ABSTRACT

QIAO, SEN. Group Signatures for Preserving Anonymity in Blockchain Supply Chain Transactions. (Under the direction of Kemafor Ogan.)

Supply chain applications have increasingly integrated blockchain platforms to eliminate the trust barrier between numerous and autonomous supply chain participants. The role of the blockchain is to record and track key transactional information that can be used for automatic, trustworthy verification steps involved in certain transactions. Supply chain workflow has multiple phases beginning from procurement, to operations management, to consumers, and each phase involves different types of transactions. For example, the procurement phase involves transactions such as requests for quotes (RFQ) for goods and services, offers and bids, invoicing, billing, product inventory, shipping, and tracking status.

In certain contexts, there is a need to anonymize the identity of transaction owners because it helps preserve competitive advantage. Sometimes, the need for privacy rises to the level of keeping information about current and previous business collaborators secret. Unfortunately, the typical identification scheme on most blockchains, the regular public key-based signature, is primarily intended to reveal transaction identity as public key-based signatures verifying the signature with the public key will always reveal the identity.

Group signatures are a cryptographic schema that successfully removes the link from the signer's public key as group signatures created only verifies the signer's group. This property allows any group member to sign a message on behalf of the whole group, allowing signed messages or group signatures can only be associated with the group. We implemented our group signature cryptographic primitives into our distributed ledger, utilizing the BigchainDB platform.

Within the marketplace system, there is a need to ensure the transaction is from the same user to conduct business. Group signature schema breaks this system by making it impossible for one user to verify another's identity. We applied a hash-based linkable technique that links group signatures together. This linkability ensures subsequent transactions by the same user are linkable, immutable, and undeniable. Hence while one doesn't know who they spoke with, they are guaranteed to know when they communicate with the same individual.

Currently, blockchains focusing on user anonymity are not related to linking group signatures. This work aims to implement group signature transactions that are linkable within a supply chain distributed ledger by utilizing existing group signatures and a Hyperledger crypto library. Our contribution is (i.) the implementation of a group signature scheme that can be linked to another transaction by the same signer and (ii.) its integration into the transaction architecture of a blockchain platform called SmartChainDB.

In our work, we identify the components needed in a linkable group signature into a transaction. We identify and implement the roles of SmartChainDB server, SmartChainDB driver, MongoDB, and group manager with respect to the user.

© Copyright 2021 by Sen Qiao

All Rights Reserved

Group Signatures for Preserving Anonymity in Blockchain Supply Chain Transactions

by
Sen Qiao

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2021

APPROVED BY:

Alessandra Scafuro

William Enck

Kemafor Ogan
Chair of Advisory Committee

DEDICATION

To Dad who believed in me and was proud of me until the very end.

BIOGRAPHY

Sen Qiao is a native of Chengdu, China, and spent his childhood there until moving to Miami, FL, at 8 years old. Sen graduated with a B.S. in Computer Science and a B.A. in Economics from University of North Carolina, Chapel Hill in the Class of 2017. After graduating, he was employed at the GIS Center at Florida International University as a full-stack developer. Sen became a Master of Science in Computer Science candidate at North Carolina State University in 2019 and is projected to complete the program in spring 2021.

ACKNOWLEDGEMENTS

I would like to thank my advisors for their help and for their patience and understanding. I would also like to thank Akash, Duy, and Varun. Akash for his accomplishments with the SmartChainDB system. Duy for his UI development. Varun for his expert understanding of cryptography and cryptographic schemes.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

Chapter 1 INTRODUCTION **1**
1.1 Security Model 2
1.1.1 Threat Model Assumptions 2
1.1.2 Trust Model Assumptions 2
1.2 Additional Conditions 3
1.3 BigchainDB 4
1.3.1 Blockchain 4
1.4 Group Signature 6
1.5 Linkable Transactions 7
1.6 Contributions 7

Chapter 2 Background and Related Works **10**
2.1 Blockchain 10
2.1.1 Blockchain Conception 10
2.1.2 Blockchain Database 10
2.1.3 Network Fault Tolerance 11
2.1.4 Blockchain Confidentiality 11
2.2 Transaction Anonymity 11
2.3 Requester Anonymity 12
2.4 Ring Signature 12
2.5 Group Signature 12
2.5.1 Linkable Group Signature in Asynchronous Blockchain Network 13

Chapter 3 Methods **14**
3.1 Assumptions and Useful Methods 14
3.1.1 Bilinear Mapping Assumption 14
3.1.2 Re-randomization 14
3.1.3 Zero Knowledge Proof 15
3.2 Pointcheval & Sanders Group Signature Scheme 15
3.2.1 GSetup & PKIJoin 15
3.2.2 GJoin 16
3.2.3 GSign 16
3.2.4 GVerify 18
3.2.5 GOpen 18

Chapter 4 Implementation **20**
4.1 Overview 20
4.1.1 SmartChainDB Model 20
4.1.2 Basic Group Signature Model 22
4.2 Libraries 22
4.2.1 Charm 22
4.2.2 Hyperledger Ursa 23

4.3 Integrating Blockchain and Group Signature 24
4.3.1 Group Signature Transaction Structure 24
4.3.2 System Components 24
4.3.3 Linkability with hashes 26
4.3.4 Requester interaction with the system 27

Chapter 5 Results 30
5.1 Group Signature vs Basic Signature 30
5.2 Increasing Group Members 31
5.3 Group Transaction vs Basic Transaction 32

Chapter 6 Conclusion and Future Work 35
6.1 Conclusion 35
6.2 Future Work 36
6.2.1 More Powerful Adversary 36
6.3 Appendix: Group Signature Pseudocode 37

BIBLIOGRAPHY 43

LIST OF TABLES

Table 5.1 Time Cost of Group Signature 31
Table 5.2 Scalability of Group Signature 32
Table 5.3 Comparing Transaction Times 33
Table 5.4 Statistical Analysis 33

LIST OF FIGURES

Figure 1.1 Television Supply Chain 2
Figure 1.2 Supplier and Manufacturer Interaction 4
Figure 1.3 BigChainDB nodes 5
Figure 1.4 Simple Blockchain 6
Figure 1.5 Hemingway has more pages in his book, so Shakespeare adopts all of Hemingway's pages (left). Both Hemingway and Shakespeare have the same number of pages, so neither adopts the other's pages (middle). Hemingway has fewer pages than Shakespeare, so eventually, Hemingway adopts all of Shakespeare's pages (right) 7
Figure 1.6 Group Signature Overview 8

Figure 3.1 GJoin Protocol 17
Figure 3.2 Signature Proof of Knowledge Protocol 18

Figure 4.1 BigChainDB Model taken from [Beh18] 21
Figure 4.2 SmartChainDB 2.0 Transaction Components 22
Figure 4.3 Basic Group Signature Model 23
Figure 4.4 Group Signature Transaction Components 24
Figure 4.5 The Overview WorkFlow First Time 25
Figure 4.6 The Overview WorkFlow After First Time 26
Figure 4.7 Current API Interactions 27
Figure 4.8 Linking Transactions 28
Figure 4.9 Linking Requester Transactions 28
Figure 4.10 Requester Setup 29

CHAPTER

1 INTRODUCTION

Blockchain marketplaces have a strong foothold in mainstream computer science. A blockchain-based marketplace allows suppliers/manufacturers and clients/requesters to connect and bid for contracts when used commercially. In today's marketplace, a supply chain captures a list of steps that trace the materials from raw suppliers to the end consumers. The supply chain is an essential conceptual system that every manufactured good goes through. All manufactured goods can be traced with supply chain mapping [YG18]. For example, an LG Television supply chain requires a raw material supplier that supplies oil and a raw material supplier that supplies rare metals, many Tier 1 suppliers that build the components such as circuit boards, glass panes, buttons, then the manufacturer LG will assemble the finished product, later the distributor such as Walmart will bring the television to the customer(see Figure 1.1). Blockchain systems can provide immutability and tamper-resistant storage to supply chain transactions between different tier suppliers to end consumers, allowing reliable tracking of manufactured goods from manufacturers to end consumers.

Inherently, blockchain allows for a decentralized system that removes a centralized market bias (a single entity controlling the marketplace that provides more favorability towards one product or causing the requesters to pay a higher price). Initially seen as an integrity-ensuring system, blockchain has shortcomings in confidentiality, such as contract anonymity and actor anonymity. We focused on preserving the anonymity of the requester. The lack of requester anonymity in blockchain transactions currently makes existing requesters in the blockchain marketplace lose certain levels of competitive advantage and opens requesters up for attacks.

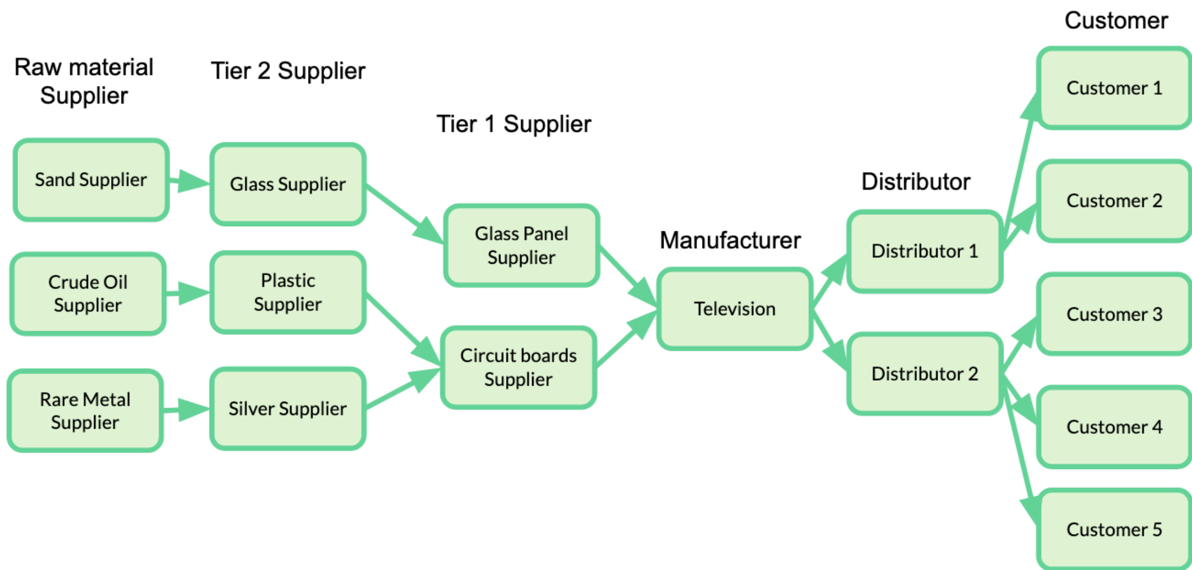


Figure 1.1 Television Supply Chain

1.1 Security Model

A security issue for existing marketplace transactions is giving away requester identity. Collaboration with other companies is considered as part of a company's competitive advantage. Losing that advantage will cause immense adverse financial effects on the requesting company. Hence our main security goal is protecting the identity of the requester.

1.1.1 Threat Model Assumptions

We assume the adversary has the power to monitor all broadcasts communications regarding transactions. The adversary can act as both requesters and suppliers or corrupt them. We assume that the adversary will not be able to corrupt the group manager. Lastly, we expect our distributed ledger to have at least 2/3 honest parties, meaning the adversary doesn't have the power to corrupt more than 1/3 of the parties participating in the blockchain.

Once adversary knows about the requester's identity, they can infer new products based on bids for suppliers. Or the adversary can act as a corrupted supplier and out-bidding other suppliers to be chosen as the supplier. Then, later on, using that positioning to provide Trojaned supplies to the requester.

1.1.2 Trust Model Assumptions

We assume the network is not compromised, and all information eventually reaches the destination at a reasonable time. We are not assuming any tampering within any network processes of hardware.

We define a trusted computing base as parts of the system that we need to run to operate correctly, and we have to trust. Our trusted computing base (TCB) is the hardware CPU and OS software. We can only guarantee the security/trust of our code and the BigchainDB system we use. The OS and the hardware that we use for our system are only trusted third parties as we can't guarantee trust of the system as we can't change the hardware or the OS code, and we assume that the markers are reputable.

1.2 Additional Conditions

While many blockchain marketplaces use smart contracts to ensure autonomous transactions, the triggering of contracts will incur extra costs on the system. The smart contracts in the manufacturer use case will make transactions inflexible as strict guidelines of smart contracts make it hard for the smart contract to be useable. Lastly, smart contracts open up another vector of attack, and if there happens to be a mistake by the programmer, the effects will be very costly [Atz17].

BigChainDB is an existing database platform with the tools to set up a decentralized marketplace. We built our decentralized marketplace system based on BigchainDB [Beh18]. We extended BigchainDB and attempted to record all transactions between Tier 1 suppliers and the manufacturer. The manufacturer's goal is to obtain something via auction, while the suppliers' goal is to fulfill the request by bidding. We identified five distinct types of actions for transactions between Tier 1 suppliers and the manufacturer: pre-request, interest, request, bidding, fulfillment.

Outlined below (and in Figure 1.2) is an example of a supply chain interaction between Tier 1 suppliers and the manufacturer Boeing. The requesters/manufacturer Boeing is a defense company who wants to cut costs by purchasing black boxes rather than designing and building their own.

1. Boeing broadcasts a pre-request for four black boxes to all Tier 1 suppliers.
2. Black box suppliers, if interested, send an interest bid.
3. Boeing sees the interest bid and sends a more detailed request to the interested suppliers.
4. Suppliers bid on the request.
5. Boeing picks the winning supplier.
6. Winning supplier fulfills the request and provides the four black boxes to Boeing.

The previous Boeing example illustrates a successful set of steps leading to the manufacturer obtaining goods from a Tier 1 supplier. However, it does not provide the defense contractor any tools to hide their identity and preserve their anonymity. We propose using group signature so that attackers cannot use the requester's identity from a transaction for competitive advantage. User identity is an asset that needs to be protected, as knowing who is purchasing what could lead to sabotaging or undercutting. Furthermore, by knowing requester identity, smaller requesters can receive adverse bias treatment by suppliers.

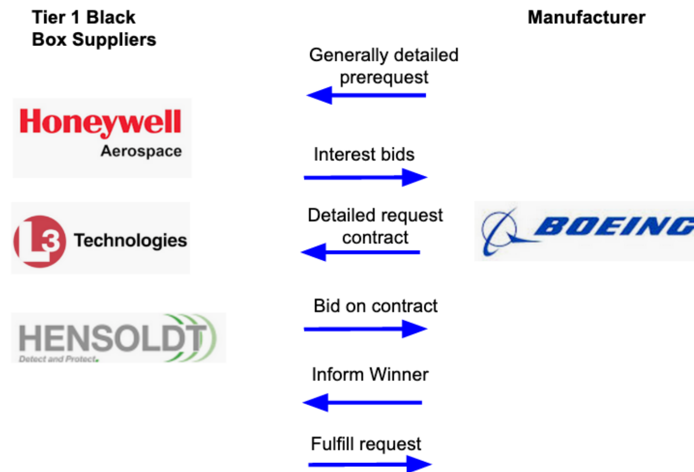


Figure 1.2 Supplier and Manufacturer Interaction

However, implementing a scheme that protects requesters' identity will cause an additional problem since the requesters and the suppliers need to be sure they are talking to the same person for the list of transactions to run. If the requesters are anonymous, suppliers cannot be sure they are talking with the same requester from one transaction to another. Therefore, in addition to keeping the requester anonymity, we would also need a way to link transactions without revealing requester identities. Lastly, while we want requester anonymity in our manufacturing marketplace space, we still need a way to remove malicious actors and hold requesters accountable for their actions. Therefore, to ensure accountability of requesters, as a last resort, the market should have the ability to identify and ban the requester.

1.3 BigchainDB

BigChainDB, as stated before, has many tools for a distributed system; those tools include multiple nodes, each containing Tendermint access location, BigChainDB Server, and MongoDB. Tendermint allows for network consensus needed for a blockchain system. While BigChainDB Server validates and adds transactions into the blockchain. MongoDB acts as data storage for each node to store the transactions. The users will interact with BigchainDB drivers by sending the transactions to the BigChainDB server. See figure 1.3

1.3.1 Blockchain

When we hear about blockchains, the first thing that comes to mind is cryptographic currencies such as Bitcoins[Ner21] and Enthereum[Niz19]. The two most crucial idea that those currencies gain from blockchain is decentralization and immutability. A blockchain is a distributed network, has a consensus system, and block immutability. Blockchain is a distributed network, meaning there is

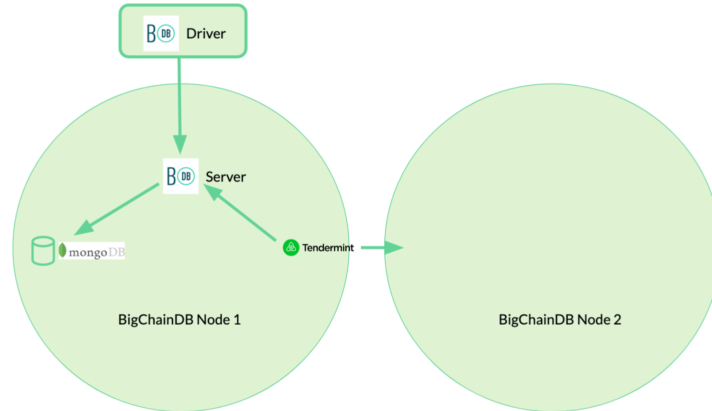


Figure 1.3 BigChainDB nodes

no central entity responsible for certifying the network's correct blockchain. [Has19] Blockchain consensus occurs when a "fair" amount of ledgers agree on the longest chain, thus, approving that chain as the correct blockchain. All blockchain blocks are immutable because an old block's relationship to a new block depends on one-way functions/hashes.

1.3.1.1 Chain of Blocks

A blockchain is created by having multiple blocks. Using hashes, these blocks each point to a single existing block. [Sam16] The preimage resistance property of hashes ensures that tampering with the blocks' relationships will be computationally infeasible. The collision resistance aspect of hashes ensures that hashing different preimages will always yield different results—these two properties of hashes making the blocks "chain" together with one after another.

To explain the chaining process, assume there exists Block A, Block B, and Block C. Let Block A contain Data_A and be the genesis block. Let Block B contain Data_B and hash of Data_A (called Hash_B). Lastly, Block C contains Data_C and hash of Data_B plus Hash_B. Blockchain makes modifying blocks difficult since if Data_B is modified in any way, the association between Block B and Block C will break. If Data_A has changed in any way, the association between Block A and Block B will break.

1.3.1.2 Consensus

A blockchain system is decentralized, which means there is no centralized entity to ensure that multiple independent parties agree on the same blockchain. Three conditions must be met agreement, validity, and termination [Gra20]. Generally, a consensus is reached in a blockchain system once a majority of parties agree on the longest blockchain. In a conflict such as multiple blockchains tied for the longest, parties will wait to accept either one until the longest blockchain appears.

To further explain blockchain systems' consensus (as shown in Figure 1.5), consider that there are two scribes: Shakespeare and Hemingway. Their goal is to contribute pages to a book. Each time

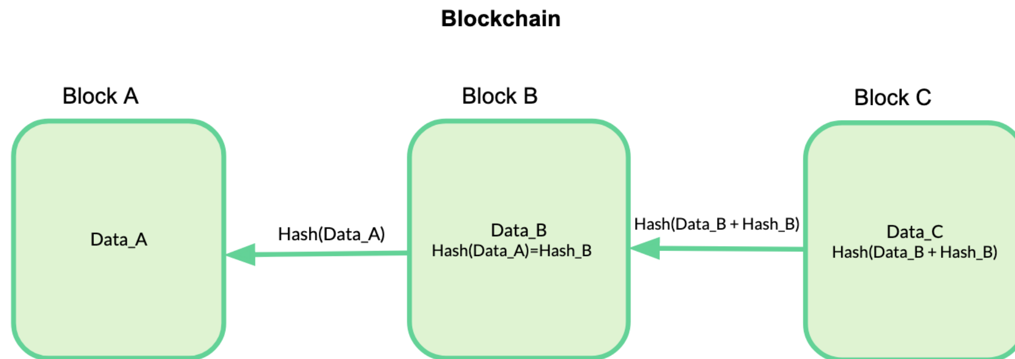


Figure 1.4 Simple Blockchain

Hemingway finishes a new page, he will show Shakespeare of his copy of the book with the new page. If Shakespeare has fewer pages in Shakespeare's book, he will adopt the new book. However, if Shakespeare has the same number of pages in Shakespeare's book, Shakespeare will wait and do nothing. If Shakespeare has more pages in Shakespeare's book, Shakespeare will not adopt the new book and attempt to show Hemingway Shakespeare's existing book, and Hemingway will copy Shakespeare's book instead.

1.4 Group Signature

By extending BigChainDB to include group signature, we provide confidentiality for requesters missing from many blockchain marketplaces. Utilizing group signature, the only information that bidders obtained about the requester's identity is the requester's group.

The schema from Bichsel and Camenisch [Bic10] displays the group signature that provides the opening function and does not use encryption. We base our group signature scheme on this Bichsel and Camenisch schema (referred to as BC schema in this paper). The key generating steps are as follows: a single group manager will generate a group manager secret key (gmsk) and group public key (gpk), the requester will generate their user secret key (usk) and user public key (upk), and the requester will then generate a group secret key (gsk) with the help of group manager. During the supply chain's requesting phase, the requester should sign their messages and be verified by any individual with the group public key. See Figure 1.6. Lastly, in extreme cases, only the group manager may open the signature to determine the user's identity if there is a problem.

The BC schema is logically sound, and we aim to expand upon it to create additional functions from existing libraries. Since group signature implementations for SmartChainDB are currently non-existent.

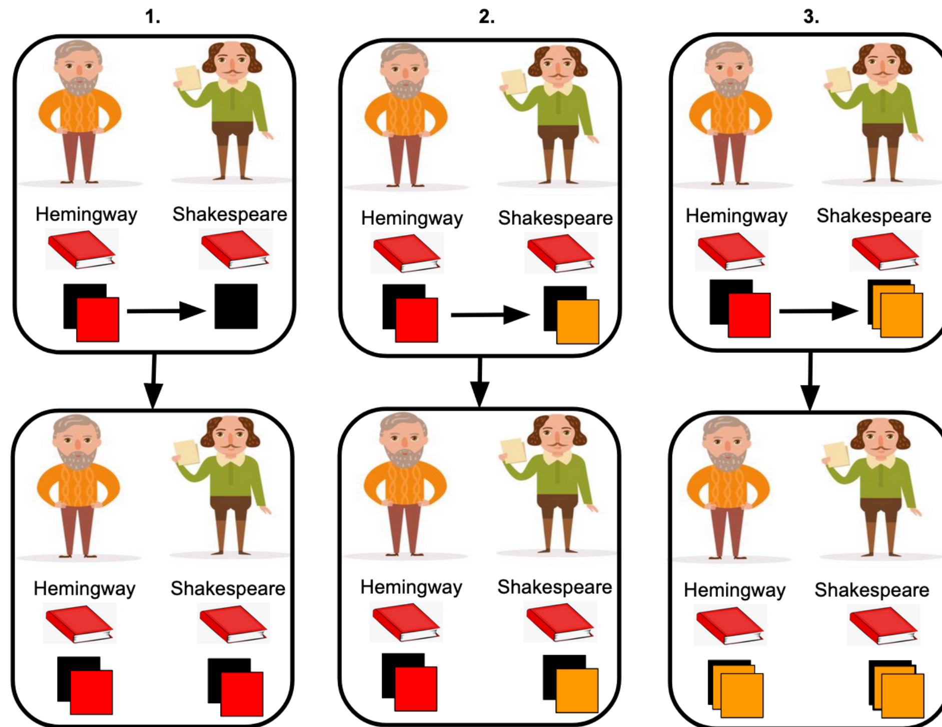


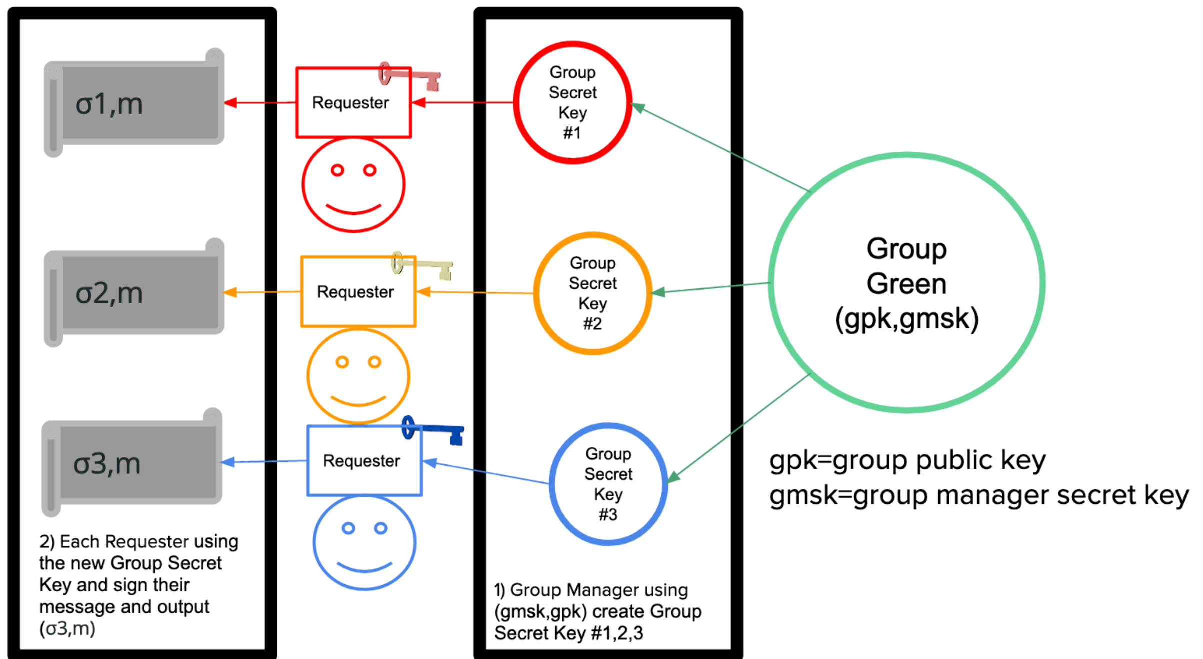
Figure 1.5 Hemingway has more pages in his book, so Shakespeare adopts all of Hemingway's pages (left). Both Hemingway and Shakespeare have the same number of pages, so neither adopts the other's pages (middle). Hemingway has fewer pages than Shakespeare, so eventually, Hemingway adopts all of Shakespeare's pages (right)

1.5 Linkable Transactions

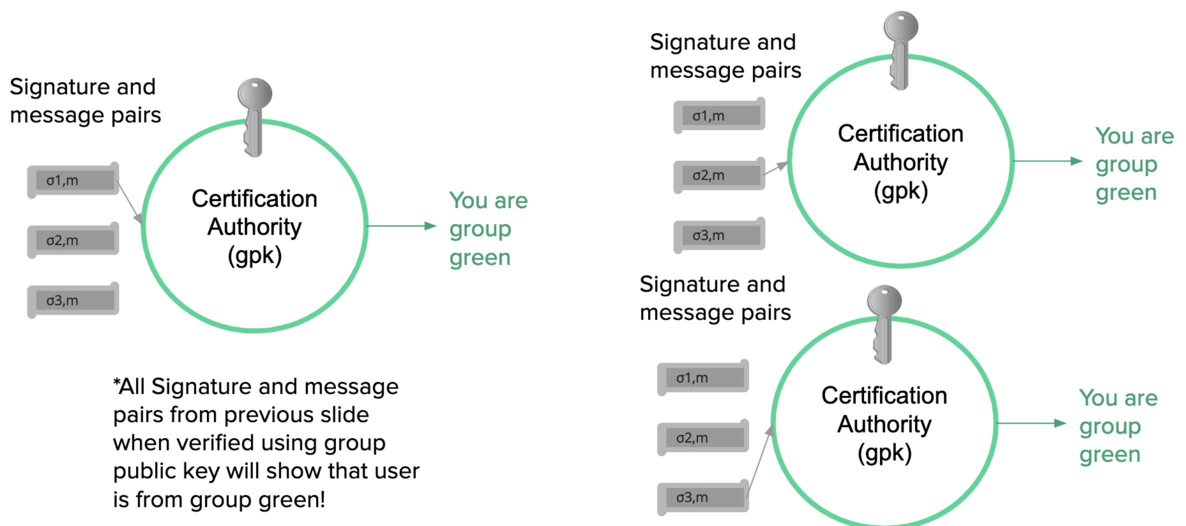
While group signature allows a requester to gain anonymity in a group, requesters still must be accountable for their previous transactions. Otherwise, any malicious requester can pretend to be the original requester within their group and send a fake request. Hence, we utilized hash and preimage pairs as the link between multiple transactions for each requester. By adding a hash of a random preimage to every requester transaction, we ensure that prerequest, request, and fulfillment transactions are from the same individual in a group as long as the latter transaction contains the previous transaction's preimage. It is computationally impossible for another group member to duplicate the preimage given the hash due to hashes' preimage-resistant property.

1.6 Contributions

This thesis will analyze a working group signature scheme for a blockchain database system (SmartChainDB). The tenants of our implementation are to be functional, practical, and well-integrated into the SmartChainDB system. The paper will start with a review of existing information on group signatures. Then, we will discuss anonymity preservation and non-repudiation quality of the scheme. Lastly,



(a) Group Signature Signing



(b) Group Signature Verifying All Signatures

Figure 1.6 Group Signature Overview

we will explain our coding and design decisions used to run the scheme.
In this paper, we make the following contributions:

1. Create a group signature with linkable transactions scheme

2. Implement a group signature linkable transactions into SmartChainDB

The remainder of this paper proceeds as follows. Chapter 1 includes a discussion of notable related works. Chapter 2 provides a list of assumptions and describes the design of our group signature. Chapter 3 gives an overview of our implementation attempts. Chapter 4 evaluates our solution and results. Chapter 5 concludes and discusses future directions for this research.

CHAPTER

2 BACKGROUND AND RELATED WORKS

2.1 Blockchain

2.1.1 Blockchain Conception

The first instance of the idea of blockchain was introduced by Haber [Hab91] in 1991. The idea of blockchain later found its way to the field of cryptocurrencies popularized to the mainstream by Bitcoins [Nak08]. Furthermore, the blockchain idea was later built upon by Kosba [Kos15] to add security to contracts and create smart contracts. These "smart contracts" are implemented to create both a decentralized and an automated marketplace of services for the Internet of Things [Chr16] [Lav18]. Today Ethereum (cryptocurrency) [Luu16] relies on smart contracts.

2.1.2 Blockchain Database

Blockchain database [Swa15], as the name implies, combines aspects of blockchain and database. Blockchain properties such as decentralization, Byzantine fault tolerance, owner-controlled assets, and immutability ensure various asset transactions are in a decentralized system can be saved forever. The aspects shared with databases, such as high transaction rate, low latency, and indexing and querying of structured data, ensure transaction speed and overall usability. Blockchain databases such as BigChainDB 2.0 system has a network of nodes comprised of Tendermint, BigChainDB, and MongoDB. Each BigChainDB node uses Tendermint to connect to other nodes. After a blockchain adds a block, having nodes reach an agreement becomes an asynchronous consensus problem. Tendermint [Buc16] [Kwo14] provides asynchronous byzantine fault tolerance of 1/3 where one-third of the nodes could be compromised, and nodes still will eventually reach consensus. BigChainDB

within a node does have properties to modify the blockchain similar to how Bitcoin's transaction ledger's backbone protocol properties [**Garay2014TheBB**].

2.1.3 Network Fault Tolerance

Like the bitcoin system, where we assume the network has asynchronous communication over the interconnects, BigChainDB has asynchronous connections within its Tendermint network. In an ideal network, all network communication is instant, and all nodes are part of a synchronous byzantine system. In the past, only synchronous and partially synchronous networks were guaranteed to end after 3t+1 rounds, while asynchronous networks couldn't be guaranteed to end [Dwo84]. More recently, Abraham proposed an asynchronous byzantine system where the system's threshold tolerance could be optimized to 2f+1 [Abr17]. Cachin [Cac00] was able to propose a cryptographic scheme that assures Byzantine Fault Tolerance of 1/3 (when adversaries control up to 1/3 of the network, while 2/3 of the nodes are honest) on an asynchronous system that is guaranteed to end at r+1 round, where round r is when one node in the network has decided. Dolev [Dol83] defines t+1 as the lower bound, meaning the minimum round needed to reach a consensus, where t is the number of processors. When modifying Tendermint in SmartChain and adding group signatures, we need to preserve the Byzantine Fault Tolerance.

2.1.4 Blockchain Confidentiality

Blockchain inherently doesn't protect the user identity; its original role was to preserve an unchanging and complete transaction history. Privacy has been a concern within the blockchain throughout the years. There are two primary schools of thought to protect confidentiality: hide the transaction itself or hide the transaction participants.

2.2 Transaction Anonymity

You can choose to hide some parts or all parts of the transaction. For instance, ZeroCoin hides only the blockchain transaction cost via a committing phase and an opening phase. The system will generate a serial number and a coin during a ZeroCoin committing phase during the minting period. During the spending/verifying period of a ZeroCoin opening phase, the system, on behalf of the spender, opens the commitment, using a non-interactive zero-knowledge and the serial number. [Mie13] Additionally, to preserve transaction anonymity, we can also omit the request's vital details in a pre-request. A pre-request could only summarize the request and pass a little bit of the detail to generate interest. And once some suppliers show interest, the requester can send a more detailed request.

2.3 Requester Anonymity

Instead of focusing on hiding the transactions, the other research looks into preserving user identity privacy. Some existing tools aim to achieve user anonymity via a mixer approach. Decentralized mixers such as CoinShuffle (or Coinparty, XIM, CoinShuffle++) [Ber19] will combine multiple transactions into a huge CoinShuffle transaction that can trace back anyone that has participated in the CoinShuffle transaction. Thus transactions from the sender cannot be linked to the transactions to the receiver, assuming there are multiple unique senders and receivers.

Another way that some achieve user anonymity is by using the crowd approach. By masking users to look alike, they become indistinguishable, and no one can trace the data back to the users. And by adding more users into the anonymity set, the better the anonymity. Ring signature and Group signature use this crowd approach, where users are part of a ring/group, and their actions are indistinguishable from other members in the ring/group.

2.4 Ring Signature

The idea of a ring signature is to have multiple signing parties, each participating in the signature process until everyone in the ring has helped the signing process. Later on, anyone with the public keys can verify the signature as coming from that particular ring. Ring signature can provide linkability using the same list of public keys of all ring members. [Liu05] [Goo19]

One advantage of the ring signature scheme is removing group manager setup complexity and group manager overhead, as a ring signature scheme never reveals users' identities. Consequentially, without the ability to keep each ring signature's accountability, members of a ring signature will have difficulty taking out specific malicious signing group members. There can never be non-repudiation, for no one could ever confirm the user's true identity.

Another disadvantage of the ring signature is when the ring adds a new member. All previous signatures will not function properly since the verification function of ring signatures uses the public keys of all members in the ring at the moment of signing. Adding additional members messes up so members with current lists of public keys can no longer verify past signatures and mess up ring signature's ability to link the two signatures as the list of public keys used for linking are different. We also see the scalability of the ring signatures functions becomes more complex than the scalability of group signatures as more people join the ring/group.

2.5 Group Signature

Group signature is the idea of anonymity of an individual within a group combined with the non-repudiation of an individual. This paper seeks to identify and build a group signatures scheme that uses suitable for an entity to publicize a statement on behalf of a group of entities without revealing its identity within the NSF project SmartChainDB.

For a time, the RSA-based Short Group Signature scheme was thought as sufficient and secure [Bon04] [Ish19]. But, later bilinear maps [Cam04] [Bon02] based group signature schemes replaced RSA based group signature schemes due to computational efficiency.

More recently, Bichsel & Camenisch [Bic10] offered fast bilinear mapping and re-randomization-based group signature scheme. Pointcheval & Sanders[Poi15] offers a solid group signature scheme implementation based on the Bichsel & Camenisch scheme that is efficient and correct. We will be implementing the existing Pointcheval & Sanders scheme [Poi15].

2.5.1 Linkable Group Signature in Asynchronous Blockchain Network

Recently few research has dove into the idea of linkable group signatures in asynchronous blockchain networks. An idea presented by Zhang [Zha19] shows that it is possible to construct a linkable group signature scheme for payers of cryptocurrencies using linear encryption group managers. However, Zhang's linkability only prevents double-spending using an amount parameter. Since no applicable linkable group signature fits our blockchain marketplace, we will link the group signature separate from the existing Pointcheval & Sanders scheme.

CHAPTER

3

METHODS

We implement the Pointcheval & Sanders signature, which is based on the Bichsel & Camenisch Group Signature Scheme [Bic10] due to its efficiency and ease of implementation as described in PS paper [Poi15].

3.1 Assumptions and Useful Methods

3.1.1 Bilinear Mapping Assumption

Based on existing Billinear Signatures [Cam04], we assume for all G1 and G2 where G1 and G2 are multiplicative cyclic group of the same prime order p and we assume e (e is the equivalent of G1 x G2 → GT) is a permissible Bilinear Map if:

1) Bilinear Property: For all g1∈G1, g2∈G2, and for all a∈Z, b∈Z, $e(g1^a, g2^b) = e(g1, g2)^{ab}$

2) Non-degenerate Property: For all g1∈G1, g2∈G2, e(g1, g2)!= 1 and is a generator of GT

3) Computable Property: There exists an efficient algorithm for computing e(g1, g2) for any g1∈G1, g2∈G2

3.1.2 Re-randomization

Re-randomizable property:
Given valid signature $(a, b, c) \in G^3$ on a message m. Signature $(a^r, b^r, c^r) \in G^3$ will also be valid for

any $r \in Z_q$ [Bic10]

3.1.3 Zero Knowledge Proof

Sigma Protocol:

Sigma Protocol is used to prove discrete logarithms such as the preimages of group homomorphisms. We obtain the Signature Proof of Knowledge by applying the Fiat-Shamir heuristic to the sigma protocol. Such that a Signature Proof of Knowledge on a message $m\epsilon\{0,1\}^*$ is represented by $\Sigma \leftarrow SPK\{(x) : y = \phi(x)\}(m)$ [Bic10]

3.2 Pointcheval & Sanders Group Signature Scheme

The Pointcheval & Sanders Group Signature Scheme [Poi15], similar to the previous basic group signatures, has three parties (User, Group Manager, Certification Authority) and shares the basic ability to Verify and Sign. We Let $g_1 \in G_1$ and $g_2 \in G_2$, where G_1 & G_2 are cyclic group of order q prime and e is the bilinear mapping of (g_1, g_2). The scheme starts with two setup functions Group Manager running $GSetup$ and User plus CA running $PKIJoin$. Then $GJoin$ is run by the User and Group Manager. Afterward, the User performs $GSign$ to create a signature. Lastly $GVerify$, $GOpen$ can be run after $GSign$.

3.2.1 GSetup & PKIJoin

$GSetup$ and $PKIJoin$ are setup steps, from Pointcheval & Sanders [Poi15], that provide parameters for the $GJoin$. They can both be run independently of the other. For $GSetup$, the Group Manager's goal is to generate a GPK (group public key) and $GMSK$ (group master secret key) by taking in 1^η as the security parameter. For $PKIJoin$, the goal is for the User to generate a $USK[i]$ (user secret key) and $UPK[i]$ (user public key) using a DSKeyGen (digital signature generator) and afterward, letting CA (Certification Authority) store $UPK[i]$.

GSetup(1^η)**:**

Inputs: 1^η

Output: $GPK, GMSK$

1. Group Manager: $x \leftarrow Z_p, y \leftarrow Z_p$
2. Group Manager: $X_2 \leftarrow g_2^x \; Y_2 \leftarrow g_2^y$
3. Group Manager: $GPK \leftarrow (g_2, X_2, Y_2)$, $GMSK \leftarrow (x, y)$

PKIJoin(i):

Inputs: i
Output: $USK[i], UPK[i]$

1. User and CA: $(UPK[i], USK[i]) \leftarrow DSKeyGen()$

3.2.2 GJoin

$GJoin$ function, from Pointcheval & Sanders paper [Poi15], utilizes the results of $PKIJoin$ & $GSetup$ and creates the secret keys and public keys that will allow users to sign as a group member and verify group signatures created by that group member. $GJoin$ function requires both User and Group Manager to participate in a series of interactions see Figure 3.1. As stated before the results of $GSetup$ and $PKIJoin$ $(USK[i], UPK[i], GMSK, GPK)$ are used in order to generate $GSK[i]$ (user group secret key) for the User to keep and $Reg[i]$ (registration information) for the Group Manager. $GSK[i]$ will allow users to sign their messages, while$Reg[i]$ will allow Group Managers to identify the message's true signer and not just the group that the signer belongs to.

GJoin($i, USK[i], UPK[i], GMSK, GPK$):

Inputs: $i, USK[i], UPK[i], GMSK, GPK$
Output: $GSK[i], Reg[i]$

1. User: generates $ski \leftarrow Z_p$, $(\tau, \tau_2) \leftarrow (g^{ski}, Y_2^{ski})$, $\eta \leftarrow sign(USK[i], \tau)$
2. User: sends (τ, τ_2, η) to Group Manager
3. Group Manager: upon receiving (τ, τ_2, η), verifies that $\hat{e}(\tau, Y_2) = \hat{e}(g, \tau_2)$ and verifies η with τ and $UPK[i]$(received from Cerification Authority). Executes a Sigma Protocol with User showing that User knows ski. Then Group Manager generates $u \leftarrow Z_p$ and create $\sigma \leftarrow (\sigma_1, \sigma_2) \leftarrow (g^u, (g^x \cdot (\tau)^y)^u)$. Lastly stores $[i, \tau, \eta, \tau_2]$ as secret register
4. Group Manager: sends (σ) to the User
5. User: upon receiving (σ), User saves $(ski, \sigma, \hat{e}(\sigma_1, Y_2))$ as $GSK[i]$

3.2.3 GSign

Once users obtain $GSK[i]$ (individual group secret key), they can sign their message m using the $GSign$ function. The User will have the ability to send σ (the signature) and m (the message) as the output. Following Pointcheval & Sanders paper [Poi15] we utilized Re-randomization and bilinear

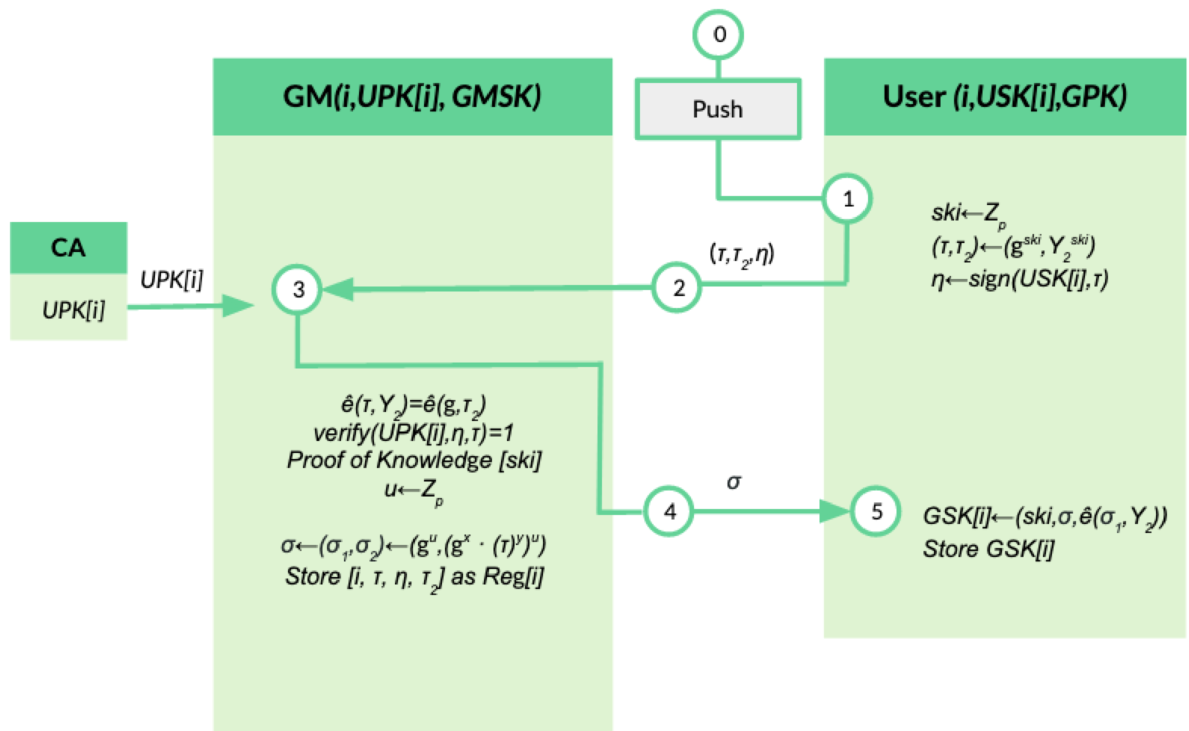


Figure 3.1 GJoin Protocol

mapping in the $GSign$.

GSign($GSK[i], m$)**:**

Inputs: $(ski, \sigma, \hat{e}(\sigma_1, Y_2)), m$
Output: μ, m

1. User: Re-randomization - generate $t \leftarrow Z_p$. Then caluates $(\sigma_1', \sigma_2') \leftarrow (\sigma_1^t, \sigma_2^t)$
2. User: Computes Signature Proof of Knowledge (Σ) for ski see Figure 3.2. User first calculates $k \leftarrow Z_p$, computes $A = \hat{e}(\sigma_1', Y_2)^k = \hat{e}(\sigma_1, Y_2)^{k \cdot t}$, then compute $c = H(\sigma_1', \sigma_2', \hat{e}(\sigma_1, Y_2)^{k \cdot t}, m)$, and then computes $s \leftarrow k + c \cdot ski$
3. User: Lastly outputs μ and m, where $\mu \leftarrow (\sigma_1', \sigma_2', c, s)$ and where c is the challenge and s is the response.

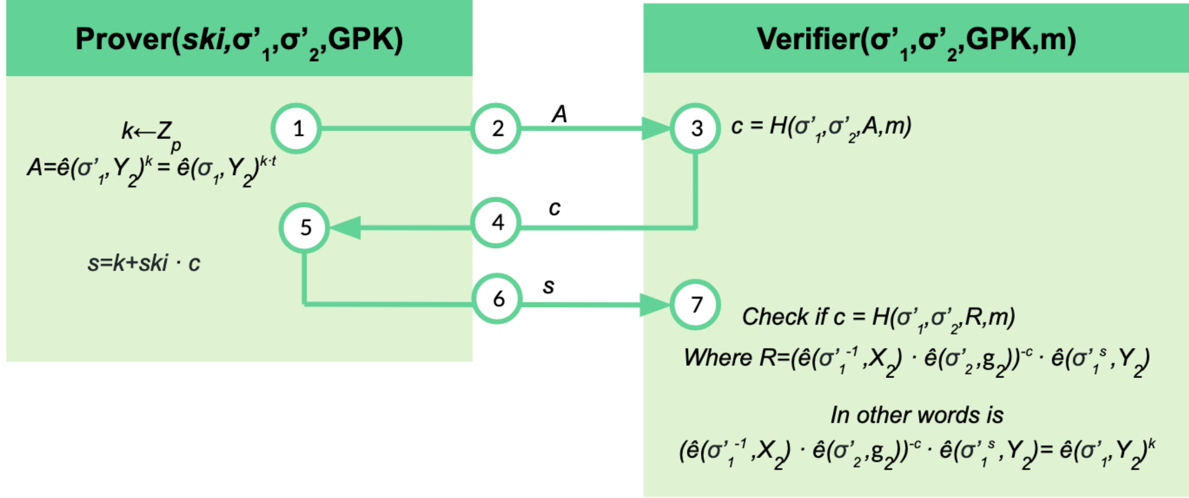


Figure 3.2 Signature Proof of Knowledge Protocol

3.2.4 GVerify

After a User has signed a message using $GSign$, any other User can verify the signature using Group Public Key. This $GVerify$ makes sure the signer of the signature is a group member.

GVerify(GPK, μ, m)**:**

Inputs: GPK, μ, m where $\mu \leftarrow (\sigma_1', \sigma_2', c, s)$ and $GPK \leftarrow (g_2, X_2, Y_2)$
Output: 0/1

1. Verifier: Compute $R \leftarrow (\hat{e}(\sigma_1'^{-1}, X_2) \cdot \hat{e}(\sigma_2', g_2))^{-c} \cdot \hat{e}(\sigma_1'^{s}, Y_2)$
2. Verifier: check $c = H(\sigma_1', \sigma_2', R, s)$ to verify the signature of knowledge, this works since $(\hat{e}(\sigma_1'^{-1}, X_2) \cdot \hat{e}(\sigma_2', g_2))^{-c} \cdot \hat{e}(\sigma_1'^{s}, Y_2) = \hat{e}(\sigma_1', Y_2)^k$ is correct
3. If the signature of knowledge is valid, then the verifier returns 1, else 0

3.2.5 GOpen

Once the contract is ready to be made, the Group Manager will reveal the signer's identity using the registration information $Reg[i]$. However, in preserving the group members' identity, $GOpen$ will not be implemented in our system for now. For completeness sake, we will still mention $GOpen$.

GOpen$(GMSK, \mu, m, Reg[i])$**:**

Inputs: $GMSK, \mu, m, [i, \tau, \eta, \tau_2]$ where $\mu \leftarrow (\sigma_1', \sigma_2', c, s)$
Output: $i(userid), \pi$

1. Group Manager: Given $\mu = (\sigma_1', \sigma_2', c, s)$ and m for each element in Reg, using $Reg[i] = [i, \tau, \eta, \tau_2]$, check if $\hat{e}(\sigma_2', g_2) \cdot \hat{e}(\sigma_1', X_2)^{-1} = \hat{e}(\sigma_1', \tau_2)$
2. Group Manager: outputs corresponding (i, τ, η) and SPK (Signature Proof of Knowledge) for τ_2

CHAPTER

4 IMPLEMENTATION

Initially, BigchainDB driver utilizes ed25519 or the Edwards-curve Digital Signature Algorithm for its public key signatures and has the inputs and outputs of the transaction to deal with the signatures. Following the original structure, we also put our ecliptical curve-based group signature and group public key in the inputs and outputs of the transaction. Then we modify the BigchainDB server the schema allow for group signatures. Lastly, we offer linkability in the form of hashes and preimages. The linkable hashes and primages are stored in the metadata under RID and nonce of the Prerequest, Request, and Fulfillment transactions.

4.1 Overview

Group signature scheme provides anonymity to individual signers in a group while ensuring non-repudiation of the individual. Abundant research exists for group signatures. However, an area seldom touched upon is the implementation of group signatures into blockchain systems. This paper seeks to identify and build a group signature scheme suitable for an entity to publicize a statement on behalf of a group of entities without revealing their identity, using the NSF project SmartChainDB.

4.1.1 SmartChainDB Model

SmartChainDB is based on BigChainDB, an open-source blockchain database. BigChainDB has various nodes, with each node containing Tendermint access location, BigChainDB Server, and MongoDB (See Figure 4.1). Tendermint allows for network consensus protocol that ensures a byzan-

tine fault tolerance of 1/3 (1/3 of nodes could be malicious, and the protocol would still work). BigChainDB Server takes care of the block building and chaining. Lastly, MongoDB serves as local storage for the blockchains. [Beh18]

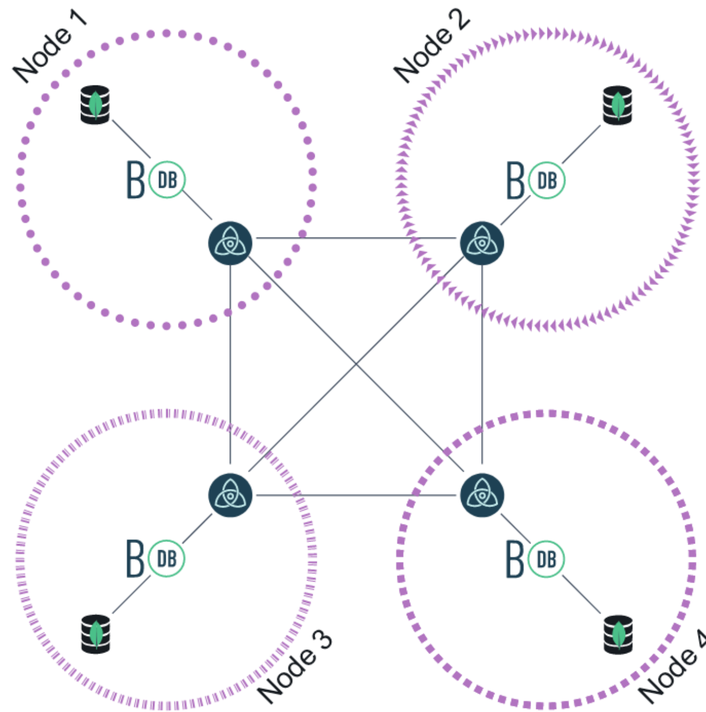


Figure 4.1 BigChainDB Model taken from [Beh18]

Within BigChainDB, each JSON object (with id, version, inputs, outputs, operation, asset, and metadata fields) represents a transaction. The id field is the unique id of the transaction. The version field indicates the BigChainDB version used, with the latest at this time being 2.0. The inputs contain the list of transactions, which indicate whether a previous output is spent or transferred (assuming the fulfillment is valid). Outputs contain the list of transaction outputs containing crypto-conditions that, once satisfied, the asset can transfer. Operation determines whether the type of transaction is to CREATE, TRANSFER, VALIDATOR_ELECTION, CHAIN_MIGRATION_ELECTION, or VOTE. The asset field in the transaction, depending on the operation, contains the transferred asset. Metadata is a field that contains any additional information in JSON form. See Figure 4.2 for basic transaction anatomy.

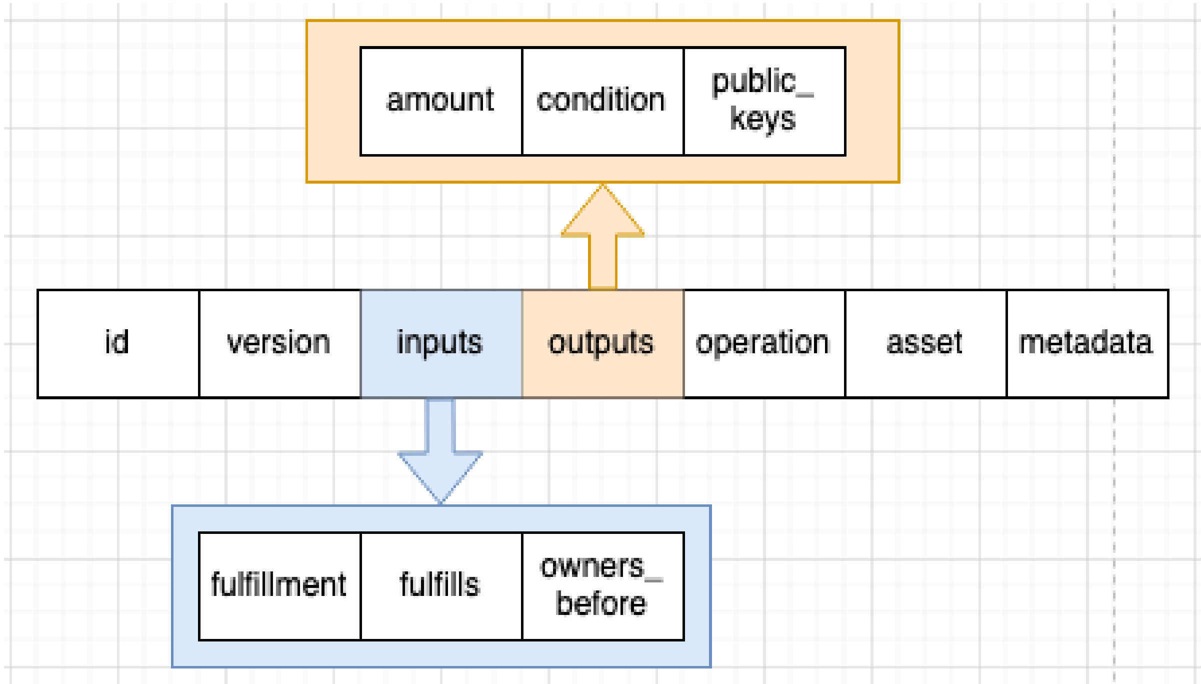


Figure 4.2 SmartChainDB 2.0 Transaction Components

4.1.2 Basic Group Signature Model

A basic Group Signature Model includes three parties: the Group Manager, Verifier, and Group Member. I will now describe this model by following Steps 0-5 in Figure 4.3. In Step 0, the Group Manager starts by creating the group, generating a GMSK (group manager secret key), and generating a GPK (group public key) for that group. In Step 1, the Group Member requests to join a group by sending a Join(i) request (where i is the Group Member id) to the Group Manager. In Step 2, the Group Manager then generates a GSK[i] (Group Secret Key for Group Member i). In Step 3, the Group Member can now sign a message m using the GSK[i]. Then, the Group Member will send signature σ and message m to the Verifier. When the Verifier receives the σ and m from Group Member and GPK from the Group Manager, at Step 4, they can verify the signer's group using Verify(σ,m, GPK). Lastly, in Step 5, when the Verifier wants to figure out the signer's identity, the Verifier will request the Group Manager to perform an Open(σ,m, GPK, GMSK), which will reveal the signer's true identity.

4.2 Libraries

4.2.1 Charm

The initial library we explored but did not end up using was the Charm Python library. The Charm library provides pre-built schemes, such as Digital Signature Schemes and Sigma Protocol. Another benefit of using the Charm Python library was that BigChainDB is native to Python. Therefore, using a Python library decreases the need for more space since BigChainDB can reuse more libraries. The

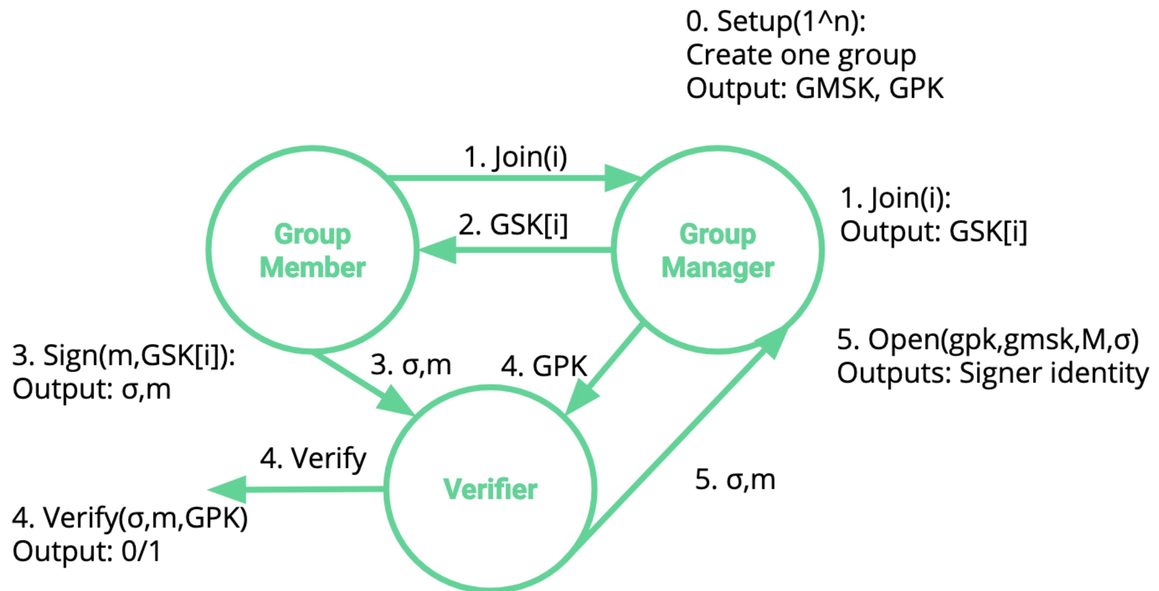


Figure 4.3 Basic Group Signature Model

Charm library also had increased speed due to the absence of overhead for language swapping. However, we did not use Charm for our project due to several workability issues. The typing of variables was ambiguous in parts of the Charm library, making it difficult to choose the correct functions to use in our project. Even though there was documentation of the helper functions that could help build group signature, examples of its usage were scarce due to the relative obscurity.

4.2.2 Hyperledger Ursa

In the end, we successfully implemented the group signature using Hyperledger Ursa library as the base. Hyperledger is an open-source, high credibility, and high-quality repository for blockchain and related technologies. Hyperledger Ursa is a Rust-based cryptographic library that provides all the necessary functions and classes for this project. These include, but are not limited to, Digital Signatures, Sigma Protocols, and the Pointcheval-Sanders [Poi15] based signature scheme, which uses Bichsel and Camenisch [Bic10] assumptions. Our project aims to make functional the Bichsel and Camenisch scheme's conceptual works, and these Hyperledger Ursa components act as great tools to build our group signature scheme. The Pointcheval-Saunders scheme also includes bilinear mapping and re-randomization. The Hyperledger Ursa library also contains some of the Bichsel and Camenisch scheme functions like: Setup, KeyGen, DSSign, and DSVerify. Another benefit to using this Rust library, since it borrows the C Apache Milagro Cryptographic Library, is that using Rust will be much faster than using Python to write the Bichsel and Camenisch scheme.

4.3 Integrating Blockchain and Group Signature

4.3.1 Group Signature Transaction Structure

To indicate that the driver uses group signature, we included a "group signature" type option for "outputs.condition.details.type" in the transaction. We preserve the previous ed25519 transaction schema as much as we can by inserting group signature in the "input.fulfillment" field in the transaction and group public key in the "input.owners_before", "output.condition.details.public_-key" and "output.public_keys". Lastly, we implemented the linkability aspect in the "metadata" section of the transaction with the tag "RID" for the ID, "hash" for the hash, and "previous_nonce" for the nonce (figure 4.4).

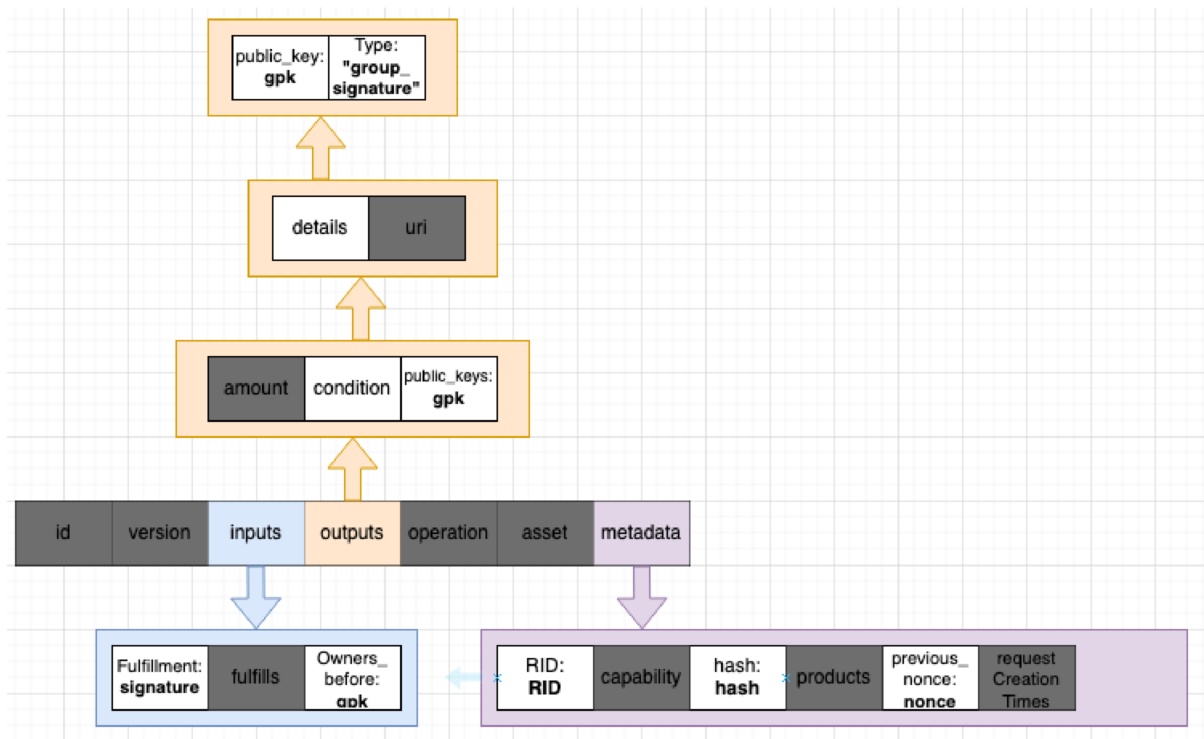


Figure 4.4 Group Signature Transaction Components

4.3.2 System Components

Within Hyperledger Ursa, there is an existing Short Randomizable signatures folder which is based on the Pointcheval & Sanders paper [Poi15]. This folder contains both simple digital signature signing and proof of knowledge functions, which created the necessary backbones of GSign and GVerify based on the Bichsel and Camenisch scheme [Bic10]. The Setup, Signing, and Verifying of the group signature are all done in Rust. Lastly, Python is used as a wrapper to call the Rust cryptographic

functions as a service. We set the Cherry.py server to take the role of Group Manager as a service.

The necessary steps for a requester utilizing a group signature are as follows. We must assume that the Group Manager Service has already set up the groups and has created a list of those groups with the corresponding group public key. We also assume that Requesters have already set up their user public and secret keys.

1. Requester sends a request to join a group within the Group Manager Service. The request will contain their public key and the id of the group they want to join.
2. Group Manager Service, our Cherrypy server, will run the Rust function GJoin and return the requester's unique group secret key.
3. After obtaining the group secret key, the requester can start the signing process by sending a message and preimage to their trusted local software.
4. The trusted local software will handle the GSinging of the requester's message and preimage using the group secret key.

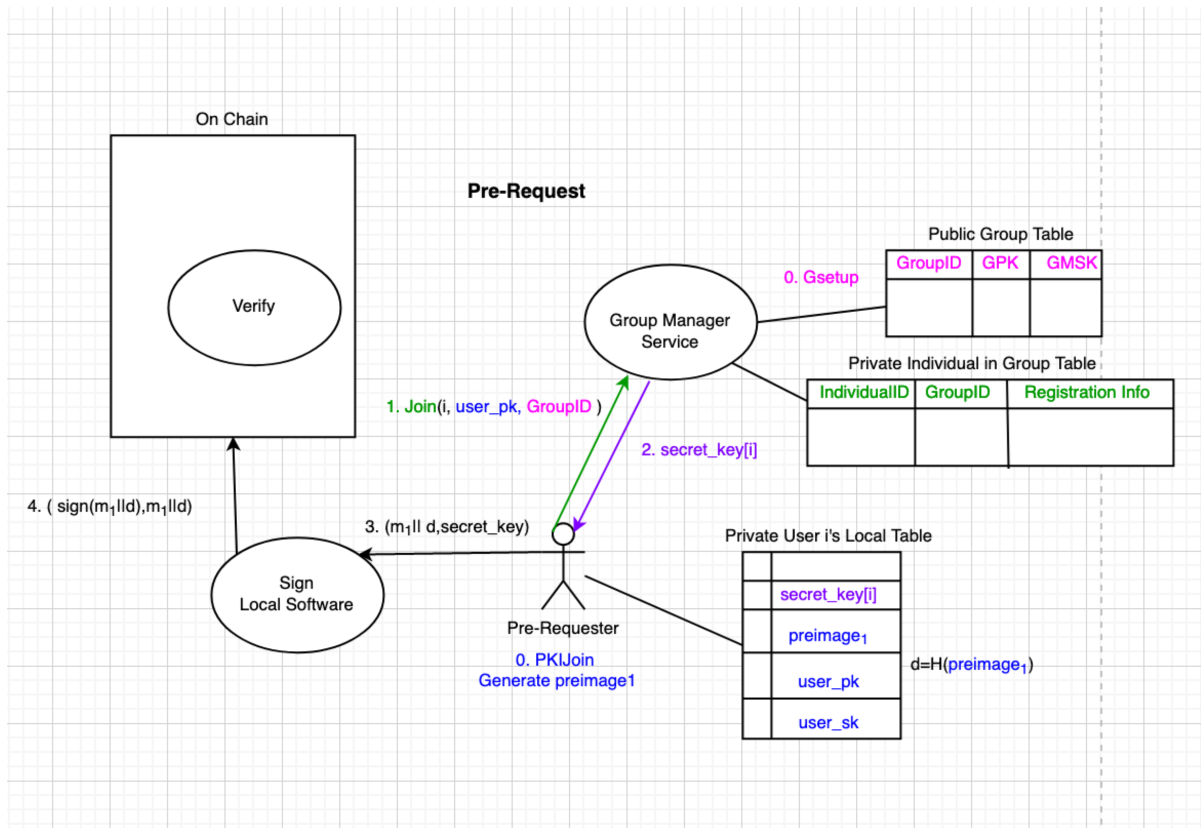


Figure 4.5 The Overview WorkFlow First Time

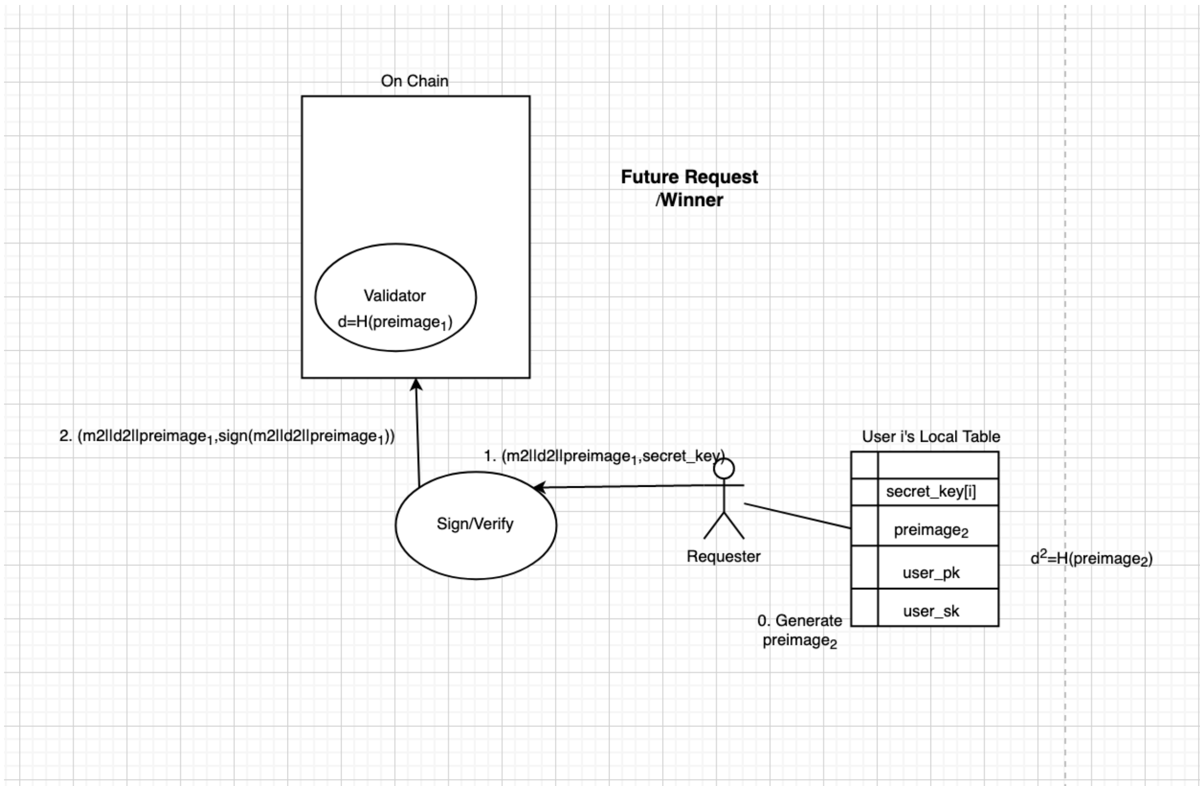


Figure 4.6 The Overview WorkFlow After First Time

When the requester wants to sign additional transactions, Steps 1 and 2 are not needed as the requester can reuse the group secret key to sign. Figure 4.7 gives a better idea of the functional interactions between applications used to build the system. Three of the applications: BigChainDB Server, MongoDB, and Group Manager Service, operate from port numbers 9984, 27017, and 8080, respectively.

4.3.3 Linkability with hashes

We offer linkability in the form of hashes and preimages. The hash's role is to ensure we can connect each transaction made by the same group member without revealing their identity. While we later reveal the preimage r to establish the connection with the hash. Figure 4.8 illustrates the hash offering an additional link to the blockchain structure. We are allowing D to link with A as long as D can provide a valid r. A valid r would satisfy $H(requestID_A + r) = H_A$, where both $requestID_A$ and H_A are known, but r is a secret.

The requestID (RID) is a unique identifier that links all subsequent transactions sent by the requester. A random 128-bit generator calculates RID and r (a nonce). We link all requester transactions by having the prerequest transaction include RID and hash0, having the request transaction include RID, r0, and hash1, and having the fulfillment transaction include RID, r1. The RID is the shortcut without hashing that will allow anyone to see which transactions are related. While r0 will

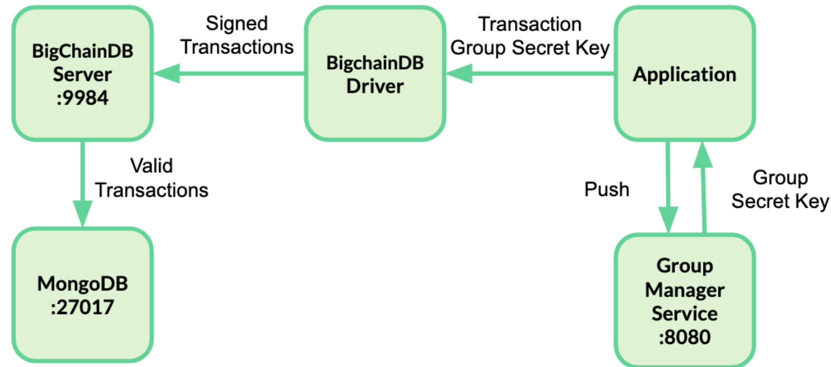


Figure 4.7 Current API Interactions

connect the request transactions with the prerequest transactions, r1 will connect the fulfillment transactions with the request transactions.

4.3.4 Requester interaction with the system

The application will talk to the group manager on the requester's behalf. A high-level setup for client interaction while using the group signature is seen in Figure 4.10. Our project's premise is to have an application that allows the requester to speak to the Group Manager Services and group databases. The requester will use the application system in this way:

1. Requester sends a push to see a list of groups.
2. System requests a list of potential groups from MongoDB.
3. MongoDB returns the list of groups to the application system.
4. The application system returns the list of potential groups to the requester. The system gives a test r (randomly generated).
5. Requester shows that they want to join a group by giving their user_id, group_id, and signature. They must sign the r using the existing secret key.
6. System verifies the signature is correct using the public key.
7. System tells the group manager to add requester to group
8. Group Manager returns the group secret key to the system.
9. system returns the group secret key to the requester.

In summary, Steps 1-4 allow the requester to see the list of groups, Steps 4-6 steps authenticate the requester's identity, and Steps 7-8 provide a secret key to the requester to create group signatures.

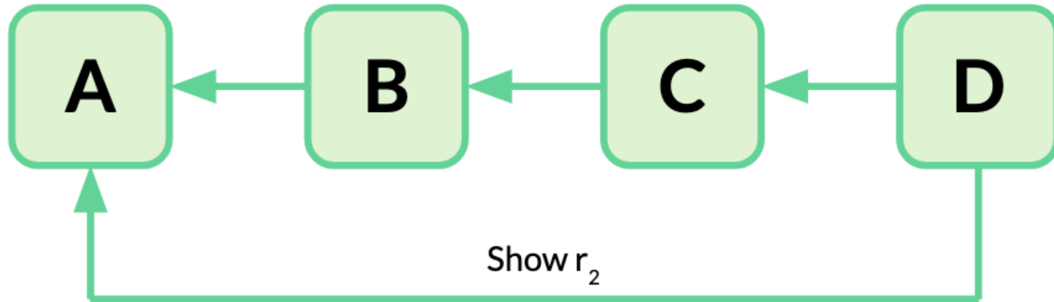


Figure 4.8 Linking Transactions

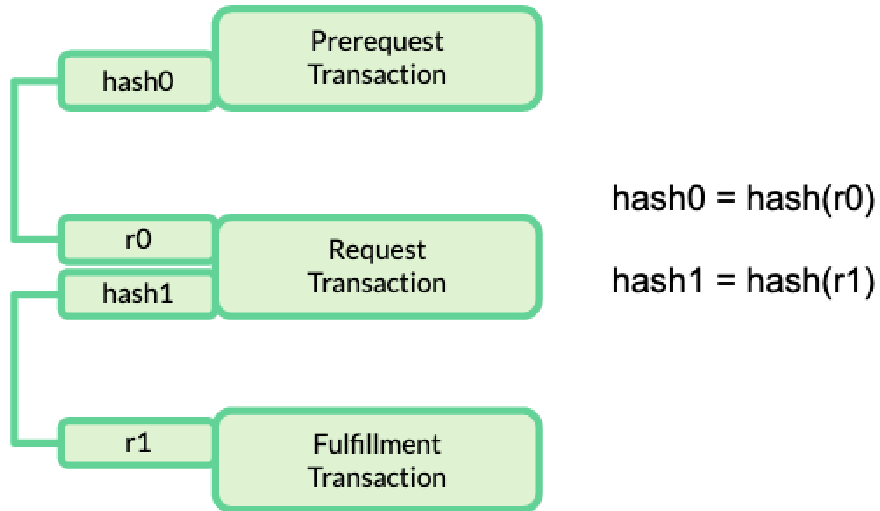


Figure 4.9 Linking Requester Transactions

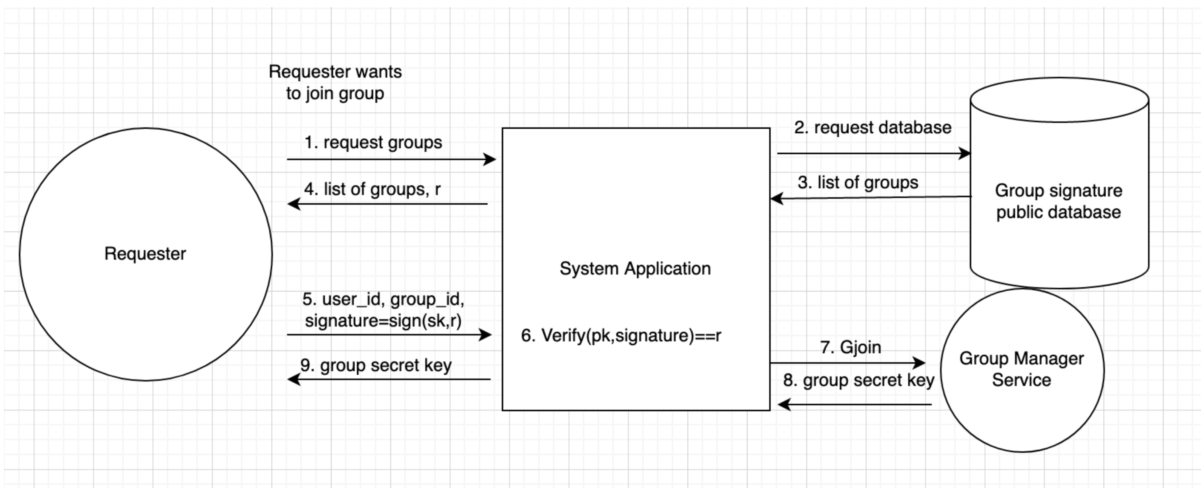


Figure 4.10 Requester Setup

CHAPTER

5 RESULTS

This chapter evaluates whether the linking group signature, following previously defined functions and proposed setup, is appropriate for BigchainDB transactions. We expect group signature and linking signatures to incur an additional time cost for the transactions. The main issue is whether our implementation is efficient enough to justify using group signature in place of the existing signature scheme. Thus we evaluate the appropriateness of our implementation based on its time and scalability performance. Experiment Setup: Running on a Ubuntu version 18.04 virtual machine, we install software components: Docker version 20.10.5, Python version 3.6.9, Cargo version 1.53.0, and MongoDB Compass version 1.26.1. We set Group Manager web service to port 8080, BigchainDB Server access port to default (9984), and MongoDB port to default (27017).

With the experiment setup, we determine the usability of our implementation by answering the following research question:

1. What is the additional cost of adding requester anonymity to BigchainDB transactions?

We aim to analyze additional costs from using group signatures with 3 experiments: time overhead of group signature compared to existing public key signing, scalability of the group signature with increased group members, and time overhead of group signature transactions compared to the old transactions.

5.1 Group Signature vs Basic Signature

To analyze the time increase of group signature compared to the existing basic public group signature (ed25519), we need to compare their respective signing and verifying time costs. Thus we ran 200

times for each of the following functions: the ed25519 sign, ed25519 verify, group signature sign, and group signature verify, all signing and verifying the same message.

Table 5.1 Time Cost of Group Signature

Average Time Cost		
Type of function	Std	Mean
Signing Digital Signature - Python (Ed25519)	0.8402744531	0.748246195ms
Verifying Digital Signature - Python (Ed25519)	0.6407945735	1.006927615ms
Signing Group Signature - RUST	0.9230209381	9.772752225ms
Verifying Group Signature - RUST	0.298527082	11.12937439ms

As we can see from Table 5.1, the average time cost (from 200 runs) of verifying a group signature is very different from the original built-in digital signature (ed25519) within BigchainDB. Additionally, the signing for a group signature costs more than 12 times the signing for an ed25519 signature. However, looking at the whole picture regarding the total time for a transaction, the 8.8 ms increase for signing is less significant. To conclude, while group signing may cost 12 times more than the ed25519 signature, this translates to just 8.8 ms overhead.

5.2 Increasing Group Members

As more members join for the group signature scheme, the groups get larger, leading to an increase in the anonymity set size. Suppose the number of members in a group increases exponentially; then group signature functions should theoretically be scalable. We compared how the signing and verifying time complexity is affected by more members join a group. This experiment compares the average time (from 200 runs) it takes for group signing and group verifying for 2, 4, 8, 16 group members.

Not surprisingly, the group signature wins here when compared to the ring signature. Table 5.2 displays changes of sign and verify of Group Signature as group size increases seem to be just a constant time complexity. At the same time, the additional members in a ring would indicate an exponential time complexity. As the ring adds more members, it adds extra cost to signing and verifying. This table solidifies our group signature over ring signature as it is more scalable when more members join the group than the ring signature when more members join the ring.

Table 5.2 Scalability of Group Signature

Group Growth		
Num of members	Types of function	Group Signature Time Cost
2	Signing Verifying	4.081926ms 4.667232ms
4	Signing Verifying	4.062314ms 4.366295ms
8	Signing Verifying	4.264439ms 4.578303ms
16	Signing Verifying	4.501189ms 4.341704ms

5.3 Group Transaction vs Basic Transaction

Lastly, we are interested in the effects our implementation has on the overall workflow of a marketplace transaction. In this experiment (Table 5.3), we ran 5 transactions Prerequest, Interest, RFQ, Bid, and Fulfillment, where we've added both linkable nonces and group signature to replace the elliptical curve group signature. While the comparison in this experiment is not appropriate as ed25519 signing calls from exclusively python libraries, ed25519 verifying calls from exclusively Java libraries, and the cryptographic code of group signature ultimately relies on the cryptographic C libraries. Instead, this experiment aims to display that despite the increase in complexity because group signature ultimately depends on the C library, it could potentially outperform Java and python libraries in terms of speed.

The coefficient of variation is under 1.0, 0.297 for the Group signature and 0.226 for the original signature, indicating high precision (low variance) within the resulting times. Based on the 95% confidence interval (using a significant level of 0.05), we can safely say that the group signature signing and verifying are statistically significantly slower than the ed25519, and we conclude that 95

Unexpectedly, the overhead of switching languages is exceptionally high. The previous Group Signature vs. Basic Signature experiment shows that the difference between running Edward elliptical curve signature and running group signature is about 19 ms (1090%) increase in the time for a group signature. From Table 5.3, we see our group signature overhead on transactions on average adds 23347 ms (377%) increase.

To summarize, the addition of hash verification, group signing, group verifying, and groups and language switching brings a huge 23347 ms overhead. Breaking down on what we introduced into the transactions, we ran an additional two group signature using the bigchainDB driver and two group verifying by the bigchainDB server within the new group signature transactions. Additionally, we ran two hash creation by the bigchainDB driver and two hatch verification by the bigchainDB server. Lastly, we added four language switches (of which two calls are python to Rust to C++ and two calls are from Java to Rust to C++). Assuming hash creation and hash verification are negligible

Table 5.3 Comparing Transaction Times

Transaction Times		
Trial (ms)	Ed25519	Group Signature
Transaction 1	12096.54662ms	31111.50953ms
Transaction 2	5829.050265ms	21922.53284ms
Transaction 3	6111.45259ms	21595.47869ms
Transaction 4	5989.259657ms	19268.20824ms
Transaction 5	5883.072631ms	34866.42132ms
Transaction 6	5949.524186ms	38999.45385ms
Transaction 7	5711.611076ms	39740.79425ms
Transaction 8	6097.671397ms	32362.10934ms
Transaction 9	5807.926217ms	39448.26101ms
Transaction 10	6017.69875ms	36020.34113ms
Transaction 11	5978.015848ms	37758.13883ms
Transaction 12	5759.222706ms	35866.61027ms
Transaction 13	5630.697453ms	33903.15254ms
Transaction 14	5671.375385ms	15219.04399ms
Transaction 15	5745.127919ms	29837.69522ms
Transaction 16	5699.859339ms	39040.97876ms
Transaction 17	6015.530188ms	21387.40648ms
Transaction 18	5977.222206ms	16074.89105ms
Transaction 19	6062.802051ms	15646.88341ms
Transaction 20	5680.885235ms	30587.86738ms

Table 5.4 Statistical Analysis

Statistical Value		
	Ed25519	Group Signature
Average:	6185.727586ms	29532.88891ms
Std (σ):	1399.817622	8775.26162
Coefficient of variation (CV):	0.2262979743	0.297135226
95% Confidence Interval:	6185.727586± 655.1348134ms	29532.88891± 4106.948858ms

(as the hash function used is built-in), the two the group signing contributing to an expected 18.04 ms increase, and the two group verifying contributing to an expected 20.24 ms increase. The leftover cost of 23308 ms increase can only be attributed to the four languages switches. Thus running in Rust has failed us as our biggest bottleneck is the four language switches.

The trade-off between the additional time cost in the server and driver in exchange for better security, the time loss makes this change extremely undesirable. The main bottleneck would be transferring the signatures and public keys via shellcode, which heavily depends on the system and not the effectiveness of the code. In contrast, in the original ed25519, the (python) BigchainDB driver signs using a native python cryptoconditions library. In the ed25519 verifying, the (Java)

BigchainDB server verifies using a native java ed25519-java library. Both do not need to call another language at all.

CHAPTER

6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

There's a reason why many code implementation of research papers are incomplete or missing. At least with signature schemes research, many schemes are complicated to implement. Other times the scheme is using outdated, time-consuming techniques that are not useable in real-life scenarios. For instance, most of the time, going through group signature schemes, the newer papers refer to older papers that use RSA instead of the bilinear technique that takes exponentially longer. Another big problem that one faces when creating a homebrew signature implementation is that the signature code you write is slow and unoptimized compared to using an existing scheme. Furthermore, cryptography is particular where one missed label in integration can destroy the schema, and even if the code works perfectly, there can still be vulnerabilities.

When a group signature implementation exists, it's usually not up to par with the targeted environment. There has been many ready to use scheme implementations from reputable companies and esteemed research university, but rarely is one that works straight away on your environment (unless you are willing to change your environment). Due to the combination of the ever-evolving nature and the high skill cap, many research papers leave coding aspects alone and pursue the theoretical aspects as less can go wrong.

We have provided progress in using Rust and Python to implement the group signature into a supply chain blockchain. If we could use one language for the whole group signature scheme, we would avoid additional overhead like sending parameters between programming languages. Unfortunately, we had to use bits and pieces of existing code with different programming language libraries as sometimes it was the only library available at the time.

6.2 Future Work

6.2.1 More Powerful Adversary

If the Adversary had more power and could corrupt the Group Manager, we would need $GJudge$. Recall that GJudge runs $DSVerify()$, $GVerify()$, and SPK for Π. These functions help us implemented $GJudge$ in the future.

Furthermore, we looked at the group signature separately from the blockchain. Further analysis is need on whether this group signature could open up vulnerabilities for the blockchain. For instance, the Group Manager's addition allows adversaries to run a Denial of Service on the blockchain network. Thus, SmartChainDB needs to control the group signature scheme and monitor if one should kill the current group signature functions.

6.3 Appendix: Group Signature Pseudocode

Algorithm 1: GSetup(1^η)**:**

```
1  pub fn For_GSetup(count_messages: usize, label: &[u8]) -> (Gpk, Gmsk) {
2
3      let g = SignatureGroup::from_msg_hash(&[label, b" : g"].concat());
4      let g_tilde = OtherGroup::from_msg_hash(&[label, b" : g_tilde"].concat());
5      let x = FieldElement::random();
6      let y = FieldElement::random();
7      let X_tilde = &g_tilde   &x;
8      let Y_tilde = &g_tilde   &y;
9      (
10         Gpk {
11             g,
12             g_tilde,
13             X_tilde,
14             Y_tilde,
15         },
16         Gmsk { x , y },
17     )
18 }
19 pub fn GSetup (count_msgs: usize, label: &[u8])->(Gpk, Gmsk){
20
21     let (gpk, gmsk) = For_GSetup(count_msgs, label);
22     println!("GSetup Successful!");
23     (gpk, gmsk)
24 }
```

Algorithm 2: PKIJoin(i):

```
1  pub fn keygen(count_messages: usize, label: &[u8]) -> (Verkey, Sigkey) {
2
3      let g = SignatureGroup::from_msg_hash(&[label, b" : g"].concat());
4      let g_tilde = OtherGroup::from_msg_hash(&[label, b" : g_tilde"].concat());
5      let x = FieldElement::random();
6      let mut Y = vec![];
7      let mut Y_tilde = vec![];
8      let X = &g   &x;
9      let X_tilde = &g_tilde   &x;
10     for _ in 0..count_messages {
11         let y = FieldElement::random();
12         Y.push(&g   &y);
13         Y_tilde.push(&g_tilde   &y);
14     }
15     (
16         Verkey {
17             g,
18             g_tilde,
19             X_tilde,
20             Y,
21             Y_tilde,
22         },
23         Sigkey { X },
24     )
25 }
26 pub fn PKIJoin (count_msgs: usize, label: &[u8])->(PublicKey,SecretKey){
27
28     let (upk_i, usk_i) = keygen(count_msgs, label);
29     let msg = FieldElementVector::random(count_msgs);
30     let sign_usk_i=Signature::new(msg.as_slice(), &usk_i, &upk_i).unwrap();
31     println!("PKIJoin Successful!");
32     (upk_i, usk_i)
33 }
```

Algorithm 3: GJoin($i, USK[i], UPK[i], GMSK, GPK$):

```
1   pub fn test_sigmaProtocol(g:amcl_wrapper::group_elem_g1::G1,y:FieldElement,Y:
        amcl_wrapper::group_elem_g1::G1)->(){
2
3       let r = FieldElement::random();
4       let A=&g & r;
5       let cha = FieldElement::random();
6       let rsp=&r-&y & cha;
7       let Check=&g & rsp+&Y & cha;
8       println!("Proof of USER knowing ski: {:?}", A==Check);
9   }
10  pub fn GJoin (i: usize, gpk: Gpk,gmsk: Gmsk, upk_i:PublicKey ,usk_i:SecretKey)->((
        usize,amcl_wrapper::group_elem_g1::G1,Signature,amcl_wrapper::group_elem_g2::G2,
        DefaultHasher), (amcl_wrapper::field_elem::FieldElement, (amcl_wrapper::
        group_elem_g1::G1, amcl_wrapper::group_elem_g1::G1), amcl_wrapper::
        extension_field_gt::GT)){
11
12      let ski= FieldElement::random();
13      let tow=&gpk.g   &ski;
14      let tow_tilde= &gpk.Y_tilde   &ski;
15      let mut hash_saved = DefaultHasher::new();
16      let n =sign_usk_i(hash_saved.clone(), tow.clone(), usk_i.clone(), upk_i.clone());
17      let res = GT::ate_pairing(&tow, &gpk.Y_tilde);
18      let res2 = GT::ate_pairing(&gpk.g, &tow_tilde);
19
20      test_sigmaProtocol(gpk.g.clone(),ski.clone(),tow.clone());
21
22      let u= FieldElement::random();
23      let sigma1=&gpk.g   &u;
24      let sigma2=&gpk.g   &gmsk.x   &u + &tow   &gmsk.y   &u;
25      let sigma=(sigma1.clone(),sigma2.clone());
26      let secret_register=(i,tow,n,tow_tilde,hash_saved);
27      let gsk_i=(ski,sigma,GT::ate_pairing(&sigma1,&gpk.Y_tilde));
28
29      println!("GJoin Successful!");
30      (secret_register,gsk_i)
31
32  }
```

Algorithm 4: GSign($GSK[i], m$):

```
pub fn GSign(gsk_i:(amcl_wrapper::field_elem::FieldElement,
    (amcl_wrapper::group_elem_g1::G1,
        amcl_wrapper::group_elem_g1::G1),
    amcl_wrapper::extension_field_gt::GT),msg:&'static  str)->(
    (amcl_wrapper::group_elem_g1::G1,
    amcl_wrapper::group_elem_g1::G1,
    amcl_wrapper::field_elem::FieldElement,
    amcl_wrapper::field_elem::FieldElement),
    DefaultHasher,&'static  str){

    let ski=gsk_i.0;
    let sigma1=gsk_i.1.0;
    let sigma2=gsk_i.1.1;
    let e=gsk_i.2;
    let t = FieldElement::random();
    let sigma1_dash=sigma1  &t;
    let sigma2_dash=sigma2  &t;
    let k = FieldElement::random();
    let e_tok_tot=e.pow(&k).pow(&t);
    let mut hash_saved = DefaultHasher::new();
    let number = H1(hash_saved.clone(),(sigma1_dash.clone(),sigma2_dash.clone(),
        e_tok_tot.clone(),msg)).to_be_bytes();
    let c = FieldElement::from_msg_hash(&number);
    let s = &k + &c  &ski;
    let mu=(sigma1_dash,sigma2_dash,c,s);

    println!("GSign Successful!");
    (mu,hash_saved.clone(), msg)

}
```

Algorithm 5: GVerify(GPK, σ, m):

```
1  pub fn GVerify(gpk: Gpk,mu:(amcl_wrapper::group_elem_g1::G1,
2      amcl_wrapper::group_elem_g1::G1,
3      amcl_wrapper::field_elem::FieldElement,
4      amcl_wrapper::field_elem::FieldElement),
5      hash_for_tuple:DefaultHasher,msg:&'static str)->bool{
6
7      let sigma1_dash=mu.0;
8      let sigma2_dash=mu.1;
9      let c=mu.2;
10     let c1=c.clone();
11     let s=mu.3;
12     let b =&c.negation();
13     let R =GT::ate_multi_pairing(vec![(&(-&sigma1_dash).scalar_mul_variable_time(b),&
           gpk.X_tilde),
14         (&sigma2_dash.scalar_mul_variable_time(b),&gpk.g_tilde),
15         (&sigma1_dash.scalar_mul_variable_time(&s),&gpk.Y_tilde)]);
16     let number = H1(hash_for_tuple.clone(),(sigma1_dash.clone(),sigma2_dash.clone(),R
           .clone(),msg)).to_be_bytes();
17     let c2 = FieldElement::from_msg_hash(&number);
18
19     println!("GVerify Successful!");
20     c1==c2
21
22 }
```

Algorithm 6: GOpen($GMSK, \sigma, m, Reg[i]$):

```
1  pub fn GOpen(gpk: Gpk, gmsk_array: Vec<(usize, amcl_wrapper::group_elem_g1::G1,
       Signature, amcl_wrapper::group_elem_g2::G2, DefaultHasher)>, mu:(amcl_wrapper::
       group_elem_g1::G1,
2      amcl_wrapper::group_elem_g1::G1,
3      amcl_wrapper::field_elem::FieldElement,
4      amcl_wrapper::field_elem::FieldElement),
5      hash_for_tuple:DefaultHasher, msg:&'static  str)->(){
6
7      let sigma1_dash=mu.0;
8      let sigma2_dash=mu.1;
9      let c=mu.2;
10     let s=mu.3;
11     for gmsk in gmsk_array{
12         let idenity_id= gmsk.0;
13         let tow = gmsk.1;
14         let n = gmsk.2;
15         let tow_tilde = gmsk.3;
16         let hash_saved = gmsk.4;
17         if GT::ate_2_pairing(&sigma2_dash,&gpk.g_tilde,&(-&sigma1_dash),&gpk.X_tilde)
               ==GT::ate_pairing(&sigma1_dash,&tow_tilde){
18             println!("The identity is User {:?}", idenity_id);
19             let true_tow_tilde=tow_tilde;
20             let true_identity=(idenity_id,tow,n);
21             let r = FieldElement::random();
22             let cha = &gpk.g & r;
23             let rsp = GT::ate_pairing(&cha,&true_tow_tilde);
24             println!("Proof of knowledge of  _tilde  {:?}", rsp==GT::ate_pairing(&
                   true_identity.1,&gpk.Y_tilde).pow(&r));
25         }
26     }
27 }
```

BIBLIOGRAPHY

[Abr17] Abraham, I. et al. “Efficient Synchronous Byzantine Consensus”. 2017.

[Atz17] Atzei, N. et al. “A Survey of Attacks on Ethereum Smart Contracts (SoK)”. *Principles of Security and Trust*. Ed. by Maffei, M. & Ryan, M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186.

[Beh18] Behrends, R. et al. *BigchainDB 2.0 The Blockchain Database*. Tech. rep. Paper version 1.0. Berlin, Germany: BigchainDB GmbH, 2018, p. 14.

[Ber19] Bernal Bernabe, J. et al. “Privacy-Preserving Solutions for Blockchain: Review and Challenges”. *IEEE Access* **7** (2019), pp. 164908–164940.

[Bic10] Bichsel, P. et al. “Get Shorty via Group Signatures without Encryption”. English. *Security and Cryptography for Networks - SCN 2010*. Vol. 6280. Other page information: 381-398 Conference Proceedings/Title of Journal: Security and Cryptography for Networks - SCN 2010 Other identifier: 2001254. Germany: Springer Berlin Heidelberg, 2010, pp. 381–398.

[Bon04] Boneh, D. & Shacham, H. “Group signatures with verifier-local revocation”. *CCS ’04*. 2004.

[Bon02] Boneh, D. et al. “Aggregate and Verifiably Encrypted Signatures from Bilinear Maps”. *IACR Cryptology ePrint Archive* **2002** (2002), p. 175.

[Buc16] Buchman, E. “Tendermint: Byzantine Fault Tolerance in the Age of Blockchains”. 2016.

[Cac00] Cachin, C. et al. “Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography”. *Journal of Cryptology* **18** (2000), pp. 219–246.

[Cam04] Camenisch, J. & Lysyanskaya, A. “Signature Schemes and Anonymous Credentials from Bilinear Maps”. *CRYPTO*. 2004.

[Chr16] Christidis, K. & Devetsikiotis, M. “Blockchains and Smart Contracts for the Internet of Things”. *IEEE Access* **4** (2016), pp. 2292–2303.

[Dol83] Dolev, D. & Strong, H. R. “Authenticated Algorithms for Byzantine Agreement”. *SIAM J. Comput.* **12** (1983), pp. 656–666.

[Dwo84] Dwork, C. et al. “Consensus in the presence of partial synchrony”. *J. ACM* **35** (1984), pp. 288–323.

[Goo19] Goodell, B. et al. “Concise Linkable Ring Signatures and Forgery Against Adversarial Keys”. `https://eprint.iacr.org/2019/654`. 2019.

[Gra20] Gramoli, V. “From blockchain consensus back to Byzantine consensus”. *Future Generation Computer Systems* **107** (2020), pp. 760–769.

[Hab91] Haber, S. & Stornetta, W. S. “How to time-stamp a digital document”. *Journal of Cryptology* **3** (1991), pp. 99–111.

[Has19] Hassan, F. et al. *Blockchain And The Future of the Internet: A Comprehensive Review*. 2019.

[Ish19] Ishida, A. et al. “Proper Usage of the Group Signature Scheme in ISO/IEC 20008-2”. *Asia CCS ’19*. 2019.

[Kos15] Kosba, A. E. et al. “Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts”. *2016 IEEE Symposium on Security and Privacy (SP)* (2015), pp. 839–858.

[Kwo14] Kwon, J. “Tendermint : Consensus without Mining”. 2014.

[Lav18] Lavanya, B. “Blockchain Technology Beyond Bitcoin : An Overview”. 2018.

[Liu05] Liu1, J. K. & Wong, D. S. “Linkable Ring Signatures: Security Models and New Schemes”. *Computational Science and Its Applications – ICCSA 2005*. Berlin, Heidelberg: Springer, 2005, pp. 614–623.

[Luu16] Luu, L. et al. “Making Smart Contracts Smarter”. *IACR Cryptology ePrint Archive* **2016** (2016), p. 633.

[Mie13] Miers, I. et al. “Zerocoin: Anonymous Distributed E-Cash from Bitcoin”. *2012 IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 397–411.

[Nak08] Nakamoto, S. “Bitcoin : A Peer-to-Peer Electronic Cash System”. 2008.

[Ner21] Nerurkar, P. et al. “Dissecting bitcoin blockchain: Empirical analysis of bitcoin network (2009–2020)”. *Journal of Network and Computer Applications* **177** (2021), p. 102940.

[Niz19] Nizamuddin, N. et al. “Decentralized document version control using ethereum blockchain and IPFS”. *Computers Electrical Engineering* **76** (2019), pp. 183–197.

[Poi15] Pointcheval, D. & Sanders, O. “Short Randomizable Signatures”. `https://eprint.iacr.org/2015/525`. 2015.

[Sam16] Samaniego, M. et al. “Blockchain as a Service for IoT”. *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2016, pp. 433–436.

[Swa15] Swan, M. “Blockchain: Blueprint for a New Economy”. 2015.

[YG18] Yu Gong Fu Jia, S. B. L. K. “Supply chain learning of sustainability in multi-tier supply chains: A resource orchestration perspective”. *International Journal of Operations Production Management* **38** (2018), pp. 1061–1090.

[Zha19] Zhang, L. et al. “An Efficient Linkable Group Signature for Payer Tracing in Anonymous Cryptocurrencies”. *Future Gener. Comput. Syst.* **101** (2019), pp. 29–38.