Decentralized Compromise-Tolerant Public Key Management Ecosystem with Threshold Validation

Jamal H. Mosakheil and Kan Yang

Abstract—This paper examines the vulnerabilities inherent in prevailing Public Key Infrastructure (PKI) systems reliant on centralized Certificate Authorities (CAs), wherein a compromise of the CA introduces risks to the integrity of public key management. We present PKChain, a decentralized and compromise-tolerant public key management system built on blockchain technology, offering transparent, tamper-resistant, and verifiable services for key operations such as registration, update, query, validation, and revocation. Our innovative approach involves a novel threshold block validation scheme that combines a novel threshold cryptographic scheme with blockchain consensus. This scheme allows each validator to validate each public key record partially and proactively secures it before inclusion in a block. Additionally, to further validate and verify each block and to facilitate public verification of the public key records, we introduce an aggregate commitment signature scheme. Our contribution extends to the development of a new, efficient, and practical Byzantine Compromise-Tolerant and Verifiable (pBCTV) consensus model, integrating the proposed validation and signature schemes with practical Byzantine Fault Tolerance (pBFT). Through a comprehensive examination encompassing security analysis, performance evaluation, and a prototype implementation, we substantiate that PKChain is a secure, efficient, and robust solution for public key management.

Index Terms—PKI, Blockchain, PKChain, Block Validation, Compromise-Tolerance.

1 Introduction

Public key cryptography (PKC) enables secure communications over the Internet using public and private key pairs. In PKC, the private key is securely held by an entity, while the corresponding public key is publicly available. Failing to authenticate the public key can lead to man-in-the-middle attacks and leak users' sensitive information. Public key management system addresses public key authenticity problem by providing a verifiable mapping from an entity's name to its corresponding public key.

The most common and predominant Public key management system is the centralized Public Key Infrastructure (PKI) which relies on Certificate Authority (CA) to bind the public key with a specific entity using digital certificates, also known as CA-based PKI. A CA is considered trustworthy and has the central role in registering and revoking a public key. However, recent incidents [1]–[4] have shown that CA may be compromised by adversaries, which will result in the single-point-of-failure in the PKI both in terms of security and availability. For example, a compromise of the root or any subordinate CA can issue a fake or fraudulent certificate by registering public keys for illegitimate principals, update or revoke public keys for existing principals [5].

Towards this problem, several solutions such as Certificate Transparency (CT) [6] and blockchain-based distributed PKIs [7]–[11] are proposed to resolve CA single-point-of-failure and make CA accountable for its action. However, existing solutions follow a reactive approach. In these existing solutions, different actions of a CA are continuously logged in a publicly verifiable data structure so that when a malicious CA issues a fake certificate with a valid signature, it can be detected by observing the log. There is no concrete mechanism to prevent a malicious CA from generating a fake certificate in the first place. The reactive approach for detecting malicious CA and fake certificates is not ideal in practice. The time window between a malicious certificate being issued and detected can be long enough and have disastrous consequences. On numerous occasions, several incidents remained undetected for several weeks, causing a great deal of damage before they were detected [12], [13]. Therefore, a proactive approach is crucial to prevent a CA from issuing a fraudulent certificate in the first place, together with resolving the CA single-point-of-failure.

Recently, several blockchain-based PKIs [7]–[11] have been proposed to distribute PKI services by replacing a conventional CA with one or more miners. The mining process generates certificates that validate public-key records after proof of work, like in public blockchains such as Bitcoin [14] or Ethereum [15]. Miners get rewarded for generating certificates and offering diligence over miner or CA misbehavior. These PKIs leverage the consistency guarantees provided by tamper-proof and consensus mechanism of the blockchain and mitigate the CA single-point-of-failure risk.

However, existing blockchain-based PKIs are distributed in terms of availability but not security necessarily. In existing blockchain-based PKIs, each miner or validator holds an absolute privilege to validate transactions (public key records) and mine a block individually, and assume that everyone would hold/access the same version of the public

• *Jamal H. Mosakheil and Kan Yang are with the Department of Computer Science, University of Memphis, TN, USA 38152. Email: {jmskheil, kan.yang}@memphis.edu.*

key blockchain. However, there is no concrete mechanism to avert *a compromised miner/CA from injecting fake public key records into the blockchain* from the beginning, as the verification is often performed only by verifying the signature. As a result, a compromise miner or CA can generate fake public key records with valid signatures that will be synchronized to all the users and cannot be removed due to the tamper resistance of the blockchain. Although these fake public-key records will eventually be detected by other miners or users due to the append-only ledger log, they still can cause severe damage by allowing enough exposure time windows to attackers.

In this paper, we present a proactive solution and propose a new decentralized, compromise-tolerant, and transparent public key management system based on blockchain technology called PKChain. PKChain eliminates the single-point-of-failure problem that the CA introduces in CA-based PKI, both in terms of security and availability. Similar to the existing distributed PKIs, PKChain also distributes trust among a set of validators instead of relying on a single validator or CA. However, contrary to existing distributed PKIs, PKChain also offers proactive security, preventing malicious users and compromised validators (CA) from injecting a fraudulent public-key record into the blockchain in the first place. The contributions of this paper are summarized as follows:

1) We propose a decentralized and compromise-tolerant public key management framework (PKChain) based on blockchain technologies, providing transparent, tamper-resistant, and verifiable public key services, including public key registration, update, query, validation, and revocation. PKChain removes the need for having separate entity for certificate revocation like in CA-based PKI.
2) We design a threshold block validation (TBV) scheme to enable a majority of block validators (e.g., t out of n) to collaboratively validate whether a public key request indeed comes from a valid user.
3) We design an aggregate commitment signature (ACS) scheme to enable a majority of block validators to collaboratively generate an aggregated signature on each public key commitment, which is public verifiable.
4) We design a new practical Byzantine Compromise-Tolerant and Verifiable (pBCTV) consensus model by integrating both TBV and ACS schemes in practical Byzantine Fault-tolerance (pBFT) consensus model, which reduces communication overhead significantly.
5) We prove that the PKChain is compromise-tolerant by showing that both TBV scheme and ACS scheme are existentially unforgeable against chosen-message attack under the Computational Diffie-Hellman assumption. Moreover, we develop a prototype of PKChain to demonstrate its functionality, efficiency, and robustness.

The rest of paper is organized as follows: Section 2 describes the related work. Section 3 describes PKChain framework, threat model, and design goals. Section 4 describes algorithm design of PKChain. We give a concrete construction of designed security schemes in Section 5. In Section 6, we describe the implementation of core functions of public key management for PKChain. Security model, security proofs and analysis are given for the designed constructions in Section 7 and performance is evaluated in Section 8. Finally, a discussion is provided in 9, and conclude this paper in Section 10.

2 Related Work

2.1 Conventional Approaches to PKI

The most common PKM system is CA-based PKI – specifically, the X.509 standard that binds a public key to an entity or principal, enabling users to validate the public key by verifying the signature in the digital certificate. However, due to CA centralized role in PKI, it introduced a single-point-of-failure in the PKI. Google proposed Certificate Transparency (CT) [6], [13] to make the TLS certificates publicly visible by recording them in public log servers. Technically, CT constitutes public log-servers, monitors, and auditors that serve as witnesses and gossip once any suspicious certificate is seen. However, CT is vulnerable to split-world attacks [16], and gossiping between CT's components introduces a key challenge when attackers control one or more witnesses and deviate them to attacker-controlled log servers to avoid suspicious behavior. Considering this challenge, Syta et al. [17] proposed a collective cosign approach where witnesses further cosign each certificate signed by the CA and recorded in the public-logs servers, which protects a witness from such secret attacks. However, witnesses still don't know for sure if the certificate signed by the CA is legit or rogue; instead, witnesses ensure that any certificate signed by the CA is exposed to public scrutiny. Later, several other studies extended the CT and proposed features to the initial proposal [18]–[22]. However, the proposed transparency solutions provide transparency to CA operations but still do not eliminate the CA single-point-of-failure and depend on interested parties to detect a fraudulent certificate once observed.

The Web of Trust (WoT) PKM is the first step towards decentralization and allows an individual to sign another individual's public key to certify their authenticity [23]. However, the compromise of a widely trusted endorser in WoT can still impact many users whose trust relies on it, and it also lacks a revocation mechanism. Zhou et al. [24], employs threshold cryptography where each CA partially signs client requests. However, it depends on a trusted third party to create shares for all participating servers [24]. Moreover, due to the lack of consensus mechanisms such as state replication, it is difficult to prevent malicious servers from generating a bogus certificate, especially if replica servers and delegate server(s) are compromised simultaneously.

2.2 Blockchain-Based PKI

Recently several blockchain-based PKIs have been proposed to distribute trusts and provide transparency.

Yakubov et al. [25] proposed an Ethereum-based PKI framework aiming to extend the X.509V3 certificate standard with blockchain compatible metadata fields. It utilizes blockchain ledger to trace CA misbehavior concerning certificate revocation. Fromknecht et al. [7] proposed CertCoin, a blockchain-based PKI that ensures identity retention, meaning that preventing multiple users from creating public

keys for the same identity. The authors in [26] uses pBFT consensus, instead of proof of work to validate certificates, and use a digital signature to verify transactions. Blockstack [11] leverages Namecoin and Bitcoin blockchain to provide a name registration service that allows users to bind public keys to their names. Authors in [10] proposed SCPKI, an Ethereum smart contract-based PKI using a web of trust model, aiming to detect rogue certificates when they are published using smart contracts. Cecoin [9] and IKP [8] proposed Ethereum-based PKI uses smart contracts to offer automatic responses to CA's misbehavior and incentivizes those who help detect CA misbehavior. Authors in [27] proposed a blockchain-based PKI system for Named Data Networking (NDN), which addressed the compromised CA problem in the NDN context using the PoW consensus algorithm. However, validators or miners must contact clients each time to verify while validating client requests, introducing a heavy burden on clients.

Similar to the CT solutions, blockchain-based PKIs also follow a reactive approach. Blockchain-based PKIs utilize blockchain as a log-based ledger to trace the CA's misbehavior and react once a fraudulent certificate is observed. However, a concrete structure to detect compromises in the first place is missing. In contrast, PKChain provides a proactive solution and also resolves the CA single-point-of-failure problem.

3 PKChain: A Decentralized Public Key Management Framework

3.1 PKChain System Model

To solve the *Compromised CA Problem* introduced by the CA in the traditional CA-based PKI systems, we propose an alternative decentralized PKI solution called PKChain. PKChain is a decentralized public key management system based on blockchain technology. We compare PKChain with CA-based PKI in Figure 1. As shown in Figure 1, in PKChain, we replace the single full-privileged CA in CA-based PKI with a set of validation nodes (denoted as validators) with lower privileges. PKChain is a permissioned blockchain and a number of designated validators are pre-selected to validate the principals who register their public keys and maintain a blockchain containing the validated public keys. The public key records are stored and maintained in a tamper-resistant blockchain, e.g., the $\mathsf{PKChain}_{\mathsf{edu}}$ manages all the public keys in the edu domain, which were managed by the CA_{edu} in CA-based PKI. PKChain follows a hierarchical structure similar to CA-based PKI. That is, the public keys of all the validators in $\mathsf{PKChain}_{\mathsf{univ}}$ are managed by the $\mathsf{PKChain}_{\mathsf{edu}}$, and similar for other domains. The PKChain framework consists of the following entities:

Validators. Validators are a set of nodes collaboratively performing the public key management functions. Unlike the CA in PKI systems, no validator can complete any public key request of registration, update, and revocation by itself in PKChain and requires the collaboration of all validators. One of the validators is selected as a Block-generator (leader node), and the rest of validators continue to act as Block-validators in each round of the consensus process. The Block-generator selection process is based on

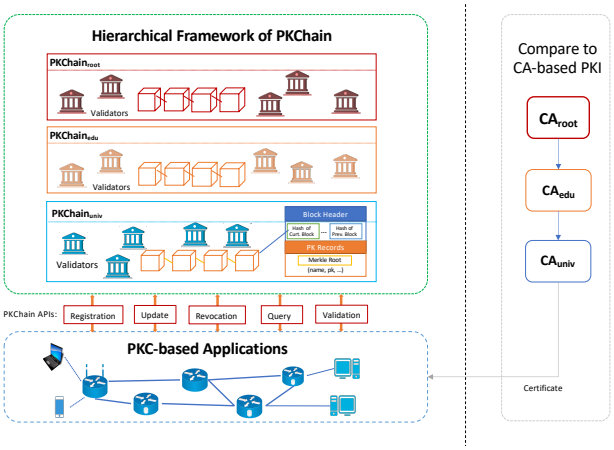

Fig. 1. PKChain Framework

round-robin algorithm or by a relaxed threshold of proof-of-work mechanism. Therefore, each node will have a fair opportunity to become a Block-generator.

- *Block-generator.* Block-generator acts as a leader node and is responsible for proposing a block by collecting the public key request from memory pool and broadcasting the proposed block to all the Block-validators. All the validators will verify all public key requests contained in the block, and only those public key requests approved by the majority of validators will be kept in the block. The Block-generator then adds the new finalized block into the blockchain and responds to the clients.
- *Block-validators.* Block-validators are the pre-selected nodes participating in block validation process upon receiving a block proposal from Block-generator. Block-validators share their commitment with Block-generator if they agree on block proposal in each round.

Client. A client could be a user or an entity that may register public keys in PKChain. Any entity or device can query or validate a public key from PKChain.

Any PKC-based application can interact with PKChain through the APIs mentioned in section 5 to send a request for public key registration, update, revocation, query, or validation.

3.2 Threat Model

The threat model is defined as follows:

- **Compromised Block-generator.** A compromised Block-generator may perform the following attacks: 1) Inject fake public key records into a block; 2) Change the validated block (which includes only the valid public key records) after the consensus mechanism (e.g., add a revoked public key record with valid signature); and 3) Fail to share new block proposal with some or all of the validators causing a temporarily DoS attack.
- **Compromised Block-validator.** We assume that at most one-third of the block-validator can be compromised by adversaries. A compromised Block-validator may perform the following attacks: 1) Make a forged public key request by impersonating the client; 2) Fail to respond to other validators or send a wrong message (e.g., a validation result or a final commit message);

and 3) Collude with other Block-validator or Block-generator to perform a Byzantine fault or public key forgery attack.
- **Compromised Client.** A malicious client or external adversary may make a forged public key request, e.g., register a public key with a different name, update/revoke an existing public key of another user.

3.3 Design Goals

PKChain accomplishes the following design goals to deal with the proposed threat model and provides all PKM services:

- *Eliminating Single-point-of-failure*. PKChain should offer a distributed PKI service and does not introduce a single-point-of-failure both in terms of availability and security.
- *Proactive Security*. PKChain should prevent individual public key validation, and the majority of validators are obliged to validate public key requests, and only the majority approved public key records are included in the blockchain.
- *Public Verifiability*. To be compatible with the existing certificate-based applications, all the public key records in the PKChain should be public verifiable by public users.
- *Compromise Tolerance*. PKChain should be robust against compromised validators as long as the majority of the validators remained honest. A consensus mechanism is required to enable validators to agree on the same state of the blockchain. Moreover, validators should be able to check whether all the records in the final block are valid (in case the Block-generator is compromised).
- *PKM Functionalities*. PKChain should offer PKM core services such as registration, query, update, and revocation of public keys. PKChain should eliminate the need for having a separate entity for certificate revocation, such as certificate revocation list (CRL) or Online Certificate Status Protocol (OCSP) in CA-based PKI.
- *Efficiency*. The PKChain should be efficient in terms of communication overhead and computation cost.

4 Algorithm Design of PKChain

In this section, we first describe the overall ideas, followed by the proposed algorithms, including a novel threshold block validation scheme, an aggregate commitment signature scheme, and a practical consensus model.

4.1 Technical Overview

Eliminating Single-point-of-failure. PKChain eliminates the single-point-of-failure problem that the CA introduces in CA-based PKI, both in terms of security and availability. Similar to the existing distributed PKIs, PKChain also distributes trust among a set of validators instead of relying on a single validator or CA. However, contrary to existing distributed PKIs, in PKChain, each validator has a lower privilege and requires other validators' collaboration to validate the final public key record.

Proactive Security. Contrary to existing PKI solutions, PKChain provides proactive security. We defined informally proactive security here as predominately preventing compromise validator(s) from infiltrating fake public key records (certificates) into the blockchain. Following challenges should be addressed to accomplish proactive security in PKChain.

1) *Preventing individual validation*. Typically, to perform individual public key validation, a user authentication method is required that enables each CA to verify that the public key request is from a valid user holding valid credentials (e.g., a private key or password). However, signature-based validation is not always applicable due to the lack of private-public key pair (e.g., new public key registration request) or compromised private key (e.g., public key update/revocation requests). Other authentication methods, such as password-based authentication, may be applicable to validate the public key request. However, to complete the validation individually, each CA may host the complete user credentials (e.g., security token), enabling a compromised CA to easily impersonate a user with such complete user credentials and produce a forged public key request.
2) *Finding honest collector*. It is challenging to select an honest validator to coordinate and collect all the evaluation results, as Block-generator may also be compromised. Indeed, finding an honest collector to collect all valid partial signatures is a challenge in existing threshold cryptography. That is why directly cosigning a certificate by a threshold number of validators does not solve the problem (discussed in detail in section 9).

We developed a Threshold Block Validation (TBV) scheme to resolve the first challenge. The TBV scheme allows each validator only to host partial credentials (security tokens). In TBV, each validator partially evaluates client requests, shares its partial result with other validators, and accepts the final evaluation results only when receiving t valid partial evaluation responses from other validators. Once each validator received t valid evaluation results, each validator signed a COMMIT message supporting the evaluation result.

The solve the second challenge, we developed an Aggregate Commitment Signature (ACS) scheme. The ACS scheme allows each validator to aggregate locally t valid partial signatures on the COMMIT message. Even though only the block-generator inserts the final aggregated signature on public-key record (certificate) into the block. However, suppose the block-generator behaves maliciously by injecting a fake aggregated signature. In that case, it will be caught by other validators, as each validator has a local copy of the aggregated signature.

Public Verifiablity. The ACS scheme also makes each public key record in PKChian public verifiable and makes PKChain compatible with existing CA-based applications that require public verifiability. Technically, the ACS scheme helps aggregate the majority of validators' TBV scheme results and provides an aggregate signature for each public key record that public users can verify.

Our ACS scheme is based on the Boneh et al. [28] signature. However, the Boneh et al. [28] signature can't be applied directly to achieve our design goals. We assume

in PKChain the validators are trustless. So, the main challenge is that no trusted entity can generate and distribute the signing keys in PKChain. Considering this challenge, we propose a distributed and verifiable key generation algorithm to generate the signing and verification keys for all the validators while integrating the PKChain identity ($h = H_2(PKCHainID)$) during the distributed key generation process. Also, in Boneh et al. [28] signature, each message signed must be distinct to be secure against existential forgery in the aggregate chosen-key model. While in PKChain, all validators sign the same final COMMIT message. In addition, our ACS scheme is more efficient in verification as it only requires three pairing operations while [28] requires $\mathcal{O}(n)$, where n is the number of signers. Fast verification of public key records (certificate) is mandated to serve existing PKI applications efficiently.

Compromise Tolerance & Efficiency. We proposed the practical Byzantine Compromise Tolerant and Verifiable (pBCTV) consensus mechanism based on the well-studied practical Byzantine Fault Tolerance (pBFT) algorithm [29] to achieve consensus among the majority of validators. We designed pBCTV to achieve the following design goals, which cannot be achieved by directly using pBFT or its variants.

1) To prevent the extra communication overhead caused by the proposed schemes (TBV & ACS). To attain this objective, we integrated the TBV scheme into the Prepare phase and the ACS scheme into the Commit phase of pBFT. Hence, the TBV and ACS schemes make full use of the message exchange in the pBFT phases, preventing them from causing extra communication overhead.
2) Unlike pBFT, where each replica shares its final result directly with the user, in pBCTV Block-generator generate the aggregate signature and submit the final block to the blockchain. As a result, it reduces client communication from $f + 1$ (f is the number of faulty nodes) to 1. As client only needs to receive one message and verify only the final aggregated signature. In contrast to pBFT, the client needs to receive and verify $f + 1$ messages.
3) Recovering previous TBV scheme results in the presence of malicious Block-generator. The Type-2 view-change explained and handled this case in section 4.4.

4.2 Threshold Block Validation Scheme

To validate whether a public key request (e.g., register, update, revoke) is from a legitimate client and not forged by a compromised validator or any other adversary, we propose a *threshold block validation scheme* to enable the validators to validate each public key request in the block collaboratively.

Definition 4.1 (Threshold Block Validation Scheme)**.** The threshold block validation (TBV) scheme consists of the following algorithms: ClientReg, ReqGen, ReqEval, and ReqVerify.

- $\mathsf{ClientReg}(S_v, ID) \rightarrow (pwd, k_c, \{h_{c,i}\}_{i \in S_v}, sp)$: Client registration algorithm is run by each client. It takes as inputs the validator set S_v and the client identity ID. It outputs a client password pwd, a client key k_c, a set of secrets $\{h_{c,i}\}_{i \in S_v}$ and a system parameter sp, where pwd and k_c are kept private by the client and each secret $h_{c,i}$ is shared to validator i.
- $\mathsf{ReqGen}(sp, k_c, pwd, ID) \rightarrow (req, P_{req})$: The request generation algorithm is run by each client. It takes as inputs the system parameter sp, client key k_c, client password pwd and client identity ID. It generates a public key request which consists of client ID and public key (ID, PK) and the type of operation requested (e.g., create, update, or revoke). The algorithm also outputs a request proof P_{req} corresponding to this request req.
- $\mathsf{ReqEval}(sp, h_{c,i}, req, P_{req}) \rightarrow Sig_i$: The request evaluation algorithm is run by each validator. It takes as inputs the system parameter sp, the secret $h_{c,i}$ associated to client identity ID, the request req, and the request proof P_{req}. It will perform the evaluation on the request and outputs an evaluation result Sig_i on this request.
- $\mathsf{ReqVerify}(sp, \{Sig_i\}_{i \in S_1}, req, P_{req}) \rightarrow$ YES/NO: The request verification algorithm is run by each validator. It takes as inputs the system parameter sp, a set of request evaluation results $\{Sig_i\}_{i \in S_1}$, the request req and the request proof P_{req}. It outputs 'YES' if the request and request proof are valid. Otherwise, it outputs 'NO'.

4.3 Aggregate Commitment Signature Scheme

To verify if a public key record is approved by a majority of validators and enable public verification, we propose an aggregate commitment signature scheme, based on [28] scheme.

Definition 4.2 (Aggregate Commitment Signature Scheme)**.** The aggregate commitment signature (ACS) scheme consists of Setup, KeyGen, Sign, SigAgg, and PubVerify algorithms.

- $\mathsf{Setup}(1^\lambda) \rightarrow pp$: The setup algorithm takes as inputs the security parameter λ and the validator set S_v. It outputs the public parameters pp.
- $\mathsf{KeyGen}(pp, S_v) \rightarrow (\{k_{s,j}, g_{s,j}\}_{j \in S_v}, k_v)$: The validator key generation algorithm is a distributed algorithm that runs by all the validators. It takes as inputs the public parameter pp and the validator set S_v. It outputs a validator signing key $k_{s,j}$ and validator verification key $g_{s,j}$ for each validator j and a public verification key k_v.
- $\mathsf{Sign}(k_{s,j}, pp, m) \rightarrow \sigma_j$. The signing algorithm is run by each validator. It takes as inputs the secret key $k_{s,j}$, the public parameters pp, and the commitment message m. It outputs a signature σ_j on the commitment message.
- $\mathsf{SigVerify}(pp, g_{s,j}, m, \sigma_j) \rightarrow 1/0$. The individual signature verification algorithm is run by each validator. It takes as input the public parameters pp, individual validator verification key $g_{s,j}$, the commitment message m, and each validator signature σ_j. It outputs 1 if the individual signature σ_j is valid (i.e., σ_j can be verified by $g_{s,j}$). Otherwise, it outputs 0.
- $\mathsf{SigAgg}(pp, \{\sigma_j\}_{j \in S_2}) \rightarrow \sigma$. The signature aggregation algorithm is run by both block generator and block validators. It takes as inputs the public parameters pp and a set of valid signatures $\{\sigma_j\}_{j \in S_2}$, where S_2 is a set of validators whose signatures are valid. It outputs an aggregated signature σ.

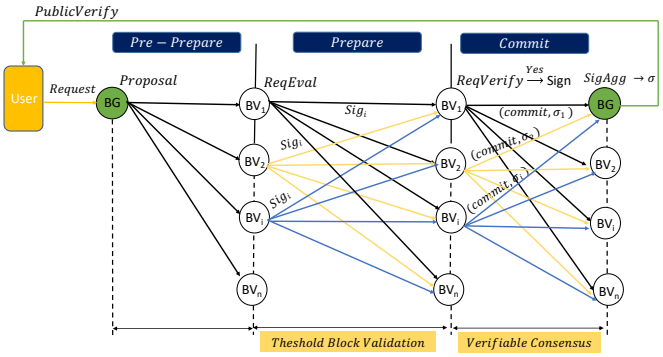

Fig. 2. pBCTV Consensus Model

- PubVerify$(pp, k_v, m, \sigma) \rightarrow 1/0$. The public verification algorithm is run by any public users. It takes as inputs public parameters pp, public verification key k_v, commitment message m, and the aggregated signature σ. It outputs 1 if the signature is valid, Otherwise outputs 0.

4.4 pBCTV Consensus Model

To cope with the attacks from compromised validators, and to achieve the consensus on the state of PKChain among all the validators, we propose a new practical Byzantine Compromise-Tolerant and Verifiable (pBCTV) consensus model. The pBCTV consensus model is based on the pBFT protocol which consists of the *PrePrepare*, *Prepare* and *Commit* phases. To achieve the compromise-tolerant, we embed the threshold block validation scheme into the pBFT protocol and take advantage of the message broadcasting in the *PrePrepare* and *Prepare* phases. We further embed the aggregate commitment signature scheme in the consensus model to enable the public verification of public key records, which also makes full use of the message broadcast in the *Commit* phase. Figure 2 describes the pBCTV consensus model.

Definition 4.3 (pBCTV Consensus Model)**.** The pBCTV consensus model consists of the following phases: System Initialization, Client Enrollment, Threshold Block Validation, Verifiable Consensus Agreement, and Public Verification.

Phase 1: System Initialization. During system initialization, ACS.Setup algorithm is run to generate system parameters pp. Then, ACS.KeyGen is run by all the block validators to obtain the secret keys $\{k_{s,j}\}_{j \in S_v}$, individual validator verification key $g_{s,j}$, and a public verification key k_v.

Phase 2: Client Enrollment. It involves two stages:

a) Name-Principal Registration: Any client is required to register to PKChain (i.e., to all the validators) by running the TBV.ClientReg algorithm and sending the name-principal registration $(ID, h_{c,i})$ to each validator i.

b) Name-Principal Authentication: The Name-Principal authentication happens only in the initial registration phase. Upon receiving the client registration $(ID, h_{c,i})$, each validator i will conduct the name-principal verification by contacting the client to identify and authenticate whether ID is indeed associated with the client (principal) corresponding to its proof of ownership documents. Technically, the identity verification that occurs here is similar to the identification and authentication process of the registration authority (RA) in CA-based PKI, where an RA verifies the requestor's identity before a CA issues a certificate. Once the name-principal identity is verified successfully, each validator i stores the corresponding client $(ID, h_{c,i})$, which can be used for future client public key requests validation, including query, update and revoke.

Phase 3: Threshold Block Validation. The TBV.ReqGen algorithm is run by the client to generate a request and request proof, which will be broadcast to the rest of Block-validators. The TBV.ReqEval algorithm is run by both Block-generator and Block-validators in the *Prepare* phase of the consensus protocol. Each validator runs this algorithm to evaluate client request, and shares its result with other validators. The request verification is performed by running the TBV.ReqVerify at the beginning of the *Commit* phase of consensus protocol.

Phase 4: Verifiable Consensus. Consists of two cases:

a) Normal Case: If the public key request is valid (i.e., TBV.Verify outputs 'YES'), a commitment message will be generated by the validator to show the support of this public key request. Then, each validator j will sign the commitment message by running ACS.Sign algorithm. The output signature σ_j will be broadcast to all the other validators (as a vote on the request). Finally, the Block-generator will verify individual signature σ_j by running ACS.SigVerify algorithm, and select a set with t valid signatures. Then, the Block-generator will aggregate all the valid signatures in this set by running the ACS.SigAgg algorithm and add the aggregated signature σ to the associated public key record in the block. If the public key request is invalid (i.e., TBV.Verify outputs 'NO'), no commitment message will be generated, and a broadcast or reject message will be broadcast to all other validators.

b) Fault/Compromised Case : A malicious Block-generator may aggregate wrong signatures trying to forge the public-key record. The Block-validators can detect malicious Block-generator behavior by running the ASC.PubVerify algorithm. Once a malicious Block-generator is detected, a view-change will occur, and a new Block-generator will be elected. The new Block-generator does not need to re-start entire consensus process by publishing a new block proposal. Instead, it can use the previous round validation results (TBV results), re-calculate the signature aggregation, and share it with rest of the Block-validators. If the majority of validators agree, then the new Block-generator will add the block to blockchain.

Phase 5: Public Verification. Any client and validators are able to verify the aggregated signature of the public key record by running the ACS.PubVerify algorithm.

5 A Concrete Construction

This section provides concrete constructions of TBV and ACS schemes and describe them with the design of pBCTV.

5.1 Phase 1: System Initialization

The ACS.Setup algorithm is run to set up the PKChain identity PKChainID and public parameter pp. The ACS.KeyGen

algorithm is run by all block validators to generate the validator signing keys $\{k_{s,j}\}_{j\in S_v}$, validator verification key $g_{s,j}$ for each validator j, and a public verification key k_v.

- $\mathsf{ACS.Setup}(1^\lambda, S_v) \rightarrow (\{k_{s,j}\}_{j\in S_v}, pp)$: Let $PG = (\mathbb{G}, \mathbb{G}_T, p, g, e)$ be a symmetric-pairing group, where $\mathbb{G}$ and $\mathbb{G}_T$ are multiplicative groups with same prime order p, g is a generator of $\mathbb{G}$, and $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be the bilinear map. Let $H_2 : \{0,1\}^* \rightarrow G$ be a hash function that maps PKChain identity to an element in $\mathbb{G}$. Let $H_3 : \{0,1\}^* \rightarrow G$ be a hash function that maps a commitment message to an element in $\mathbb{G}$. The algorithm chooses a PKChain identity $\mathsf{PKChainID}$ and computes $h = H_2(\mathsf{PKChainID})$ and outputs public parameter as

$$pp = (G, G_T, p, g, e, H_2, H_3, \mathsf{PKChainID}, h, t).$$

- $\mathsf{ACS.KeyGen}(pp, S_v) \rightarrow (\{k_{s,j}, g_{s,j}\}_{j\in S_v}, k_v)$: Because there is no centralized entity that can generate the signing keys for all the validators, we propose a distributed algorithm to share the secrets among all the validators. Each validator i first checks if $h = H_2(\mathsf{PKChainID})$. If not, it aborts. Then, each validator i chooses a random secret $s_i \in \mathbb{Z}_p^*$ and a polynomial function $f_i(x)$ with degree $t-1$ as

$$f_i(x) = a_{i,t-1}x^{t-1} + \cdots + a_{i,1}x + s_i$$

where $a_{i,t-1}, \cdots, a_{i,1}$ are randomly selected from $\mathbb{Z}_p^*$. The validator i will compute n_v tuples $(j, h^{f_i(j)}, g^{f_i(j)})_{1\le j\le n_v}$, where n_v is the total number of validators (i.e., $n_v = |S_v|$). Then, the validator i sends $(j, h^{f_i(j)}, g^{f_i(j)}, g^{s_i})$ to each validator j, while keeping $(i, h^{f_i(i)}, g^{f_i(i)})$ itself. Upon receiving $n_v - 1$ different tuples from all the other validators, each validator j will compute the individual signing key as

$$k_{s,j} = \prod_{i=1}^{n_v} h^{f_i(j)} = h^{s'_j}$$

and individual verification key as

$$g_{s,j} = \prod_{i=1}^{n_v} g^{f_i(j)} = g^{s'_j}$$

where $s'_j = \sum_{i=1}^{n_v} f_i(j)$ is the new implicit secret share. So, if the distributed secret sharing is correct, at least t out of n_v validators can reconstruct the total secret as

$$s = \sum_{j=1}^{t} c_j s'_j = \sum_{j=1}^{t} c_j \sum_{i=1}^{n_v} f_i(j) = \sum_{i=1}^{n_v}\sum_{j=1}^{t} c_j f_i(j) = \sum_{i=1}^{n_v} s_i$$

The correctness of distributed secret sharing scheme can be verified by checking whether the public verification key generated from new (t, n_v)-secret shares and previous (n_v, n_v)-secret shares are consistent, i.e.,

$$k_v = \prod_{j=1}^{t} (g_{s,j})^{c_j} \stackrel{?}{=} k'_v = \prod_{i=1}^{n_v} g^{s_i}.$$

If the above equation does not hold, it terminates and reruns the key generation algorithm until the above equation holds.

5.2 Phase 2: Client Enrollment

The client enrollment includes the following two steps:

a) Name-Principal Registration. Any client is required to register to the PKChain (i.e., all the validators) by running the $\mathsf{TBV.ClientReg}$ algorithm and sending the name-principal registration $(ID, h_{c,i})$ to the corresponding validator i.

- $\mathsf{TBV.ClientReg}(S_v, ID) \rightarrow (pwd, k_c, \{h_{c,i}\}_{i\in S_v}, sp)$: To be consistent with the aggregate commitment signature scheme, we will use the same set of group parameters defined in the public parameter pp. Let $H : \{0,1\}^* \rightarrow G$ be a hash function that maps arbitrary public key requests to an element in $\mathbb{G}$. Let $H_1 : \{0,1\}^* \rightarrow G$ be a hash function that maps a password pwd to an element in $\mathbb{G}$. The system parameter sp consists of $(G, G_T, p, g, e, H, H_1, \mathsf{PKChainID})$. The client first chooses an ID and password pwd for the client enrollment. Then, it selects a random client key k_c from $\mathbb{Z}_p^*$ and constructs a polynomial with degree $t-1$ as follows:

$$g(x) = a_{t-1}x^{t-1} + ... + a_1 x + a_0,$$

where a_0 is set to the client key k_c. For each validator i, the algorithm generates a share of the client key $k_{c,i} = g(i)$ and computes $h_{c,i} = H_1(pwd)^{k_{c,i}}$. Then, the name-principal registration pair $(ID, h_{c,i})$ is sent to the validator i.

b) Name-Principal Authentication. Upon receiving the name-principal registration pair $(ID, h_{c,i})$, each validator i contacts the principal to verify the ID via different channels. For example, if an entity is a user, legal ID is required, and proof of ownership is required if an entity is a device or organization. Once the name-principal authentication is successful, each i will store $(ID, h_{c,i})$ in its database. Thus, it enables validators to avoid contacting the principal and doing the name-principal validation for each public key request.

5.3 Phase 3: Threshold Block Validation

The $\mathsf{TBV.ReqGen}$ algorithm is run by the client to generate a request and request proof, which will be broadcast to the rest of the Block-validators. The $\mathsf{TBV.ReqEval}$ algorithm is run by both Block-generator and Block-validators in the *Prepare* phase of the consensus protocol. Each validator runs this algorithm to evaluate the client request and shares its result with other validators. The request verification is done by running the $\mathsf{TBV.ReqVerify}$ algorithm at the beginning of the *Commit* phase of consensus protocol.

- $\mathsf{TBV.ReqGen}(sp, k_c, pwd, ID) \rightarrow (req, P_{req})$: The request generation algorithm generates a public key request which consists of the pair of the client, public key (ID, PK), the type of operation requested (e.g., create, update, or revoke), and the timestamp, denoted as

$$req = (\mathsf{PKChainID}||\mathsf{ID}||\mathsf{PK}||\mathsf{OP\_Type}||\mathsf{Timestamp})$$

Where operation types (OP_Type):

$$OP\_Type = Create/Update/Revoke$$

The algorithm also generates a request proof P_{req} corresponding to this request req as

$$P_{req} = \Big(H_1(pwd)^{k_c} H(req)^u, H(req)^r, g^u, g^r\Big)$$

where $u, r \leftarrow \mathbb{Z}_p^*$.

- $\mathsf{TBV.ReqEval}(sp, h_{c,i}, req, P_{req}) \rightarrow Sig_i$: The request evaluation algorithm is run by each Block-validator and the Block-generator. The signature Sig_i is computed as:

$$Sig_i = \Big(e\big(h_{c,i} \cdot H(req)^{t_i}, g^r\big), g^{t_i}\Big), \quad t_i \leftarrow \mathbb{Z}_p^*.$$

- $\mathsf{TBV.ReqVerify}(sp, \{Sig_i\}_{i\in S_1}, req, P_{req}) \rightarrow$ YES/NO: The request verification algorithm is run by each Block-validator at the beginning of *Commit* phase. Basically, after receiving t different signatures from a set of validators (i.e.,S_1), each validator i performs the signatures aggregation and verifies the proof of request as:

$$\prod_{i\in S_1} \left(\frac{e\big(h_{c,i} H(req)^{t_i}, g^r\big)}{e\big(H(req)^r, g^{t_i}\big)}\right)^{z_i} \stackrel{?}{=} \frac{e\big(H_1(pwd)^{k_c} H(req)^u, g^r\big)}{e(H(req)^r, g^u)}$$

where z_i is the Lagrange coefficient. It outputs 'YES' if the request and its proof are valid, otherwise, outputs 'NO'.

Correctness:

$$\begin{aligned}
&\prod_{i\in S_1} \left(\frac{e\big(h_{c,i} \cdot H(req)^{t_i}, g^r\big)}{e\big(H(req)^r, g^{t_i}\big)}\right)^{z_i} \\
&= \prod_{i\in S_1} e(h_{c,i}, g^r)^{z_i} \\
&= e(H_1(pwd), g^r)^{\sum_{i\in S_1} z_i k_{c,i}} \\
&= e(H_1(pwd), g^r)^{k_c} \\
&= \frac{e\big(H_1(pwd)^{k_c} \cdot H(req)^u, g^r\big)}{e(H(req)^r, g^u)}
\end{aligned}$$

5.4 Phase 4: Verifiable Consensus

We will describe this phase in two cases: a) Normal Case and b) Fault/Compromise Case.

1) Normal Case: If the public key request is valid (i.e., TBV.Verify outputs 'YES'), a commitment message will be generated by the validator to show the support of this public key request. Then, each validator j will sign the commitment message by running ACS.Sign:

- $\mathsf{ACS.Sign}(k_{s,j}, pp, m) \rightarrow \sigma_j$. The signing algorithm outputs a signature σ_j on commitment message m as

$$\sigma_j = (\sigma_{j,1} = k_{s,j} \cdot H_3(m)^{r_j}, \sigma_{j,2} = g^{r_j}).$$

If the public key request is invalid (i.e., TBV.Verify outputs 'NO'), no commitment message will be generated, and a broadcast or a reject message will be broadcast to all the other validators.

Each validator j sends its signature, σ_j (as a vote), to other validators. Block-generator verifies each validator signature σ_j by running the following $ACS.SigVerify$ algorithm, and add only the valid σ_j to the list.

- $\mathsf{ACS.SigVerify}(pp, g_{s,j}, m, \sigma_j) \rightarrow 1/0$. The individual signature verification algorithm uses the validator verification key $g_{s,j}$ to evaluate signature σ_j (vote) for the commitment message m using the following equation:

$$e(\sigma_{j,1}, g) \stackrel{?}{=} e(H_2(\mathsf{PKChainID}), g_{s,j}) \cdot e(H_3(m), \sigma_{j,2})$$

If the above equation holds, it outputs 1, otherwise 0.

Correctness:

$$\begin{aligned}
e(\sigma_{j,1}, g) &= e(k_{s,j} \cdot H_3(m)^{r_j}, g) \\
&= e(h^{s'_j}, g) \cdot e(H_3(m)^{r_j}, g) \\
&= e(h, g^{s'_j}) \cdot e(H_3(m), g^{r_j}) \\
&= e(H_2(\mathsf{PKChainID}), g_{s,j}) \cdot e(H_3(m), \sigma_{j,2})
\end{aligned}$$

Based on the outputs of the individual signature verification, the Block-generator will generate a set S_2 consisting of t valid signatures, and aggregate all the valid signatures in S_2 by running the ACS.SigAgg algorithm, and add the aggregated signature σ to the public key record in the block. Actually, each validator can aggregate the signature upon receiving at least t valid signatures $(i.e., S_2)$ by running the following algorithm:

- $\mathsf{ACS.SigAgg}(pp, \{\sigma_j\}_{j\in S_2}) \rightarrow \sigma$. The signature aggregation algorithm outputs an aggregated signature σ as

$$\sigma = (\sigma_1 = \prod_{j\in S_2} \sigma_{j,1}^{c_j}, \sigma_2 = \prod_{j\in S_2} \sigma_{j,2}^{c_j})$$

2) Fault/Compromise Case: View Change. The view change protocol provides liveness by allowing the PKChain to ensure progress when the Blockchain-generator fails or acts maliciously. The pBCTV view-change mechanism prevents a temporary DoS attack and a Block-generator malicious behavior. There are two types of view-change mechanism that occurs as follow in pBCTV:

Type 1 View-change. Type 1 view-change occurs similar to the pBFT's view-change when one of these events happened: 1) A time-out occurs due to network delay. 2) When a Block-generator does not share the proposal with the rest of the Block-validators or proposed malicious data, similar to pBFT, one of the Block-validator initiates a view-change and send a view-change message to the rest of Block-validators, and if a majority agree, then a view-change occurs. A new Block-generator is elected to send the proposal again. 3) To start a new round after completing a previous normal round.

Type 2 View-change. Type 2 view-change will occur when a malicious Block-generator tried to change the validated block (e.g., inject a previous public key record with a valid signature). To detect such malicious behaviors of Block-generator, the validated block will be broadcast to all the Block-validators again for the final confirmation. Block-validators will compare the received validated block with the local one and broadcast the *approve* or *reject* confirmation messages to all the Block-validators. The validated block will be accepted and added to the chain if more than t *approve* confirmation messages are received. Otherwise, the Block-generator does not behave honestly, and a Type 2 view-change will occur. A new Block-generator will be elected and uses the previously available validation results (TBV result) and send its local validated block to other validators for further confirmation.

5.5 Phase 5: Public Verification

Clients and validators can verify public key record signature with public parameters pp and public verification key k_v:

- ***ACS.PubVerify***$(pp, k_v, m, \sigma) \rightarrow 1/0$. The public verification algorithm will use the public verification key k_v to evaluate the commitment message m and the aggregated signature σ using the following equation:

$$e(\sigma_1, g) \stackrel{?}{=} e(H_2(\mathsf{PKChainID}), k_v) \cdot e(H_3(m), \sigma_2)$$

If the aggregated signature is valid, it outputs 1, otherwise outputs 0.
Correctness:

$$\begin{aligned} e(\sigma_1, g) &= e(\prod_{j \in S_2} (k_{s,j} \cdot H_3(m)^{r_j})^{c_j}, g) \\ &= e(h^{\sum_{j \in S_2} c_j s'_j}, g) \cdot e(\prod_{j \in S_2} H_3(m)^{r_j})^{c_j}, g) \\ &= e(h, g^s) \cdot e(H_3(m), \prod_{j \in S_2} g^{r_j c_j}) \\ &= e(H_2(\mathsf{PKChainID}), k_v) \cdot e(H_3(m), \sigma_2) \end{aligned}$$

6 PKM Core Functions

Each core function of a public key management system is implemented as an API in PKChain.

$\mathbf{PKChain.Register}(name, pk) \rightarrow Success/Fail$. Principal can register a name and bind a corresponding public key to this name. The client will choose a name for a principal (owned or managed by this client) and generate a public-private key pair. Then, a public key request is generated as $(\mathsf{ID}||(\mathsf{name}, \mathsf{pk})||\mathsf{Create})$ and sent to the PKChain memory pool. When the record is added to the blockchain, the client receives the result 'Success'; otherwise, it receives 'Fail'. The client can check and verify the signature of the public key.

$\mathbf{PKChain.Query}(name) \rightarrow pk_{name}/NotFound$. The query function returns the latest and valid public key corresponding to a given name. Any validator can retrieve the entire chain and return the corresponding valid pk_{name} or 'NotFound' to indicate no valid public key is found for the given $name$.

$\mathbf{PKChain.Validate}(name, pk) \rightarrow Valid/Invalid$. Given a pair of name and public key $(name, pk)$, the validation function returns whether this pair is valid or invalid. It calls the Query function with the $name$, then compares the output with the given public key pk. If they are the same, it returns 'Valid'. Otherwise, it returns 'Invalid'. This validation function will remove the need for having a separate entity for certification revocation (e.g., Certification Revocation List or Online Certificate Status Protocol) like in CA-based PKI.

$\mathbf{PKChain.Update}(name, pk^*) \rightarrow Success/Fail$. The update function enables any principal to request to update the corresponding public key for a registered name. When updating a public key for a given name, the $(name, pk^*)$ request goes to PKChain. The *TBV* scheme is used to figure out if this is a request from a legitimate user and there is already a registered user to the PKChain. If so, the public key will be updated successfully, and a 'Success' will be returned to the client. Otherwise, it returns 'Fail'.

$\mathbf{PKChain.Revoke}(name, pk) \rightarrow Success/Fail$. If a public key is compromised, it needs to be revoked. The *revoke* operation is similar to the *update* operation. The *TBV* scheme is used to figure out if this request comes from a legitimate user who has already registered to PKChain. If so, the public key will be revoked successfully, and a 'Success' will be returned to the client. Otherwise, it returns 'Fail'.

Because blockchain is an immutable ledger, we cannot modify content in previous blocks. Instead, a new block is attached to the blockchain indicating an update or revoked public key.

To secure the communication among validators and validators know and have access to each other public keys the public keys of all the validators maintaining $\mathsf{PKChain}_{\mathsf{univ}}$ are stored in the blockchain $\mathsf{PKChain}_{\mathsf{edu}}$. Likewise, the public keys of validators maintaining $\mathsf{PKChain}_{\mathsf{edu}}$ are stored in the blockchain $\mathsf{PKChain}_{\mathsf{root}}$. The public keys of the root validators are well-known and stored in the blockchain, which is shared with all validators in the initial setup. Due to the majority principle and tamper-proof of blockchain, a misbehaving root validator has limited capability to cause damage.

7 Security Analysis of PKChain

This section first describes the complexity assumption, security model, and security proofs for TBV and ACS schemes, followed by a security analysis of pBCTV model.

7.1 Computational Diffie-Hellman Assumption

Definition 7.1 (CDH)**.** Let $\mathbb{G}$ be a multiplicative group with prime order p. The CDH problem on $\mathbb{G}$ states that given (g, g^a, g^b) with a randomly chosen generator g and $a, b \in \mathbb{Z}_p$, an algorithm $\mathcal{A}$ has advantage ϵ in solving the CDH problem if $\Pr[\mathcal{A}(g, g^a, g^b) = g^{ab}] \geq \epsilon$.

We say that the (τ, ϵ)-CDH assumption holds if no τ-time algorithm has a non-negligible advantage ϵ in solving the CDH problem.

7.2 Security Model

We define the security model for TBV and ACS as follows:

Definition 7.2 (TBV Security Game (Selective EU-CMA Security)). We define the security of TBV under a selective existential unforgeability against chosen-message attacks (EU-CMA) game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ whose running time is probabilistic polynomial in a security parameter λ and threshold t as follows.

- **Initial**. In the initial phase, the adversary $\mathcal{A}$ specifies a set of compromised validators S_c where the maximum size of S_c is $t-1$.
- **Setup**. The challenger $\mathcal{C}$ runs the client registration algorithm and sends the system parameter sp to the adversary $\mathcal{A}$. For all the identities in the compromised set S_c, the secrets $\{h_{c,i}\}_{i \in S_c}$ will also be sent to the adversary $\mathcal{A}$.
- **Query**. The adversary makes *request proof* queries on requests that are adaptively chosen by the adversary on an identity that are not in the compromised set S_c.
- **Forgery**. The adversary $\mathcal{A}$ returns a forged request proof P^*_{req} on some request req^* that has not been queried. The adversary $\mathcal{A}$ wins the game if P^*_{req} can pass the request verification after running the ReqEval and ReqVerify algorithms.

Definition 7.3 (TBV Security). A threshold block validation scheme is (t, q_s, ϵ)-secure in the selective EU-CMA security model if there exists no adversary who can win the TBV security game in time t with non-negligible advantage ϵ after it has made q_s proof queries.

Definition 7.4 (ACS Security Game). We define the security of ACS under a selective existential unforgeability against chosen-message attacks (EU-CMA) game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ whose running time is probabilistic polynomial in a security parameter λ and threshold t as follows.

- **Initial**. In the initial phase, the adversary $\mathcal{A}$ specifies a set of compromised validators S_c where the maximum size of S_c is $t-1$.
- **Setup**. The challenger $\mathcal{C}$ runs the setup algorithm and sends the public parameter pp to the adversary $\mathcal{A}$. It runs the key generation algorithm to generate a set of validator signing key $k_{s,j}$ and a public verification key k_v. For all the validators in the compromised set S_c, the signing keys $\{k_{s,j}\}_{i \in S_c}$ will also be sent to the adversary $\mathcal{A}$.
- **Query**. The adversary makes signature queries on commit messages that are adaptively chosen by the adversary.
- **Forgery**. All the participating validators from the set of S_v sign the commit message m_i and sends it to Block-generator. Here, we consider a Block-generator acts as an adversary $\mathcal{A}$ which collects the σ_{j,m_i} from all participating validators and aims at forging the aggregate signature. The adversary $\mathcal{A}$ returns a forged aggregated signature σ^* on commit message m^* that has not been queried. The adversary $\mathcal{A}$ wins the game if σ^* can be publicly verified.

Definition 7.5 (ACS Security). An aggregate commitment signature scheme is (t, q_s, ϵ)-secure in the selective EU-CMA security model if there exists no adversary who can win the ACS security game in time t with non-negligible advantage ϵ after it has made q_s signature queries.

7.3 Security of TBV Scheme

Theorem 1 (Security of TBV)**.** *The proposed TBV scheme is selective EU-CMA secure under the Computational Diffie-Hellman assumption.*

Proof. Suppose there exists an adversary $\mathcal{A}$ who can (t, q_s, ϵ)-break the TBV scheme in the selective EU-CMA security model, i.e., $\mathcal{A}$ can forge a valid request proof P_{req} with non-negligible advantage, we can construct a simulator $\mathcal{B}$ in polynomial time to solve the CDH problem. Given as input a problem instance (g, g^a, g^b) over the pairing group $\mathbb{G}$, $\mathcal{B}$ controls the random oracle, runs $\mathcal{A}$, and works as follows.

Initial. In the initial phase, the adversary $\mathcal{A}$ specifies a set of compromised validators S_c where the maximum size of S_c is $t-1$.

Setup. Let public parameters be $pp = (G, G_T, g, p, e)$. The simulator $\mathcal{B}$ randomly chooses an identity ID and specify an $i^* \in [1, q_{H_1}]$ where q_{H_1} is the total number of query for the H_1 oracle. The password selected is the i^* query for the H_1 oracle. $\mathcal{B}$ also selects a random secret k from $\mathbb{Z}_p^*$ and shared it using a t-degree polynomial function $f(x)$ and obtain secret shares $\{k_1, \cdots, k_{n_v}\}$. The client key k_c is implicitly set as $k_c = a \cdot k$.

To simulate the secret $h_{c,i}$, the $\mathcal{B}$ will first simulate the two random oracles:

- $\mathcal{O}(H)$: $\mathcal{B}$ prepares a hash list, which is empty at the beginning, to record all queries and responses as follows: If req_i is already in the hash list, $\mathcal{B}$ responds to this query following the hash list. Otherwise, $\mathcal{B}$ randomly chooses w_i from $\mathbb{Z}_p^*$ and sets $H(req_i)$ as $H(req_i) = g^{w_i}$. The simulator $\mathcal{B}$ responds to this query with $H(req_i)$ and adds $(i, req_i, w_i, H(req_i))$ to the hash list.
- $\mathcal{O}(H_1)$: Let the i-th hash query be pwd_i. If pwd_i is already in the hash list, $\mathcal{B}$ responds to this query following the hash list. Otherwise, $\mathcal{B}$ randomly chooses y_i from $\mathbb{Z}_p^*$ and sets $H_1(pwd_i)$ as

$$H_1(pwd_i) = \begin{cases} g^{b+y_i} & \text{if } i = i^* \\ g^{y_i} & \text{otherwise} \end{cases}$$

  The simulator $\mathcal{B}$ responds to this query with $H_1(pwd_i)$ and adds $(i, pwd_i, y_i, H_1(pwd_i))$ to the hash list.

The secret $h_{c,i}$ will be simulated as $h_{c,i} = H_1(pwd_i)^{k_i}$. The simulator $\mathcal{B}$ will also share the compromised secrets $\{h_{c,i}\}_{i \in S_c}$ to the adversary.

Query. The adversary makes a request proof query in this phase. The proof can be simulated as

$$\begin{aligned} P_{req_i} &= \Big(H_1(pwd)^{k_c} H(req_i)^{u_i}, H(req_i)^{r_i}, g^{u_i}, g^{r_i}\Big) \\ &= \Big((g^{y_i})^{ak} H(req_i)^{u_i}, (g^{w_i})^{r_i}, g^{u_i}, g^{r_i}\Big) \\ &= \Big((g^a)^{y_i k} (g^{w_i})^{u_i}, (g^{w_i})^{r_i}, g^{u_i}, g^{r_i}\Big). \end{aligned}$$

where $u_i, r_i \xleftarrow{R} \mathbb{Z}_p^*$.

Forgery. The adversary $\mathcal{A}$ returns a forged request proof P_{req}^* on some request req^* that has not been queried. If the query of pwd^* to $\mathcal{O}(H_1)$ is not the i^*-th queried message in the hash list, abort. Otherwise, we have $H(pwd^*) = g^{b+r_{i^*} y_{i^*}}$. According to the request proof definition and simulation, a valid proof should follow the same structure as follows:

$$\begin{aligned} P_{req^*} &= \Big(H_1(pwd)^{k_c} H(req^*)^u, H(req^*)^r, g^u, g^r\Big) \\ &= \Big((g^{b+y_{i^*}})^{ak} (g^{w^*})^u, (g^{w^*})^r, g^u, g^r\Big). \end{aligned}$$

The simulator $\mathcal{B}$ computes

$$\frac{(g^{b+y_{i^*}})^{ak} g^{w^* u}}{(g^a)^{y_i k} g^{w_i u_i}} \cdot \frac{(g^{u_i})^{w_i}}{(g^u)^{w^*}} = g^{abk}$$

Then, the $g^{ab} = (g^{abk})^{1/k}$ as the solution to the CDH problem instance. This completes the simulation and the solution.

Indistinguishable Simulation. The correctness of the simulation has been explained above. The randomness of the simulation includes all random numbers in the key generation and the responses to hash queries:

$$k, a, y_1, \cdots, y_{i^*-1}, b + y_{i^*}, y_{i^*+1}, \cdots, y_{q_{H_1}}, w_1, \cdots, w_{q_H}.$$

According to the simulation, a, b, y_i, w_i, r_i, u_i are randomly chosen, it is easy to see that they are random and independent from the point of view of the adversary. Therefore, the simulation is indistinguishable from the real attack.

Probability of successful simulation and useful attack. If the simulator successfully guesses i^*, all queried request proofs are simulatable, and the forged request proof P_{req^*} is reducible because the password pwd_{i^*} cannot be chosen for a request proof query, and it will be used for the request proof forgery. Therefore, the probability of successful simulation and useful attack is $\frac{1}{q_{H_1}}$ for q_{H_1} queries.

Advantage and time cost. Suppose the adversary $\mathcal{A}$ breaks the TBV scheme with (t, q_s, ϵ) after making q_{H_1} queries to the random oracles H_1. The advantage of solving the CDH problem is therefore $\frac{\epsilon}{q_{H_1}}$. Let T_s denote the time cost of the simulation. We have $T_s = \mathcal{O}(q_{H_1} + q_s)$, which is mainly dominated by the oracle response and the request proof generation. Therefore, $\mathcal{B}$ will solve the CDH problem with $(t + T_s, \epsilon/q_{H_1})$. This completes the proof. □

7.4 Security of ACS Scheme

Theorem 2 (Security of ACS)**.** *The proposed ACS scheme is EUF-CMA secure under the Computational Diffie-Hellman assumption.*

Proof. Suppose there exists an adversary $\mathcal{A}$ who can (t, q_s, ϵ)-break the aggregate commitment signature scheme under the EU-CMA security model. The main focus of $\mathcal{A}$ is to forge the aggregated signature σ. We construct a simulator $\mathcal{B}$ to solve the CDH problem. Given as input a problem instance $(g, g^{a'}, g^b)$ over the pairing group $\mathbb{G}$, $\mathcal{B}$ controls the random oracle, runs $\mathcal{A}$, and works as follows.

Initial. In the initial phase, the adversary $\mathcal{A}$ specifies a set of compromised validators S_c where the maximum size of S_c is $t-1$.

Setup. Let public parameters be $(\mathbb{G}, \mathbb{G}_T, g, p, e, H_2, H_3, h)$, where H_2 be the random oracle controlled by simulator $\mathcal{B}$.

- $\mathcal{O}(H_2)$: $\mathcal{B}$ randomly chooses an integer $j^* \in [1, q_{H_2}]$ and set the challenge PKChain identity be $\mathsf{PKChainID}^* = \mathsf{PKChainID}_{j^*}$, where q_{H_2} represents the number of hash queries to the random oracle H_2. $\mathcal{B}$ prepares a hash list, where the hash list is empty at the beginning, to record all queries and responses as follows: If $\mathsf{PKChainID_j}$ is already in the hash list, $\mathcal{B}$ responds to this query following the hash list. Otherwise, $\mathcal{B}$ randomly chooses h_j from $\mathbb{Z}_p^*$ and set $H_2(\mathsf{PKChainID_j})$

$$H_2(\mathsf{PKChainID_j}) = \begin{cases} g^b & \text{if } j = j^* \\ h_j & \text{otherwise} \end{cases}$$

  The $\mathcal{B}$ responds to the query with $H_2(\mathsf{PKChainID_j})$ and adds $(j, \mathsf{PKChainID_j}, H_2(\mathsf{PKChainID_j})$ to the hash list.
- $\mathcal{O}(H_3)$: Before receiving queries for commitment message m_i, $\mathcal{B}$ randomly chooses an integer $i^* \in [1, q_{H_3}]$, where q_{H_3} represents the number of hash queries to the random oracle H_3. Then, $\mathcal{B}$ prepares a hash list to record all queries and responses as follows, where the hash list is empty at the beginning. Let the $i-th$ hash query be m_i. If m_i is already in the hash list, $\mathcal{B}$ responds to this query following the hash list. Otherwise, $\mathcal{B}$ randomly chooses w_i from $\mathbb{Z}_p^*$ and sets $H_3(m_i)$ as

$$H_3(m_i) = \begin{cases} g^{w_i} & \text{if } i = i^* \\ g^{b+w_i} & \text{otherwise} \end{cases}$$

  The simulator $\mathcal{B}$ responds to this query with $H_3(m_i)$ and adds $(i, m_i, w_i, H_3(m_i))$ to the hash list.

Then the simulator $\mathcal{B}$ chooses a random secret $s \in \mathbb{Z}_p^*$ and share s into multiple pieces $(s_1, \cdots, s_{n_v})$. Instead of running the ACS.KeyGen, the simulator can simply generate these secret shares. The secret key $k_{s,j}$ is define as follows based on CDH problem instance: $k_{s,j} = (g^b)^{a' \cdot s_j} = g^{b \cdot a_j}$, where $a_j = a' \cdot s_j$ and $g^{a_j} = (g^{a'})^{s_j}$. The public verification key k_v is simulated as $k_v = g^{a' \cdot s} = g^a$, where $a = a' \cdot s$. The secret key s is similar to a as in the CDH problem. The public key is available from the problem instance.

Query. The adversary makes signature queries in this phase. For a signature query on m_i, if m_i is the i^*-th queried request message in the hash list, abort. Otherwise, we have $H_3(m_i) = g^{b+w_i}$. We can simulate the signature σ_j for message m_i as follows:

- *Simulation of* $\sigma_{j,1}$: $\mathcal{B}$ randomly chooses $r'_{ij} \in \mathbb{Z}_p^*$, set $r_{ij} = r'_{ij} - a_j$, and computes $\sigma_{j,1}$ as

$$\begin{aligned} \sigma_{j,1} &= g^{b \cdot a_j} \cdot H(m_i)^{r_{ij}} = g^{b \cdot a_j} \cdot (g^{b+w_i})^{r'_{ij} - a_j} \\ &= (g^{a_j})^{-w_i} \cdot H(m_i)^{r'_{ij}} \end{aligned}$$

- *Simulation of* $\sigma_{j,2}$:

$$\sigma_{j,2} = g^{r_{ij}} = g^{r'_{ij} - a_j} = g^{r'_{ij}} \cdot (g^{a_j})^{-1}$$

- *Aggregation*: Upon receiving all the σ_j from Block-validators, the simulator $\mathcal{B}$ computes the signatures aggregation as:

$$\begin{aligned} \sigma_1 &= \prod_j \sigma_{j,1}^{c_j} = \prod_j (g^{a_j})^{-w_i \cdot c_j} \cdot H(m_i)^{r'_{ij} \cdot c_j} \\ \sigma_2 &= \prod_j \sigma_{j,2}^{c_j} = \prod_j g^{r'_{ij} \cdot c_j} \cdot g^{-a_j \cdot c_j} \end{aligned}$$

- *Validation of Simulated and Aggregated Signatures:* The aggregated signatures can be validated as:

$$\begin{aligned} e(\sigma_1, g) &= e\Big(\prod_j g^{-a_j \cdot w_i \cdot c_j}, g\Big) \cdot e\Big(H(m_i)^{\sum_j r'_{ij} \cdot c_j}, g\Big) \\ &= e\Big(H_3(m), \prod_j g^{r'_{ij} \cdot c_j}\Big) \cdot e\Big(g^{w_i}, g^a\Big) \\ &= e(H_3(m), \sigma_2) \cdot e(H_2(\mathsf{PKChainID}), k_v). \end{aligned}$$

Forgery. In a real scenario, the σ_{j,m_i} is a single signature created by validator j for commit message m_i. All the participating validators from the set of S_v sign the commit message m_i and sends it to Block-generator. Here, we consider a Block-generator acts as an adversary $\mathcal{A}$ which collects the σ_{j,m_i} from all participating validators and aims at forging the aggregate signature.

The adversary $\mathcal{A}$ returns a forged aggregated signature σ^* on commit message m^* that has not been queried. If m^* is not the i^*-th queried message in the hash list, abort.

Otherwise, we have $H_3(m^*) = g^{w_{i^*}}$. According to the request signature definition and simulation, a valid signature follows the following structure:

$$\sigma_1^* = (g^{ab} \cdot H(m^*)^r, g^r)$$

Then, the simulator $\mathcal{B}$ can compute

$$\left(\frac{\sigma_1^*}{(\sigma_2^*)^{w_i^*}}\right)^{1/s} = \left(\frac{g^{ab} \cdot g^{w_i^* \cdot r}}{g^{w_i^* \cdot r}}\right)^{1/s} = \left(g^{ab}\right)^{1/s} = g^{a'b}$$

as the solution to the CDH problem. This completes the simulation and the solution.

Indistinguishable Simulation. The correctness of the simulation has been explained above. The randomness of the simulation includes all random numbers in the key generation and the responses to hash queries, and the signature generation. They are as follows:

$$a, b, a', s, r'_{ij}, b+w_1, \cdots, b+w_{i^*-1}, w_{i^*}, b+w_{i^*+1}, \cdots, b+w_{q_{H_3}}$$

According to the setting of the simulation, where a, a', b, w_i, r'_{ij} are randomly chosen, it is easy to see that they are random and independent from the point of view of the adversary. Therefore, the simulation is indistinguishable from the real attack.

Probability of successful simulation and useful attack. If the simulator successfully guesses i^*, all queried signatures are simulatable, and the forged signature is reducible because the commit message m_{i^*} cannot be chosen for a request signature query, and it will be used for the signature forgery. Similarly, for the query of PKChainID*. Therefore, the probability of successful simulation and useful attack is $\frac{1}{q_{H_2}+q_{H_3}}$ for q_{H_2} and q_{H_3} queries.

Advantage and time cost. Suppose the adversary $\mathcal{A}$ breaks the ACS scheme with (t, q_s, ϵ) after making q_{H_2} and q_{H_3} queries to the random oracles H_2 and H_3. The advantage of solving the CDH problem is $\frac{\epsilon}{q_{H_2}+q_{H_3}}$. Let T_s denote time cost of the simulation. We have $T_s = \mathcal{O}(q_{H_2}+q_{H_3}+q_s)$, which is mainly dominated by the oracle response and the signature generation. Therefore, $\mathcal{B}$ will solve the CDH problem with $(t + T_s, \epsilon/(q_{H_2} + q_{H_3}))$. This completes the proof. □

7.5 Security of pBCTV consensus model

Theorem 3. *The practical Byzantine Compromise-tolerant and Verifiable consensus model with n_v validators can tolerate at most $\lfloor \frac{n_v-1}{3} \rfloor$ compromised validators if the threshold block validation scheme and the aggregate commitment signature scheme is EUF-CMA secure.*

Proof. Suppose there are n_c compromised validators in the system, all the n_c compromised validators may not respond to any messages in both Prepare and Commit phases in the pBCTV consensus mechanism. In order to finish the consensus procedure, each validator will continue to the next phase upon receiving at least $n_v - n_c$ different messages. However, sometimes the messages from uncompromised validators may be delayed, and the messages from the compromised validators are counted in the received messages that are carried for the next phase. That is, there may be n_c messages from the compromised validators among those who received $n_v - n_c$ messages. To guarantee that at lease $n_c + 1$ messages are valid messages, we requires $n_v - 2n_c \geq n_c + 1$. Thus, we have $n_c \leq \lfloor \frac{n_v-1}{3} \rfloor$ and the threshold t can be set to $n_c + 1$ or any value between $[n_c + 1, n_v - 2n_c]$. If both TBV and ACS schemes are existentially unforgeable against the chosen message attack, then any adversary, including the compromised validators, cannot forge a valid public key request and a valid signature of a public key record in the blockchain. □

TABLE 1
Communication Overhead

Process	pBCTV	pBFT
Initialization	$n_v(n_v-1)\cdot 4\|p\|$	n/a
Enrollment	$n_v \cdot 2\|p\|$	n/a
Proposal	$\|req\|+4\|p\|$	$\|req\|$
Pre-Prepare	$n_v(\|req\|+4\|p\|)$	$n_v\|req\|$
Prepare	$n_v(n_v-1)\cdot 2\|p\|$	$n_v(n_v-1)\|vote\|$
Commit	$n_v(n_v-1)(\|m\|+2\|p\|)$	$n_v(n_v-1)\|m\|$

8 Performance Evaluation

We develop a prototype of PKChain and implement the schemes in Python. The TBV and ACS schemes are implemented using the charm framework and pairing-based cryptography library (pbc-0.15.14), GPM 6.2.0, and OpenSSL 1.1.1. For EC with bilinear maps or pairing the NIST-approved curve, MNT159 representing an asymmetric curve with the 159-bit base field is used, together with Hashlib(SHA256) library for secure hashes and digest. We utilized the round-robin-0.0.1 package for round-robin protocol implementation. We also implemented pBFT to use as a baseline and compare the performance evaluation between pBCTV and pBFT.

We run the experimental workloads on 20 local standalone Docker-1.3.0 containers; each represented a validator node. One node is elected as a Block-generator, and the remaining nodes work as Block-validators in each round. We used the urllib3 library for client implementation. The Docker-1.30 hosts on Intel(R) CPU @ 1.6 GHz Dual-Core Intel Core i5, 8GB RAM, and hosts macOS Catalina (10.15.17) operating system.

8.1 Communication Overhead

We evaluate the communication overhead in each stage of the PKChain (especially the pBCTV consensus model), which is compared to the communication overhead of the pBFT consensus model. The communication overhead of each process is shown in Table 1 where p is an element size in the group $\mathbb{G}$, $|req|$ is the size of public key request, $|vote|$ is the size of vote message in pBFT, and $|m|$ is the size of commitment message. We can see that our proposed pBCTV consensus model does not increase much communication overhead compared to the pBFT consensus model. This is because both the threshold block validation scheme and the aggregate commitment signature scheme fully use of the existing message exchange in the pBFT protocol.

8.2 Computation Cost

We evaluate block confirmation time between pBCTV and pBFT. Figure 3 shows when the number of validators increases, in the presence of compromised validator nodes,

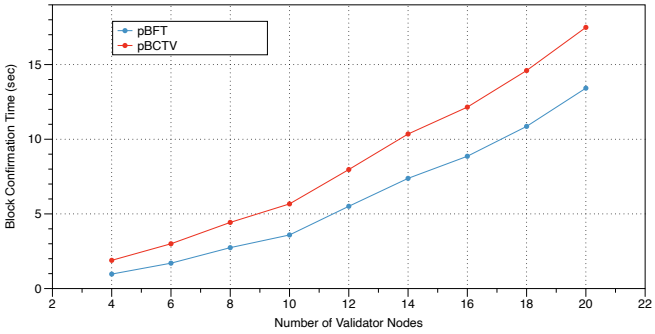

Fig. 3. Comparison of Block Confirmation Time

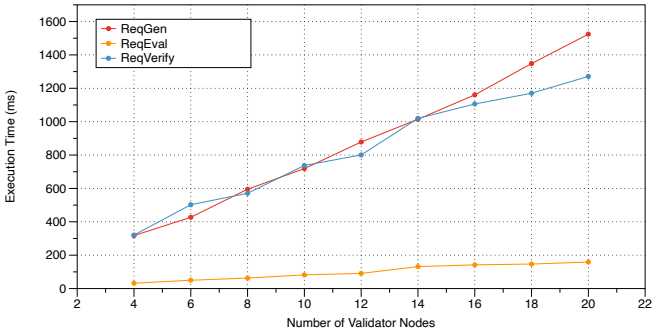

Fig. 4. Computation Cost of TBV

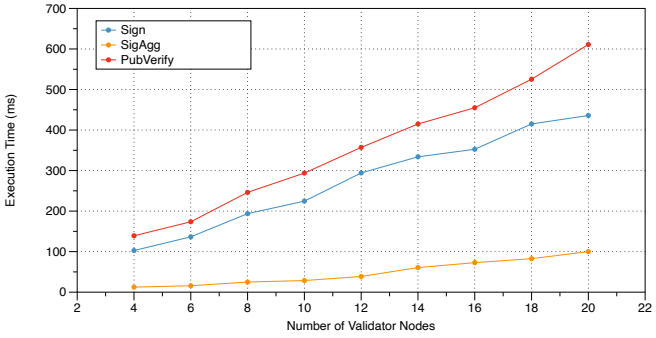

Fig. 5. Computation Cost of ACS

communication overhead increases, which is obvious from the BFT $(n \times n)$ message broadcast. There is an expected overhead in pBCTV compared to pBFT due to the cryptographic operations. Our pBFT implementation uses MAC similar to pBFT [30], instead of digital signature, while pBCTV involves digital signatures based on EC bilinear pairing. There is a trade-off between security and performance, and to achieve the design goal of PKChain, a certain level of overhead is caused due to cryptographic operations. Previous works (e.g., Zyzzyva [31] or pBFT [29]) have shown that the bottleneck in BFT protocols is actually cryptography, not network usage [32]. Concerning throughput, pBCTV leverages pBFT batching in the form of a block to lower bound on the number of authentication operations performed during consensus.

We evaluate the computation cost of each algorithm proposed in the pBCTV. Figure 4 describes the executing time of each algorithm on each validator in the threshold block validation scheme. We can see that only the TBV.ReqVerify is dependent on the number of validators because it has to collect a set of request evaluations $\{Sig_i\}$ from at least t uncompromised validators. Given the received $n_v - n_c$ request evaluations, the probability that a validator selects a set consisting of t uncompromised request evaluations is $1/C^t_{n_v-n_c}$. Therefore, in the worst case, a validator has to repeat all the $C^t_{n_v-n_c}$ combinations and outputs 'NO' if all the combinations fail on the verification. That is, as long as one set with t request evaluations passes the request verification algorithm, it outputs 'YES'.

Figure 5 shows the executing time of each algorithm on each validator, where only the ACS.SigAgg is dependent on the number of validators. Similarly, during the signature aggregation, the Block-generator (and each Block-validator) will able to compute the aggregated signature with a set of t signatures from uncompromised validators. Based on the result of individual signature validation, it is easy to select such a set for signature aggregation.

8.3 Scalability Analysis

The pBCTV consensus mechanism is designed based on the pBFT protocol, where the message exchange is the main concern when scaling to a large network. The threshold block validation scheme is integrated into the pre-prepare and prepare phases of the pBFT protocol, so the pBCTV is as efficient as the pBFT protocol. The scalability of PKChain is further analyzed as follows:

Number of Validators in PKChain: PKChain is a permissioned blockchain. It is usually not required that everyone participate in the consensus process. The number of validators n_v depends on the number of compromised validators n_c that can be tolerant, i.e., $n_v \geq 3n_c + 1$.

New Message Broadcasting Protocol: Instead of using P2P communication between nodes, some data synchronization methods may be used to improve the efficiency of message broadcasting, e.g., the synchronization method [33] designed based on Name Data Networking [34].

Dynamic Change of Validators: In PKChain, removal of a malicious validator will not affect the threshold value. When a new validator is added, secret shares can be redistributed among validators including the newly joined validator using a key re-distribution mechanism (similar to the ACS.KeyGen), which does not require to re-bootstrap the system.

Message Complexity: There are some other approaches that reduce the $\mathcal{O}(n^2)$ message complexity in the BFT algorithms. For instance, chain-based approach [35], [32] (to use multiple instances run in parallel), tree-based approach [36] (where leader becomes root of the tree and other nodes services as child nodes), the collector-based approach [37] where a set of nodes serves a collector to collect the vote in prepare and commit phases of pBFT, or a set of root committee [38] perform BFT operation and the rest of nodes services passively in the network. We are planning to leverage one of these techniques in our future work to improve the scalability of the PKChain while maintaining security.

PKChain Compatibility with CA-based PKI: To better scale, reduce the complexity of the public key blockchain, and makes PKChain compatible with CA-based PKI, we keep

the hierarchical structure in PKChain similar to CA-based PKI. In PKChain, the public keys of all the validators in $\mathsf{PKChain}_{\mathsf{univ}}$ are managed by the $\mathsf{PKChain}_{\mathsf{edu}}$, and similar for other domains allowing structure compatibility with CA-based PKI. Therefore, PKChain can work with the existing CA-based PKI by replacing an arbitrary number of CAs with PKChains. If PKChain replaces all the CAs, it will be a new public key management system that completely replaces CA-based PKI. Moreover, to make existing CA-based PKI applications compatible with PKChain. Each public-key record in PKChain is publicly verifiable similar to the CA-based PKI.

9 Discussion on PKChain v.s. Threshold Co-Signing

PKChain solves the following two technical challenges that we assume it is not possible to use threshold signature directly.

Challenge 1. We assume there is no initial PKI setup between users and validators. So, validators may not have credentials like public keys to authenticate and validate user certificate requests, To deal with this challenge, typically, there are two approaches to perform validations (1) each CA may contact the principal (client) and do name-principal validation through some out-of-band channels for each request, as described in [27], which will incur a heavy burden on the user; and (2) a user authentication method may enable each CA to verify that public key request is from a legitimate user holding valid credentials (e.g., a private key or password).

However, in our paper, we claimed that signature-based validation is not always applicable due to the lack of private-public key pair (e.g., new public key registration request) or compromised private key (e.g., public key update/revocation requests. One approach to this problem is that a user shares an initial public key (without a certificate) to each validator through an authenticated and confidential channel. This simple approach presumably works, but it also requires the name-principal validation procedure. Since the secure channel is a one-way authenticated channel, enabling only the user to authenticate each validator. As a result, an adversary may forge a pair of public and secret keys. To further authenticate whether the request comes from a valid user, the validator has to contact the user via a different channel. Following this signature-based approach, each validator is able to complete the validation of a request, and a trusted coordinator has to collect a threshold number of partial signatures (votes). One can use a consensus mechanism like a practical Byzantine Fault-Tolerant (pBFT) to eliminate a trusted coordinator.

In this paper, we took an alternative approach to signature-based authentication. We solved the above challenges by designing a Threshold Block Validation (TBV) mechanism, which is a collaborative password-based authentication approach. Password-based authentication is considered convenient and user-friendly compared to private key management in signature-based authentication, signifying that no security devices are required to store the signing key in signature-based authentication. However, unlike signature-based authentication, in password-based authentication, each CA may host the completed user credentials (e.g., security token), enabling a compromised CA to easily impersonate a user with such completed user credentials and produce a forged public key request (certificate).

To further compare the above two approaches, the (signature-based authentication + pBFT) scheme needs to receive a $2f + 1$ (f is the maximum number of fault/compromised validators) number of messages to move to the commit phase. In our proposed (password-based authentication + pBCTV) scheme, it only needs a set of $f + 1$ messages from honest validators, which means that a validator can proceed to the commit phase if it receives $f + 1$ (best case) to $2f + 1$ (worst case) messages.

Challenge 2. As shown in the COMMIT phase of the pBCTV, each validator locally aggregates all the received signatures on the commit message. Suppose a malicious block-generator generates a fake aggregate signature. In that case, any honest validator can verify and compare the fake aggregated signature with its own aggregated signature and initiate a Type-2 View-change and reshare the aggregate signature. This will not be directly possible without a consensus mechanism and will require each validator to reshare its signature with a newly selected aggregator (in the presence of a malicious aggregator). However, in PKChain, each validator already has all the signatures and does not require to restart the partial signature-sharing process again.

10 Conclusion

In this paper, we proposed a novel compromise-tolerant and verifiable public key management system, namely PKChain, which can provide transparent, tamper-proof, and verifiable public key services, including public key registration, update, query, validation, and revocation. To validate public key requests, we proposed a threshold block validation scheme, where a majority of block validators can work together to validate if the request comes from a valid user. We also proposed an aggregate commitment signature scheme to enable the public verification of the public key. We further design a new pBCTV consensus model by integrating the threshold block validation scheme and the aggregate commitment signature scheme with the pBFT protocol. The security analysis and prototype implementation show that PKChain is secure, efficient, robust, and provides proactive security.

References

[1] "Comodo group inc.: Comodo report of incident," https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html, (March 2011).

[2] N. van der Meulen, "Diginotar: Dissecting the first dutch digital disaster," *Journal of Strategic Security*, vol. 6, no. 2, pp. 46–58, 2013.

[3] R.-G. Holz, "Empirical analysis of public key infrastructures and investigation of improvements," Ph.D. dissertation, Technische Universität München, 2014.

[4] J. Stapleton, "PKI_under_attack_issa0313.pdf." [Online]. Available: https://cdn.ymaws.com/www.issa.org/resource/resmgr/JournalPDFs/PKI_Under_Attack_ISSA0313.pdf

[5] C. Ellison and B. Schneier, "Ten risks of pki: What you're not being told about public key infrastructure," *Comput Secur J*, vol. 16, no. 1, pp. 1–7, 2000.

[6] B. Laurie, A. Langley, and E. Kasper, "Rfc6962: Certificate transparency," *Request for Comments. IETF*, 2013.

[7] C. Fromknecht, D. Velicanu, and S. Yakoubov, "A decentralized public key infrastructure with identity retention." p. 803, 2014.

[8] S. Matsumoto and R. M. Reischuk, "Ikp: Turning a pki around with blockchains." *IACR Cryptology ePrint Archive*, vol. 2016, p. 1018, 2016.

[9] B. Qin, J. Huang, Q. Wang, X. Luo, B. Liang, and W. Shi, "Cecoin: A decentralized PKI mitigating MitM attacks," *Future Generation Computer Systems*, 2017.

[10] M. Al-Bassam, "Scpki: A smart contract-based pki and identity system," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. ACM, 2017, pp. 35–40.

[11] M. Ali, J. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains," in *2016 USENIX Annual Technical Conference*, 2016, pp. 181–194.

[12] N. Blagov and M. Helm, "State of the certificate transparency ecosystem," *Network*, vol. 43, 2020.

[13] "What is certificate transparency? - certificate transparency," https://certificate.transparency.dev/.

[14] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[15] Ethereum, https://www.ethereum.org/, 2017.

[16] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency-rfc 6962," *IETF RCFs*, 2013.

[17] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities" honest or bust" with decentralized witness cosigning," in *2016 IEEE Symposium on Security and Privacy (SP)*. Ieee, 2016, pp. 526–545.

[18] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail." in *NDSS*, 2014, pp. 1–14.

[19] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable key infrastructure (aki) a proposal for a public-key validation infrastructure," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 679–690.

[20] P. Szalachowski, S. Matsumoto, and A. Perrig, "Policert: Secure and flexible tls certificate management," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 406–417.

[21] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "Arpki: Attack resilient public-key infrastructure," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 382–393.

[22] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "{CONIKS}: Bringing key transparency to end users," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 383–398.

[23] S. Garfinkel, *PGP: pretty good privacy*. O'Reilly Media, Inc., 1995.

[24] L. Zhou, F. B. Schneider, and R. Van Renesse, "Coca: A secure distributed online certification authority," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 329–368, 2002.

[25] A. Yakubov, W. Shbair, A. Wallbom, D. Sanda *et al.*, "A blockchain-based pki management framework," in *The First IEEE/IFIP International Workshop on Managing and Managed by Blockchain (Man2Block) colocated with IEEE/IFIP NOMS 2018, Tapei, Tawain*, 2018.

[26] M. Toorani and C. Gehrmann, "A decentralized dynamic pki based on blockchain," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1646–1655.

[27] K. Yang, J. J. Sunny, and L. Wang, "Blockchain-based decentralized public key management for named data networking," in *The International Conference on Computer Communications and Networks*, 2018.

[28] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2003, pp. 416–432.

[29] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[30] M. Castro and B. Liskov, "Authenticated byzantine fault tolerance without public-key cryptography," Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, Tech. Rep., 1999.

[31] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, pp. 1–39, 2010.

[32] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 297–306.

[33] T. Li, W. Shang, A. Afanasyev, L. Wang, and L. Zhang, "A brief introduction to NDN Dataset Synchronization (NDN Sync)," in *IEEE MILCOM*, 2018.

[34] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[35] S. Duan, H. Meling, S. Peisert, and H. Zhang, "Bchain: Byzantine replication with high throughput and embedded reconfiguration," in *International Conference on Principles of Distributed Systems*. Springer, 2014, pp. 91–106.

[36] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 139–151, 2018.

[37] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: a scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 568–580.

[38] M. M. Jalalzai, C. Busch, and G. G. Richard, "Proteus: a scalable bft consensus protocol for blockchains," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019, pp. 308–313.