K+1)\|2q+p\|$	$(2K+1)\|2Mq+Mp\|$
$queryPublicKey(\cdot)$	$(K+2)\|2q+p\|$	$(K+2)\|2q+p\|$
$generateAndSendSecretKey(\cdot)$	$(K+2)\|3q+p\|$	$(K+2)\|3q+p\|$
$querySecretKeys(\cdot)$	$(N+K)\|(N+1)q+p\|$	$(N+K)\|(N+2M-1)q+Mp\|$

Based on Table 5.4, in B-GKAP1, $5\times K+N+5$ network transmissions occurs. Eventually, the communication complexity of the join operation is $C_t = O(N+K)$.

For B-GKAP2, $generateAndSendPublicKeyv2(\cdot)$ and $querySecretKeysv2(\cdot)$ are executed for each organization. The transmission cost of other functions are same with B-GKAP1. Therefore, there are total $M(3\times K+N+1)+2\times K+4$ network transactions. If we consider that M is negligible against K and N, communication complexity is $C_t = O(N+K)$.

5.1.1.6. B-GKAP Leave. In leave operation of B-GKAP, for the leaving participant U_i, participant U_{i-1} executes $generateAndSendPublicKey(\cdot)$. Later on, participants U_{i-1} and U_{i-2} perform $queryPublicKey(\cdot)$. Moreover, participants U_{i-1} and U_{i-2} execute $generateAndSendSecretKey(\cdot)$. Eventually, all participants execute $querySecretKeys(\cdot)$.

Based on Table 5.5, for B-GKAP1, total network transmission is $N+5$ and communication complexity of leave operation is $C_t = O(N)$ where N is the remaining participant count in the group.

For B-GKAP2, $generateAndSendPublicKeyv2(\cdot)$ and $querySecretKeysv2(\cdot)$ are executed for each organization. For the organization count M, total number of network operations is $M(N+1)+4$, and communication complexity is $C_t = O(N)$.

Table 5.5. Communication cost of B-GKAP leave operation.

Function	**B-GKAP1** C_t	**B-GKAP2** C_t
$generateAndSendPublicKey(\cdot)$	$\|2q+p\|$	$\|2Mq+Mp\|$
$queryPublicKey(\cdot)$	$\|4q+2p\|$	$\|4q+2p\|$
$generateAndSendSecretKey(\cdot)$	$\|6q+2p\|$	$\|6q+2p\|$
$querySecretKeys(\cdot)$	$N\|(N+1)q+p\|$	$N\|(N+2M-1)q+Mp\|$

5.1.2. Computation Complexity Analysis

In computational cost analysis, we consider modular exponential operations as principal factor for calculating our results since other operations such as XOR, multiplication, and addition can be regarded as negligible in comparison with modular exponential operations. Time cost of these operations can be stated as $T_{exp} = O(x^y \mod z)$. In this section computational cost is represented as C_c. On the other hand, 'Block Size' parameter of HF Orderer also affect the performance of the system. For the 'Block Size' of B, and the number of GKA participants N, N/B blocks will be processed by the orderer in each parameter distribution round. Therefore, the network additionally perform $(N/B)T_{exp}$ modular exponential operations during the following operations.

5.1.2.1. B-GKAP1 Key Computation.

Key computation complexity analysis is expressed for B-GKAP participants and the network for this model.

Table 5.6. Computation complexity of B-GKAP1 for each participant.

Function	C_c
$generateAndSendPublicKey(\cdot)$	$O(1)T_{exp}$
$queryPublicKey(\cdot)$	$O(1)T_{exp}$
$generateAndSendSecretKey(\cdot)$	$O(1)T_{exp}$
$querySecretKeys(\cdot)$	$O(1)T_{exp}$

According to the Table 5.6, for each participant, total key computation complexity of the listed functions can be calculated as $2T_{exp} + 2T_{exp} + 4T_{exp} + 3T_{exp} = 11T_{exp}$ which can be simplified as $C_c = O(1)T_{exp}$.

Table 5.7. Computation complexity of B-GKAP1 for the network.

Function	C_c
$generateAndSendPublicKey(\cdot)$	$O(1)T_{exp}$
$queryPublicKey(\cdot)$	$O(N)T_{exp}$
$verifyPublicKeys(\cdot)$	$O(N)T_{exp}$
$verifySecretKeys(\cdot)$	$O(N)T_{exp}$
$querySecretKeys(\cdot)$	$O(N)T_{exp}$

As specified by the Table 5.7, for the network, total key computation complexity of the listed functions can be calculated as $2T_{exp} + 2NT_{exp} + 2NT_{exp} + 3NT_{exp} + 4NT_{exp} = (2 + 11N)T_{exp}$ which can be simplified as $C_c = O(N)T_{exp}$.

5.1.2.2. B-GKAP2 Key Computation. In this section, for the organization count M, key computation analysis is described for each participant and organization respectively.

Table 5.8. Computation complexity of B-GKAP2 for each participant.

Function	C_c
$generateAndSendPublicKeyv2(\cdot)$	$O(1)T_{exp}$
$queryPublicKeyv2(\cdot)$	$O(1)T_{exp}$
$generateAndSendSecretKeyv2(\cdot)$	$O(1)T_{exp}$
$querySecretKeysv2(\cdot)$	$O(1)T_{exp}$

Based on Table 5.8, for each participant, total number of modular exponential operations during key computation of the listed functions is $2MT_{exp} + 2MT_{exp} + 4T_{exp} + 3MT_{exp} =$

$(7M+4)T_{exp}$. Since the organization count M is significantly less than the participant count, N, we can simplify this statement as $C_c = O(1)T_{exp}$.

Table 5.9. Computation complexity of B-GKAP2 for each organization.

Function	C_c
$generateAndSendPublicKeyv2(\cdot)$	$O(1)T_{exp}$
$queryPublicKeyv2(\cdot)$	$O(N)T_{exp}$
$verifyPublicKeysv2(\cdot)$	$O(N)T_{exp}$
$verifySecretKeysv2(\cdot)$	$O(N)T_{exp}$
$querySecretKeysv2(\cdot)$	$O(N)T_{exp}$

According to Table 5.9, for each organization, total key computation complexity of the listed functions can be calculated as $2T_{exp} + (2N/M)T_{exp} + 2NT_{exp} + (3N/M)T_{exp} + 4NT_{exp} = (N(6M+5)/M+2)T_{exp}$. If we consider M is negligible compared to N, the expression can be simplified as $C_c = O(N)T_{exp}$.

5.1.2.3. B-GKAP Join Operation. During join operation, let the number of participants joining the group be K and participants be $U_{N+1}, U_{N+2}, \ldots, U_{N+K}$. As stated in Section 3.2, to join K participants into the group, K participants should fetch temporary public keys from the network, $K+1$ participants should perform $generateAndSendPublicKey(\cdot)$ function and $K+2$ participants should execute $generateAndSendSecretKey(\cdot)$. And finally, $K+N$ participants should perform $querySecretKeys(\cdot)$. For both B-GKAP1 and B-GKAP2 models, since computational complexity of each function is $O(1)T_{exp}$, the complexity of join operation is $C_c = O(1)T_{exp}$ as well.

While in join operation of K participants, the network participant should execute $generateAndSendPublicKey(\cdot)$ for K participants, $verifyPublicKeys(\cdot)$ for $K+1$ participants, $verifySecretKeys(\cdot)$ for $K+2$ participants. And finally for $K+N$ participants, the network should run $querySecretKeys(\cdot)$ function. As defined in key computation analysis, for B-GKAP1 and B-GKAP2 models, the total cost of these function execution for K joining participants is $C_c = O(K+N)T_{exp}$.

5.1.2.4. B-GKAP Leave Operation. During leave operation, let the leaving participant be U_i, as stated in Section 3.2, U_{i-1} executes *generateAndSendPublicKey*$(\cdot)$, and *generateAndSendSecretKey*$(\cdot)$ is performed by U_{i-1} and U_{i-2}. Lastly, the remaining participants execute *querySecret-Keys*$(\cdot)$. For the participants, computation complexity of each function is $O(1)T_{exp}$. Therefore, for B-GKAP1 and B-GKAP2 models, complexity of leave operation is $C_c = O(1)T_{exp}$.

When a participant U_i leaves from the group, the network executes *verifyPublicKeys*$(\cdot)$ for U_{i-1}, and *verifySecretKeys*$(\cdot)$ for U_{i-1} and U_{i-2}. And finally, the network executes *sendTemporaryPublicKey*$(\cdot)$, *querySecretKeys*$(\cdot)$ functions for the all participants in the network. In this case, for the models B-GKAP1 and B-GKAP2, leave operation complexity for the network is $C_c = O(N)T_{exp}$

5.2. Key Computation Complexity Comparison

In this section, we compare the communication and computation complexity of known GKA protocols. During the comparison, we take into account the complexity of each participant. Moreover, for the communication complexity, only transmitted messages by each participant are considered.

Table 5.10. Communication complexity of known protocols.

Protocol	C_c **of a participant**
Protocol in [26]	$(N+1)\|q\|+2\|p\|$
Protocol in [27]	$(N+2)\|q\|+4\|p\|$
Protocol in [28]	$(N+2)\|q\|+4\|p\|$
Protocol in [23]	$(N+2)\|q\|+4\|p\|$
KAP-PBC [11]	$(N+4)\|q\|+2\|p\|$
GKAP-MANET [6]	$2\|q\|+5\|p\|$
B-GKAP	$5\|q\|+2\|p\|$

According to Table 5.10, B-GKAP is more efficient than most of the protocols in terms of communication complexity for each participant. Additionally, in terms of total communication complexity, the other protocols perform network transmission to every other participant in the key agreement group. On the other hand, B-GKAP participants only transmit messages to the limited number of network participants. In other words, when the number of participants increases in B-GKAP, the number of network transmissions increases linearly instead of exponentially.

Table 5.11. Computation complexity of known protocols.

Protocol	C_c **of a participant**
Protocol in [23]	$O(N)T_{exp}$
Protocol in [5]	$\leq O(\log_3 N)T_{exp}$
Protocol in [9]	$\leq O(\log_2 N)T_{exp}$
Protocol in [61]	$O(\log_2 N)T_{exp}$
GKAP-MANET [6]	$O(N)T_{exp}$
KAP-PBC [11]	$O(N)T_{exp}$
B-GKAP	$O(1)T_{exp}$

In Table 5.11, the key computation complexity of known protocols is specified. In terms of computation complexity for each participant, our protocol performs better than other protocols. The reason is that in B-GKAP, participants only perform verification of the network participants. In other protocols, each participant performs verification of other participants.

5.3. Simulation Environment

In order to simulate B-GKAP, we have used a machine with Intel® Core$^{\text{TM}}$ i7-4870HQ (2.2GHz $\times$ 4), L2 Cache 256KB, L3 Cache 6MB, 16GB RAM and 256GB HDD space. The operating system of the machine is macOS Mojave (version 10.14.16). Since Hyperledger Fabric network components runs on docker containers, we have used Docker Engine (version 18.06.1-ce-mac73 (26764) stable) [62], and to orchestrate the containers we have utilized Docker Compose (version 1.22.0) [63]. We have used Hyperledger Fabric version 1.4.3 [64]. For both the B-GKAP chaincode and B-GKAP participant simulation, we have used Go programming language (version 1.13.3 darwin/amd64) [65].

5.4. Simulation Cases and Results

In this section, we present several simulation cases and results. As the first case, effects of batch size and batch count parameters are investigated. Later on, we compare key computation performance of B-GKAP with our base GKAP model [11]. Lastly, performance results of B-GKAP1 and B-GKAP2 models are presented.

During simulations, for each participant in B-GKAP, a new process is started. After all processes are up and running, participants can start interacting with B-GKAP network. We measure the time for the whole key computation operation and for each step in B-GKAP. At the end of each simulation, these results are written to a file. To determine key computation time, we compute the mean key computation times for each participant. Additionally, the whole operation is repeated multiple times before being presented in the graphs.

5.4.1. Effect of Batch Timeout and Batch Size

During our experiments, we investigate three parameters which are ‘Batch Timeout’, ‘Batch size’ and group key agreement participant count. ‘Batch Size’ and ‘Batch Timeout’ are two main parameters of Hyperledger Fabric (HF) Orderer component [66]. Before the transactions are ordered, the orderer either waits for certain amount of transactions or waits for a timeout counter to proceed ordering and disseminating collected transactions to the peers. In this case, the number of maximum transactions in batch is controlled by ‘Batch Size’, and the timeout can be controlled by ‘Batch Timeout’ parameters.

For this experiment, we have used B-GKAP1 model and we set participant count as 20, 50, 80 and for ‘Batch Timeout’ parameter as 1, 2, 3 *seconds*. Finally, we set ‘Batch Size’ sequentially as 10, 20, ..., 100.

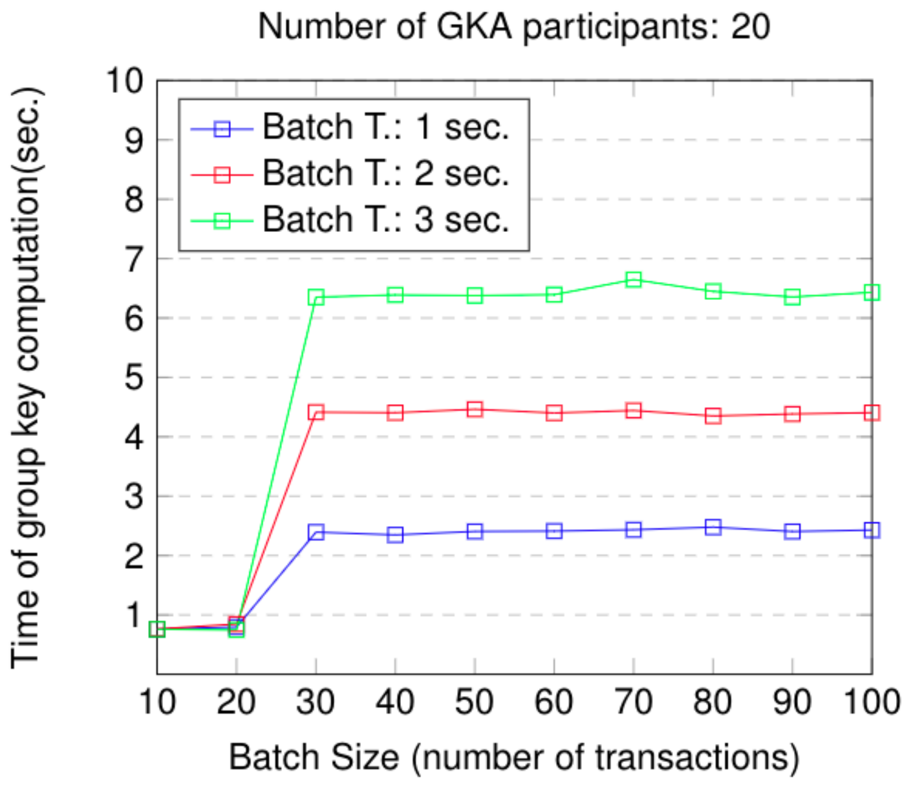


Figure 5.1. Effect of Batch Size and Batch Timeout Parameters with 20 participants.

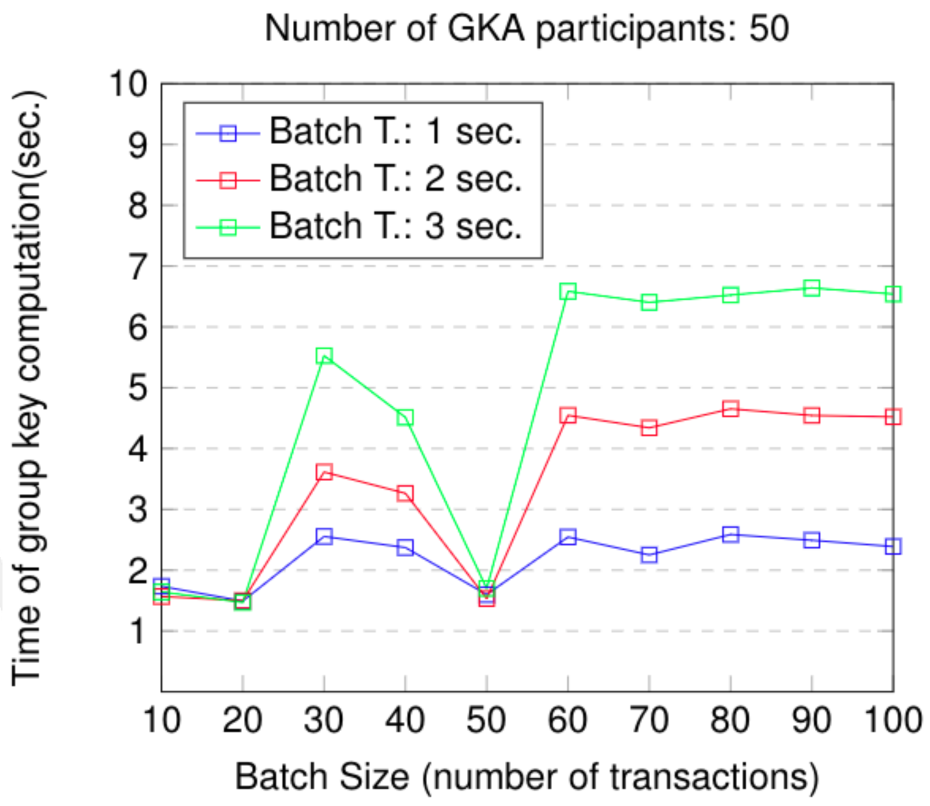


Figure 5.2. Effect of Batch Size and Batch Timeout Parameters with 50 participants.

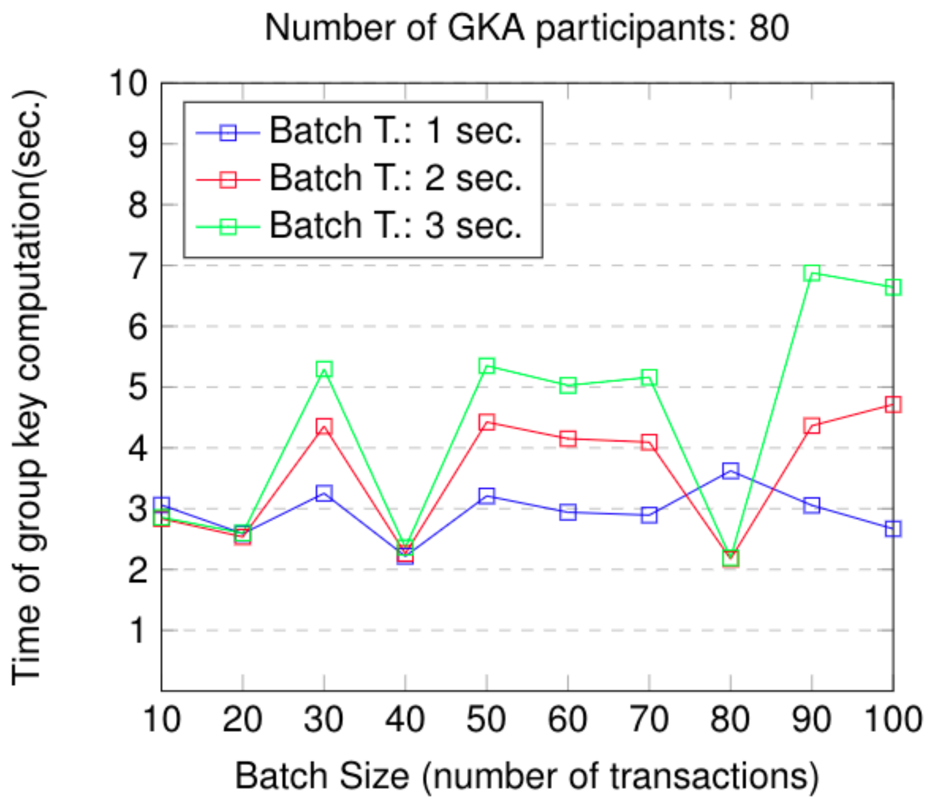


Figure 5.3. Effect of Batch Size and Batch Timeout Parameters with 80 participants.

In B-GKAP, for functions $verifyPublicKeys(\cdot)$ and $verifySecretKeys(\cdot)$, the orderer needs to perform ordering and dissemination operations. In case of B-GKAP1, the number of incoming transactions for each function is equal to the participant count. Therefore, if we set participant count as N and batch size as K, the orderer needs to perform ordering operation for N/K times if no timeout occurs.

Figure 5.1, 5.2, and 5.3 show that, until the ‘Batch Size’ parameter reaches to the participant count, number of incoming transactions to the orderer exceeds the batch size. Therefore, the orderer processes the transactions as multiple batches. And if there is not enough transaction to fill the last batch, the orderer waits for the ‘Batch Timeout’, which therefore increases the key computation time. When the batch size reaches to the participant count, group key computation time drops down to the minimum level if the timeout does not occur. The reason is that it is possible to process all the transactions in a single transaction block which increases the efficiency of the system. Finally, after the ‘Batch Size’ parameter surpasses the participant count, incoming transactions cannot fill the ‘Batch Size’ and the orderer must wait for the ‘Batch Timeout’. This condition explains that when the ‘Batch Timeout’ parameter increases, group key computation time increases accordingly.

5.4.2. Conventional versus B-GKAP Implementation

To implement the conventional method in [11], for each group key agreement participant, a new process is started and an HTTP server is established. Port numbers for HTTP servers are determined by $basePort + participantId$. For instance, if the base port is 2000, and the participant count is 100, port numbers are $2000, 2001, 2002, \ldots, 2100$. Thus, participants use HTTP GET and POST methods to broadcast and get the protocol variables.

Let n be the participant count. Figure 5.4, compares the performance of conventional and B-GKAP methods. In this comparison, we have used model B-GKAP1. As the orderer parameters we have set batch size to n, and for the batch timeout we set 5 seconds. The setup of B-GKAP is two peers and one orderer.

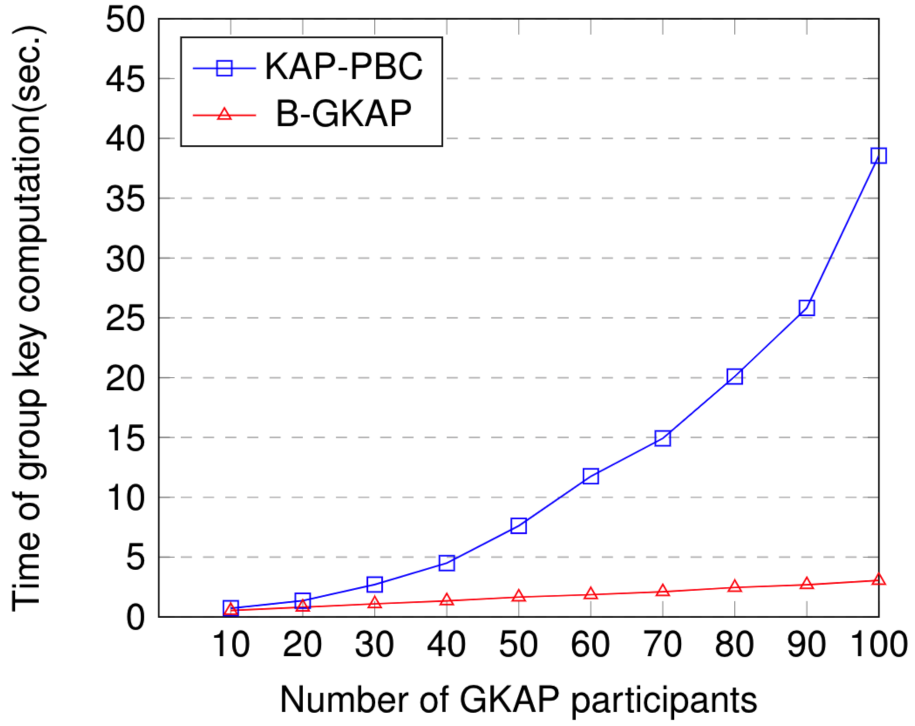


Figure 5.4. B-GKAP and Conventional Model Comparison

The results show that group key computation time of the conventional method increases exponentially as participant count increases. On the other hand, group key computation of B-GKAP increases linearly. First of all, communication complexity of the conventional method is $O(N^2)$ [11], in comparison, communication complexity of B-GKAP is $O(N)$ as stated in Section 5.1.1. For the computational complexity, the conventional method has $O(N)$ for each participant while B-GKAP has $O(1)$ for each participant and $O(N)$ for the network.

5.4.3. Comparison of B-GKAP Models

In this section, we compare key computation performance of B-GKAP1 and B-GKAP2 models. As simulation parameters, batch timeout is set to 5 seconds, and batch size parameter is set to N (participant count) for B-GKAP1 and $N/2$ for the B-GKAP2. In B-GKAP2, secret keys of the participants are divided between each organization, and batch size parameter affects each organization individually. Network setup of B-GKAP1 is two peer one orderer, on the other hand, for B-GKAP2, each of the two organizations has two peers.

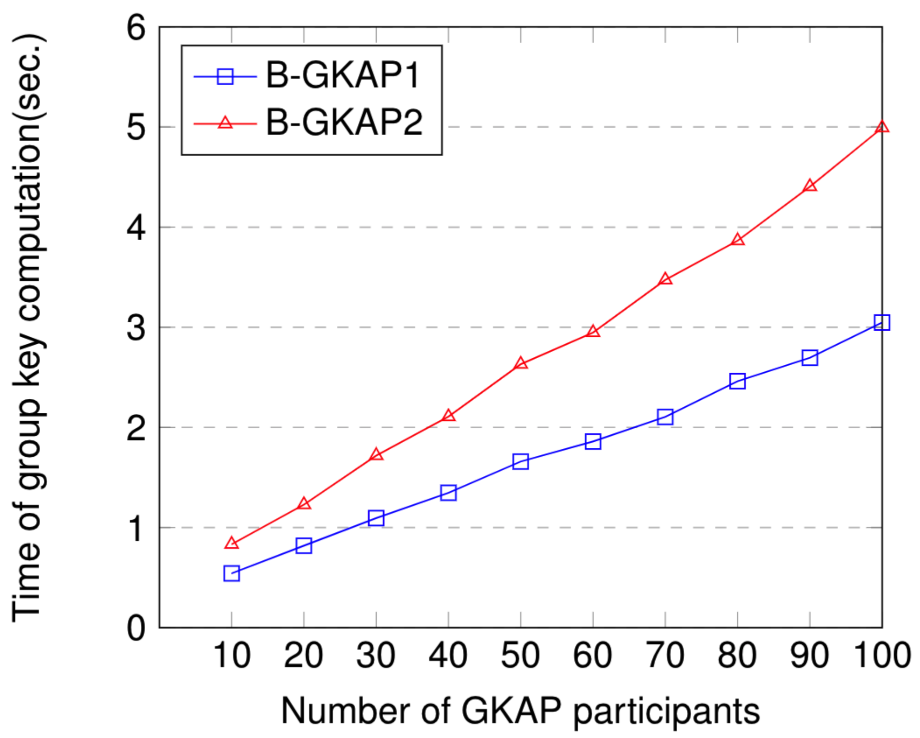


Figure 5.5. B-GKAP1 and B-GKAP2 Model Comparison

Figure 5.5 shows that group key computation time for two B-GKAP models are linearly increased when number of participants increased. B-GKAP2 has taken more time to compute group keys. There are two reasons for this result. The first reason is that each participant has to connect to the endpoints of two organizations simultaneously, therefore load of the network increased. The second reason is that, for N participant number, the orderer needs to process N incoming transactions as two transaction blocks for the function $verifyPublicKeys(\cdot)$.

6. CONCLUSION AND FUTURE STUDY

In this study, we present B-GKAP which employs Hyperledger Fabric blockchain platform to speed up the group key computation. With our protocol, the computation overhead of the group key agreement participants is decreased significantly by migrating the verification of the distributed parameters to the network participants. Thus, participants with low computation power and energy resource can easily adopt our protocol. Additionally, we have reduced the number of network transmissions for group key computation, leave and join operations. Hence, for the IoT systems where participant group changes frequently, our solution provides more efficient dynamic operations. Furthermore, with the B-GKAP2 model, we have distributed secret keys of the participants among the organizations via Fabric Channels. In this way, malicious organizations cannot generate group keys without colluding.

Hyperledger Fabric platform provides immutable storage property for stored variables via blockchain ledger. Additionally, in Fabric, not only valid but also invalid transactions are stored in the ledger. This feature makes our system auditable for further investigations.

Another important feature of Hyperleger Fabric is its modular architecture [67]. For instance, its consensus protocol can be replaced with more efficient methods as a future extension. Moreover, with its modular membership service provider, various authentication schemes can be utilized depending on the usage area of the protocol.

Additionally, to overcome the problem of colluding organizations, Fabric chaincode runtime environment, and ledger storage can be transferred to a Trusted Execution Environment (TEE) [68–71]. In this way, even organizations cannot access the secret keys of the participants.

REFERENCES

1. Nakamoto, S., *Bitcoin: A peer-to-peer electronic cash system*, Tech. rep., Manubot, 2008.

2. Androulaki, E., Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, A. Barger, S. Cocco, J. Yellick, V. Bortnikov, C. Cachin, K. Christidis, A. Caro, D. Enyeart and G. Laventman, "Hyperledger fabric: a distributed operating system for permissioned blockchains", pp. 1–15, 04 2018.

3. Diffie, W. and M. Hellman, "New Directions in Cryptography", *Information Theory, IEEE Transactions on*, Vol. 22, pp. 644 – 654, 12 1976.

4. Tseng, Y.-M., "A communication-efficient and fault-tolerant conference-key agreement protocol with forward secrecy", *Journal of Systems and Software*, Vol. 80, pp. 1091–1101, 07 2007.

5. Dutta, R. and R. Barua, "Dynamic Group Key Agreement in Tree-Based Setting", Vol. 2005, pp. 101–112, 07 2005.

6. Ermiş, O., Ş. Bahtiyar, E. Anarim and U. Çağlayan, "A secure and efficient group key agreement approach for mobile ad hoc networks", *Ad Hoc Networks, Volume 67, Pages 24-39, Elsevier*, 2017.

7. Konstantinou, E., "Cluster-based Group Key Agreement for Wireless Ad hoc Networks", *2008 Third International Conference on Availability, Reliability and Security*, pp. 550–557, March 2008.

8. Dutta, R. and T. Dowling, "Secure and Efficient Group Key Agreements for Cluster Based Networks", *Transactions on Computational Science*, Vol. 4, pp. 87–116, 01 2009.

9. Kim, Y., A. Perrig and G. Tsudik, “Tree-based Group Key Agreement”, *ACM Transactions on Information and System Security*, Vol. 7, 02 2002.

10. Vukolić, M., *Hyperledger fabric: towards scalable blockchain for business*, Tech. rep., Tech. rep. Trust in Digital Life 2016. IBM Research, 2016. URl: https://www. zurich. ibm. com/dccl/papers/cachin_dccl. pdf, 2015.

11. Ermiş, O., Ş. Bahtiyar, E. Anarim and U. Çağlayan, “A Key Agreement Protocol with Partial Backward Confidentiality”, *Computer Networks, Volume 129, Part 1, Pages 159-177, Elsevier*, 2017.

12. “Hyperledger Fabric”, `https://www.hyperledger.org/projects/fabric`, accessed: 2019-09-30.

13. Boneh, D., “The Decision Diffie-Hellman Problem”, *Proceedings of 3rd Algorithmic Number Theory Symposium*, Vol. 1423, 06 1998.

14. Wu, Q., Y. Mu, W. Susilo, b. Qin and J. Domingo-Ferrer, “Asymmetric Group Key Agreement”, Vol. 5479, pp. 153–170, 04 2009.

15. Schneier, B., *Applied Cryptography, 20th Anniversary Edition*, 1995.

16. McCurley, K., “The discrete logarithm problem”, *Proceedings of Symposia in Applied Mathematics*, Vol. 42, 01 1991.

17. Diffie, W., P. Oorschot and M. Wiener, “Authentication and Authenticated Key Exchanges.”, *Des. Codes Cryptography*, Vol. 2, pp. 107–125, 06 1992.

18. Ingemarsson, I., D. Tang and C. Wong, “A conference key distribution system”, *IEEE Transactions on Information Theory*, Vol. 28, pp. 714–719, 09 1982.

19. Burmester, M. and Y. Desmedt, “A secure and efficient conference key distribution system”, *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 275–286, Springer, 1994.

20. Steiner, M., G. Tsudik and M. Waidner, “Key agreement in dynamic peer groups”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 8, pp. 769–780, Aug 2000.

21. Chuang, Y.-H. and Y.-M. Tseng, “An efficient dynamic group key agreement protocol for imbalanced wireless networks”, *Int. Journal of Network Management*, Vol. 20, pp. 167–180, 07 2010.

22. Dutta, R. and R. Barua, “Constant Round Dynamic Group Key Agreement”, pp. 74–88, 09 2005.

23. Ermiş, O., Ş. Bahtiyar, E. Anarim and U. Çağlayan, “An improved conference-key agreement protocol for dynamic groups with efficient fault correction”, *Security and Communication Networks*, Vol. 8, pp. 1347–1359, 05 2015.

24. Tzeng, W.-G., “A secure fault-tolerant conference-key agreement protocol”, *Computers, IEEE Transactions on*, Vol. 51, pp. 373–379, 05 2002.

25. Cheng, J.-C. and C.-S. Laih, “Conference key agreement protocol with non-interactive fault-tolerance over broadcast network”, *Int. J. Inf. Sec.*, Vol. 8, pp. 37–48, 02 2009.

26. Huang, K.-H., Y.-F. Chung, H.-H. Lee, F. Lai and T.-S. Chen, “A conference key agreement protocol with fault-tolerant capability”, *Computer Standards and Interfaces*, Vol. 31, pp. 401–405, 02 2009.

27. Tseng, Y.-M., “An Improved Conference-Key Agreement Protocol with Forward Secrecy.”, *Informatica, Lith. Acad. Sci.*, Vol. 16, pp. 275–284, 01 2005.

28. Ermiş, O., Ş. Bahtiyar, E. Anarim and U. Çağlayan, “An Improved Fault-Tolerant Conference-Key Protocol with Forward Secrecy”, , 11 2013.

29. Abdel-Hafez, A., A. Miri and L. Orozco-Barbosa, “Authenticated Group Key Agreement Protocols for Ad hoc Wireless Networks.”, *I. J. Network Security*, Vol. 4, pp. 90–

98, 01 2007.

30. Lu, Y., G. Xu and Y. Yang, “Anonymous three-factor authenticated key agreement for wireless sensor networks”, *Wireless Networks*, Vol. 25, 11 2017.

31. Challa, S., A. K. Das, V. Odelu, N. Kumar, S. Kumari, K. Khan and A. Vasilakos, “An efficient ECC-based provably secure three-factor user authentication and key agreement protocol for wireless healthcare sensor networks”, *Computers and Electrical Engineering*, Vol. 69, 08 2017.

32. Tang, W., K. Zhang, J. Ren, Y. Zhang and X. Shen, “Flexible and Efficient Authenticated Key Agreement Scheme for BANs Based on Physiological Features”, *IEEE Transactions on Mobile Computing*, Vol. 18, No. 4, pp. 845–856, April 2019.

33. Shen, J., S. Chang, J. Shen, Q. Liu and X. Sun, “A lightweight multi-layer authentication protocol for wireless body area networks”, *Future Generation Computer Systems*, Vol. 78, pp. 956 – 963, 2018, `http://www.sciencedirect.com/science/article/pii/S0167739X16306963`.

34. Islam, S. H., M. S. Obaidat, P. Vijayakumar, E. Abdulhay, F. Li and M. K. C. Reddy, “A robust and efficient password-based conditional privacy preserving authentication and group-key agreement protocol for VANETs”, *Future Generation Computer Systems*, Vol. 84, pp. 216 – 227, 2018, `http://www.sciencedirect.com/science/article/pii/S0167739X17308439`.

35. Haber, S. and W. Stornetta, “How to Time-Stamp a Digital Document”, *Journal of Cryptology*, Vol. 3, 09 1999.

36. Merkle, R., “Protocols for Public Key Cryptosystems”, pp. 122–134, 04 1980.

37. Sankar, L. S., M. Sindhu and M. Sethumadhavan, “Survey of consensus protocols on blockchain applications”, *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pp. 1–5, Jan 2017.

38. Buterin, V., "A next-generation smart contract and decentralized application platforme", https://github.com/ethereum/wiki/wiki/White-Paper, accessed: 2019-12-13.

39. "Ethereum 2.0 Phases", https://docs.ethhub.io/ethereum-roadmap, accessed: 2019-12-13.

40. King, S. and S. Nadal, "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake", , 12 2019.

41. Lin, I.-C. and T.-C. Liao, "A survey of blockchain security issues and challenges", *International Journal of Network Security*, Vol. 19, pp. 653–659, 09 2017.

42. The Linux Foundation, "Hyperledger Fabric", https://www.hyperledger.org/.

43. The Linux Foundation, "The Linux Foundation – Supporting Open Source Ecosystems", https://www.linuxfoundation.org/.

44. Lamport, L., R. Shostak and M. Pease, "The Byzantine generals problem", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 4, No. 3, pp. 382–401, 1982.

45. Castro, M. and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery", *ACM Transactions on Computer Systems (TOCS)*, Vol. 20, No. 4, pp. 398–461, 2002.

46. Zheng, X. S. D. H. W. H., Z, "Blockchain Challenges and Opportunities : A Survey Shaoan Xie Hong-Ning Dai Huaimin Wang", *International Journal of Web and Grid Services, Pages 1-24*, 2016.

47. Ongaro, D. and J. Ousterhout, "In search of an understandable consensus algorithm", *2014 USENIX Annual Technical Conference (ATC)*, pp. 305–319, 2014.

48. Cloud Native Computing Foundation, "gRPC", https://grpc.io.

49. GitHub, "LevelDB", https://github.com/syndtr/goleveldb/.

50. The Apache Software Foundation, "Apache CouchDB", `https://couchdb.apache.org/`.

51. The Apache Software Foundation, "Apache Kafka", `http://kafka.apache.org/`.

52. Barger, A., Y. Manevich, B. Mandler, V. Bortnikov, G. Laventman and G. Chockler, "Scalable Communication Middleware for Permissioned Distributed Ledgers", *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, pp. 23:1–23:1, ACM, New York, NY, USA, 2017, `http://doi.acm.org/10.1145/3078468.3078492`.

53. Tseng, Y.-M., "A communication-efficient and fault-tolerant conference-key agreement protocol with forward secrecy", *Journal of Systems and Software*, Vol. 80, pp. 1091–1101, 07 2007.

54. Schnorr, C., "Efficient Identification and Signatures for Smart Cards", pp. 239–252, 01 1995.

55. Bellare, M. and A. Palacio, "GQ and Schnorr Identification Schemes: Proofs of Security against Impersonation under Active and Concurrent Attacks", Vol. 2442, pp. 149–162, 09 2002.

56. Morita, H., J. Schuldt, G. Hanaoka and T. Iwata, "On the Security of the Schnorr Signature Scheme and DSA Against Related-Key Attacks", pp. 20–35, 03 2016.

57. Lee, S., J. Kim and S. Hong, "Security weakness of Tseng's fault-tolerant conference key agreement protocol", *Journal of Systems and Software*, Vol. 82, pp. 1163–1167, 07 2009.

58. IBM, "Private and confidential transactions with Hyperledger Fabric", `https://developer.ibm.com/tutorials/category/blockchain/page/3`.

59. Paverd, A., A. Martin and I. Brown, "Modelling and automatically analysing privacy

properties for honest-but-curious adversaries", *University of Oxford, Tech. Rep*, 2014.

60. Cornell University, "Introduction to Cryptography", `http://www.cs.cornell.edu/courses/cs6830/2011fa/scribes/lecture22.pdf`.

61. Perrig, A., "Efficient collaborative key management protocols for secure autonomous group communication", pp. 192–202, 1999.

62. Docker Inc., "About Docker Engine", `https://docs.docker.com/engine/`.

63. Docker Inc., "Overview of Docker Compose", `https://docs.docker.com/compose/`.

64. The Linux Foundation, "Install Samples, Binaries and Docker Images", `https://hyperledger-fabric.readthedocs.io/en/release-1.4/install.html`.

65. Google, "The Go Project", `https://golang.org/project/`.

66. Nathan, S., P. Thakkar and B. Vishwanathan, "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform", , 09 2018.

67. "Modularity of Hyperledger Fabric, url = https://hyperledger-fabric.readthedocs.io/en/release-1.4/whatis.htm, note = Accessed: 2019-09-30", .

68. Brandenburger, M., C. Cachin, R. Kapitza and A. Sorniotti, "Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric", , 05 2018.

69. Brandenburger, M. and C. Cachin, "Challenges for Combining Smart Contracts with Trusted Computing", *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, pp. 20–21, ACM, New York, NY, USA, 2018, `http://doi.acm.org/10.1145/3268935.3268944`.

70. Das, B., S. Kanchanapalli and R. Vigneswaran, "Enhancing Security and Privacy of Permissioned Blockchain using Intel SGX", *Third Workshop on Blockchain Technologies*

and its Applications, 02 2019.

71. Schunter, M., "Intel Software Guard Extensions: Introduction and Open Research Challenges", *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pp. 1–1, ACM, New York, NY, USA, 2016, `http://doi.acm.org/10.1145/2995306.2995307`.