PROCESS: Privacy-Preserving On-Chain Certificate Status Service

Meng Jia[1], Kun He[1], Jing Chen[1], Ruiying Du[1], Weihang Chen[1], Zhihong Tian[2], Shouling Ji[3]
[1]*School of Cyber Science and Engineering, Wuhan University, Wuhan, China*
[2]*Guangzhou University, Guangzhou, China*
[3]*Zhejiang University, Hangzhou, China*
Email:{jiameng, hekun, chenjing, chenweihang}@whu.edu.cn, duraying@126.com, tianzhihong@gzhu.edu.cn, sji@zju.edu.cn

***Abstract*—Clients (e.g., browsers) and servers require public key certificates to establish secure connections. When a client accesses a server, it needs to check the signature, expiration time, and revocation status of the certificate to determine whether the server is reliable. The existing solutions for checking certificate status either have a long update cycle (e.g., CRL, CRLite) or violate clients' privacy (e.g., OCSP, CCSP), and these solutions also have the problem of trust concentration. In this paper, we present PROCESS, an online privacy-preserving on-chain certificate status service based on the blockchain architecture, which can ensure decentralized trust and provide privacy protection for clients. Specifically, we design *Counting Garbled Bloom Filter* (CGBF) that supports efficient queries and *Block-Oriented Revocation List* (BORL) to update CGBF timely in the blockchain. With CGBF, we design a privacy-preserving protocol to protect clients' privacy when they check the certificate statuses from the blockchain nodes. Finally, we conduct experiments and compare PROCESS with another blockchain-based solution to demonstrate that PROCESS is suitable in practice.**

I. Introduction

Public Key Infrastructure (PKI) plays an important role in today's Internet, in which Certificate Authorities (CAs) manage public key certificates for authentication and establishment of secure connections. For example, in the Hypertext Transfer Protocol Secure (HTTPS), a browser authenticates the accessed server by the server's certificate, and establishes a TLS (Transport Layer Security) [1] connection with that server.

However, the status of a certificate authorized by a CA may become invalid in various circumstances, even before the expiration time set by the CA [2]. For example, the private key of the server (which corresponds to the public key in the server's certificate) is compromised. In this case, CAs need to revoke the invalid certificates and publish the revocations to clients (e.g., browsers). Otherwise, clients may still consider those certificates as valid, and attackers can perform man-in-the-middle attacks accordingly, such as tampering with information and forging identity [3].

The most common mechanisms for publishing revocations are *Certificate Revocation List* (CRL) [2] and *Online Certificate Status Protocol* (OCSP) [4]. In CRL, CAs periodically push a large list of revoked certificates to clients for checking certificate statuses. However, due to the high update latency and growing network overhead, CRL may be superseded by a more efficient alternative, OCSP. In OCSP, a client can run an on-demand query for a single certificate's status to an OCSP server, rather than downloading the entire list of the revoked certificates. Unfortunately, there are two urgent problems in existing OCSP mechanisms: *trust concentration* [5] and *privacy leakage*. Trust concentration concerns the compromise of the centralized OCSP server, in which incorrect certificate statuses may then be returned to clients, while privacy leakage refers to the problem that an OCSP server can learn which server (e.g., the website) the querying client is interested in from its on-demand certificate status query.

Our goal is to solve above two problems in OCSP. The blockchain technology, famous for having both distributed storage and anonymity features, seems an ideal solution. However, applying blockchain in this field is not trivial due to the following challenges. First, the practical application is heavily constrained because of the *limited block size*, when confronting massive data storage. Existing solutions based on the blockchain technology, such as CertLedger [6], ECBC [7] and CertChain [8], encode all the revoked certificates into a data structure (e.g., Merkle Hash Tree (MHT), MPT-Chain, and Bloom filter), and store this structure in a block. Due to the huge number of certificates (Let's Encrypt [9] had issued over 538 million certificates for 223 million domain names by January 2019), even those highly compact data structures still cannot be stored in a relatively small block and are inefficient in practice. Therefore, we need to design a suitable on-chain data structure when solving the trust concentration problem through the blockchain architecture. Second, the *privacy leakage* problem [10] is more subtle when we apply the blockchain technology in OCSP. Due to the sensitivity of certificates, solutions need to be designed in the form of an alliance chain, which means that only OCSP servers act as the blockchain nodes. Clients who check certificate statuses do not participate in the blockchain, and thus the anonymity feature of blockchain cannot protect clients' privacy [8]. Some solutions (e.g., CertLedger [6]) employ OCSP Must Staple mechanism [11, 12] to protect privacy. However, OCSP Must Staple requires a custom extension for established and standardized certificates, and thus it is not widely adopted [13]. Third, existing solutions for privacy protection (e.g., Private Information Retrieval (PIR) [14]) has a *high latency* because of their complex calculations [15]. Clients may be reluctant to check the certificate status due to the poor efficiency and high

latency [16].

In this paper, we propose a PRivacy-preserving On-chain CErtificate Status Service, called PROCESS. Specifically, CAs upload the information of the revoked certificates to blockchain nodes, which update global revocation statuses by data structures designed for online revocation status query. A client queries the blockchain nodes for the revocation status through a privacy protection protocol, where the client can set parameters to balance privacy protection and latency requirements. The main contributions are as follows.

- Towards the limited block size challenge, we propose an on-chain data structure called *Block-Oriented Revocation List* (BORL), which is to accommodate *Counting Garbled Bloom Filter* (CGBF) and update CGBF timely in the blockchain architecture, where CGBF stores global revocation statuses for efficient query and supports both insertion and deletion.
- Regarding the privacy leakage and high latency challenges, we propose an efficient privacy-preserving on-chain certificate status check protocol. Moreover, clients can set parameters in the protocol to balance privacy requirements and latency limitations.
- We conduct experiments and analyze the performance of PROCESS. Compared with the existing work [8], PROCESS has advantages in terms of space cost, privacy, and latency.

This paper is organized as follows. In Section II, we describe the models, design goals, and building blocks. Then, we present CGBF and BORL in Section III and PROCESS in Section IV. In Section V and Section VI, we analyze the security and evaluate the performance of our proposal, respectively. Finally, we review related work in Section VII and conclude in Section VIII.

II. Preliminaries

A. System Model

PROCESS consists of four entities: *certificate authority* (CA), *server*, *client*, and *blockchain node*, as shown in Fig. 1.

- A CA is responsible for authorizing, updating, and revoking public key certificates for servers.
- A server (e.g., a website) publishes information or provides network services to clients.
- A client (e.g., a browser) checks the validity of the certificates before it establishes secure connections (e.g., TLS connections) with servers.
- Blockchain nodes are designed to maintain a distributed ledger storing the certificate statuses from CAs and reply to the certificates' status queries from clients.

B. Threat Model

From the practical perspective, we assume that the capabilities of an adversary are as follows.

- The adversary can eavesdrop, tamper, and forge messages between communicating entities in an untrusted network.

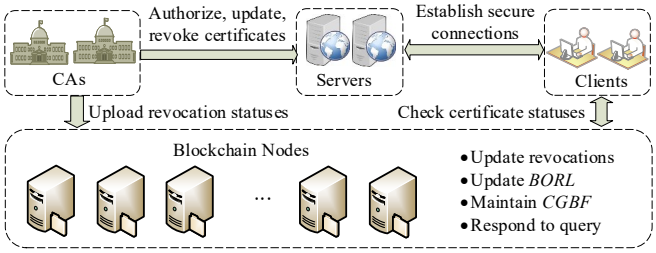

Fig. 1. Deployment of PROCESS

- The adversary can control a number of blockchain nodes, but cannot control more than 51% computing power in blockchain.
- The adversary cannot obtain the chameleon hash private keys of the blockchain nodes, and cannot sign the revocation information or calculate the random number of the chameleon hash without the entity's private key.

C. Design Goals

We aim to design a certificate status service that satisfies the following goals.

- **Timeliness.** After a CA uploads the revoked information to the blockchain nodes, the nodes can update the global information in time.
- **Privacy.** The client does not disclose the information of the server that it plans to access when querying the certificate status.
- **Compatibility.** PROCESS can be compatible with current PKI architecture without any custom extensions for certificates.

D. Chameleon Hash

A chameleon hash function is a hash function that allows one to find arbitrary collisions with a trapdoor [17, 18]. We employ the following chameleon hash scheme $\mathcal{CH} = (\mathsf{Gen}, \mathsf{ReHash}, \mathsf{Hash}, \mathsf{Check}, \mathsf{Adapt})$.

- $\mathsf{Gen}(1^\lambda)$. With the input of the security parameter λ, this algorithm generates a group $\mathbb{G}$ of prime order p with a generator g. Then, it chooses a random $x \in \mathbb{Z}_p$ and outputs the private key $sk \leftarrow x$ and public key $pk \leftarrow g^x$.
- $\mathsf{ReHash}(pk, msg, r)$. With the input of the public key pk, a message msg, and a randomness r, this algorithm outputs the hash $h \leftarrow g^{msg} pk^r$.
- $\mathsf{Hash}(pk, msg)$. With the input of the public key pk and a message msg, this algorithm outputs a randomness $r \in \mathbb{Z}_p$ and the hash $h \leftarrow \mathsf{ReHash}(pk, msg, r)$.
- $\mathsf{Check}(pk, msg, r, h)$. With the input of the public key pk, a message msg, a randomness r, and a hash h, this algorithm outputs 1 if $h = g^{msg} pk^r$ and 0 otherwise.
- $\mathsf{Adapt}(sk, msg, r, h, msg')$. With the input of the private key sk, a message msg, a randomness r, a hash h, and a new message msg', this algorithm outputs a new randomness $r' \leftarrow (msg + r \cdot sk - msg')/sk$.

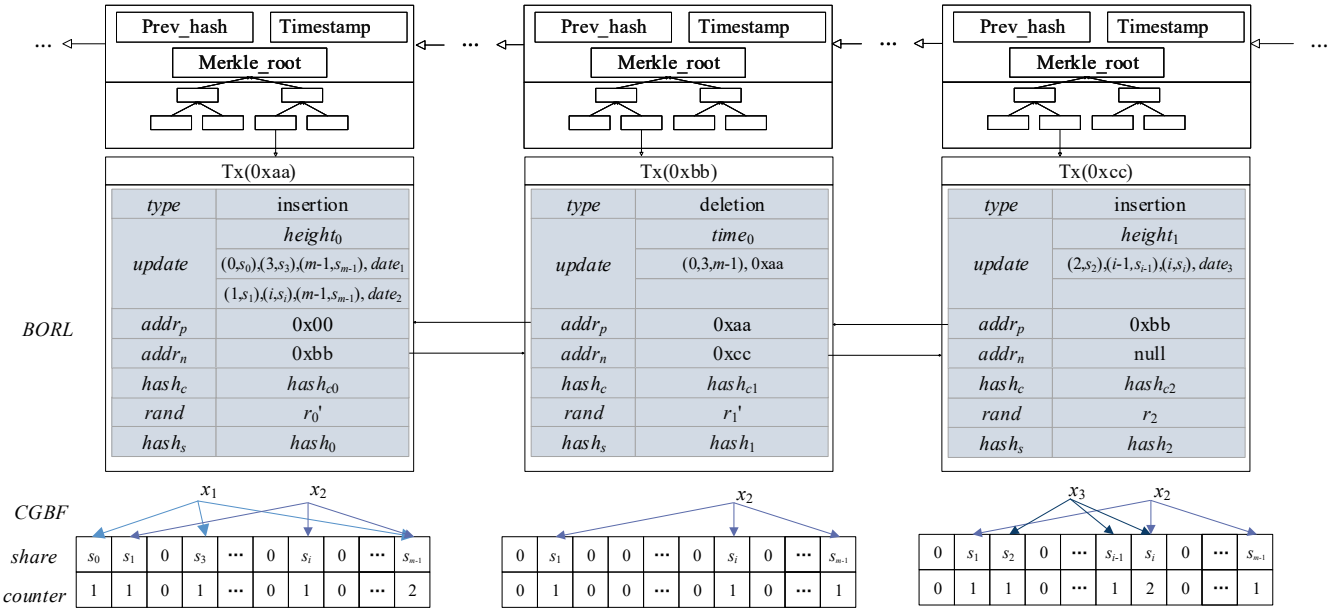

Fig. 2. An illustration of $BORL$ and $CGBF$ where $k = 3$. First, we insert $x_1 = s_0 \oplus s_3 \oplus s_{m-1}, x_2 = s_1 \oplus s_i \oplus s_{m-1}$ into $CGBF$ and generate a $BORL$ node in a transaction addressed by $0xaa$. Then, we delete x_1 from $CGBF$ in $0xbb$, and insert $x_3 = s_2 \oplus s_{i-1} \oplus s_i$ into CGBF in $0xcc$.

E. Digital Signature

Let $\mathcal{DS} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme, such as RSA. The key generation algorithm $\mathsf{Gen}(1^\lambda)$ takes as input a security parameter λ, and outputs a pair of public-private keys (pk, sk). The signing algorithm $\mathsf{Sign}(sk, msg)$ takes as input the private key sk and a message msg, and outputs a signature sig. The verification algorithm $\mathsf{Verify}(pk, (msg, sig))$ takes as input the public key pk and a message-signature pair (msg, sig), and outputs a bit where 0/1 indicates that the message-signature pair is invalid/valid.

III. Revocation Status Structure

We propose a data structure for storing and retrieving certificate statuses in the blockchain architecture. Roughly speaking, the revoked certificates are encoded into a structure, called *Counting Garbled Bloom Filter* (CGBF). Then, CGBF is decomposed into a number of structured pieces for block storage, called *Block-Oriented Revocation List* (BORL). The data structures of CGBF and BORL are shown in Fig. 2.

A. Counting Garbled Bloom Filter

Essentially, retrieving a certificate status is testing whether a given certificate is in the set of revoked certificates. Garbled Bloom Filter (GBF) is an efficient data structure for membership testing, whose false positive is negligible compared to traditional Bloom Filter [19]. Also, inserting new revocation statuses into GBF does not change existing data. Therefore, it looks that we can store the insertion into new blocks instead of rewriting old ones, if we store GBF directly in blocks. Unfortunately, GBF does not support deletion, which means that revocation statuses cannot be deleted from GBF when they are expired. The increasing number of revoked certificates will quickly reach the upper bound of GBF and cause the time-consuming reconstruction of GBF.

To support both insertion and deletion, we propose an updatable data structure, called *Counting Garbled Bloom Filter* (CGBF) as shown in Fig. 2. CGBF is an array and each element consists of a string and a counter. The string represents a share of a revocation status, and the counter indicates the number of revoked certificates that use the share. Formally, a CGBF instance $CGBF$ is parameterized by $(m, n, \lambda, k, \mathcal{H})$. That means $CGBF$ can encode at most n revoked certificates in an m-length array, and every share in the array is a λ-bit string. Every revoked certificate uses k shares that are located by k independent uniform hash functions $\mathcal{H} = \{\mathsf{H}_0, \ldots, \mathsf{H}_{k-1}\}$, where $\mathsf{H}_j : \{0,1\}^* \to \{0, \ldots, m-1\}$ $(0 \le j \le k-1)$.

For convenience, the i-th $(0 \le i \le m-1)$ share/counter in $CGBF$ are denoted by $CGBF[i].share$/$CGBF[i].counter$, respectively. For any revocation status $rs \in \{0,1\}^\lambda$ stored in $CGBF$, we have $rs = \bigoplus_{j=0}^{k-1} CGBF[\mathsf{H}_j(rs)].share$.

B. Block-Oriented Revocation List

In general, the size of CGBF will eventually exceed the capacity of a block. Therefore, we need to design a data structure that stores CGBF as a number of small pieces and supports efficient insertion and deletion. A trivial solution is to divide a CGBF instance by elements. For example, the first piece of $CGBF$ contains the first two elements $(CGBF[0], CGBF[1])$, the second piece contains the third and fourth elements $(CGBF[2], CGBF[3])$, so on and so forth. Unfortunately, this solution is inefficient since revocation status insertion and deletion will cause the frequent modification of all counters, namely rewriting of existing pieces.

To accommodate CGBF in the blockchain architecture, we design a data structure called *Block-Oriented Revocation List* (BORL) as shown in Fig. 2. BORL is a doubly linked list that every node of the list contains the CGBF update (insertion or deletion) over a period of time and can be stored in a transaction on the blockchain. The doubly linked design is to eliminate the forking of CGBF update. Since deletion needs the certificates' expiration time, to avoid traversing the BORL and improve the efficiency, we also design the BORL state which is possessed locally by every blockchain node.

Formally, a BORL instance $BORL$ is parameterized by $(u, d, \mathcal{CH}, (m, n, \lambda, k, \mathcal{H}), \mathsf{H})$. The integers u and d are the upper bounds of updated revocation statuses contained in a transaction for insertion and deletion, respectively. The chameleon hash scheme $\mathcal{CH}$ (Section II-D) is employed to modify the next address stored in the transaction on the blockchain to implement the doubly linked design. The parameter $(m, n, \lambda, k, \mathcal{H})$ describes the CGBF instance stored in $BORL$. We also use a cryptographic hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ to generate the digest of the BORL state and CGBF instance.

A BORL node is a seven-tuple $(type, update, addr_p, addr_n, h_s, h_c, r)$. The boolean $type$ determines whether the update stored in the transaction is for insertion or deletion. The CGBF update is stored in $update$. More specifically, if $type$ is insertion, $update$ consists of $height$ and a number of $((\mathsf{H}_0(rs), CGBF[\mathsf{H}_0(rs)].share), \ldots, (\mathsf{H}_{k-1}(rs), CGBF[\mathsf{H}_{k-1}(rs)].share)), date_{rs})$. The integer $height$ indicates the last block in which the revocation status has been inserted into $CGBF$ and the revocation status of a revoked certificate $rs = \bigoplus_{j=0}^{k-1} CGBF[\mathsf{H}_j(rs)].share$ is expired after the timestamp $date_{rs}$. If $type$ is deletion, $update$ consists of $time$ and a number of $((\mathsf{H}_0(rs), \ldots, \mathsf{H}_{k-1}(rs)), addr_{rs})$. It indicates that certificates expired before timestamp $time$ are deleted from $CGBF$, and the expired revocation status rs is stored in a transaction addressed by $addr_{rs}$ when inserting. The integers $addr_p$/$addr_n$ represent the previous/next BORL nodes, respectively. The string h_s and h_c are the hash value of the BORL state and $CGBF$ before the update is performed, respectively. The integer r is used by $\mathcal{CH}$ to modify $addr_n$.

We define the BORL state of the instance $BORL$ as $BORL.state = (addr_{ins}, addr_{del}, \{date_l, rs_l, addr_{rs_l}\}_{l=0}^{\gamma-1})$. The addresses $addr_{ins}$/$addr_{del}$ indicate the transaction addresses of latest BORL nodes for insertion/deletion, respectively. The array $\{date_l, rs_l, addr_{rs_l}\}_{l=0}^{\gamma-1}$ records the information of all γ revoked certificates stored in $CGBF$, where the revocation status rs_l is expired after the timestamp $date_l$ and $addr_{rs_l}$ is the transaction address where rs_l is inserted.

IV. PROCESS

A. Overview

We aim to design an online service for storing and retrieving the revocation statuses of certificates issued in current PKI systems. To solve the trust concentration problem and the block size challenge, we adopt the blockchain architecture along with the proposed BORL structure. Then, the revocation statuses can be confirmed by and stored on a number of blockchain nodes via traditional consensus. To solve the privacy leakage problem and the latency challenge, we design an efficient privacy-preserving certificate status check protocol for the CGBF structure. Our solution, named PROCESS, consists of four phases: *initialization*, *certificate revocation*, *BORL update*, and *certificate status check*.

Algorithm 1 Revocation

1: **procedure** REVOCATION($certs$, C, $sk_C^{\mathcal{DS}}$)
2: $revo \leftarrow \mathsf{Revoke}(certs)$
3: $sig \leftarrow \mathcal{DS}.\mathsf{Sign}(sk_C^{\mathcal{DS}}, (C, revo))$
4: **return** $((C, revo), sig)$

Algorithm 2 Package

1: **procedure** PACKAGE($pk_C^{\mathcal{DS}}$, $C, revo, sig, CGBF$)
2: **if** $\mathcal{DS}.\mathsf{Verify}(pk_C^{\mathcal{DS}}, ((C, revo), sig)) = 1$ **then**
3: **return** $\mathsf{TransGen}((C, revo), sig)$

B. Initialization

In PKI systems, CAs generate public-private key pairs for authorizing, revoking, and verifying certificates, which are also used in our solution to sign revocation information. For the sake of completeness, those key pairs are generated in the initialization phase. On the other hand, blockchain nodes need to prepare essential materials for CGBF and BORL.

CA. A CA C generates a pair of public-private keys $(pk_C^{\mathcal{DS}}, sk_C^{\mathcal{DS}})$ by $\mathcal{DS}.\mathsf{Gen}(1^\lambda)$, where λ is the security parameter. Then, the CA publishes the public key $pk_C^{\mathcal{DS}}$ and keeps the private key $sk_C^{\mathcal{DS}}$ secret.

Blockchain Node. A blockchain node N generates a pair of public-private keys $(pk_N^{\mathcal{CH}}, sk_N^{\mathcal{CH}})$ by $\mathcal{CH}.\mathsf{Gen}(1^\lambda)$, where λ is also the security parameter. Then, the node deploys a smart contract $\mathsf{Confirmation}(pk_N^{\mathcal{CH}}, \cdot, \cdot)$ on the blockchain, which accepts two parameters (see Section IV-D), and keeps the private key $sk_N^{\mathcal{CH}}$ secret. Finally, the node initializes an empty CGBF instance $CGBF$ with the global parameters $(m, n, \lambda, k, \mathcal{H})$ and the BORL state of the instance $BORL$ parameterized by $(u, d, \mathcal{CH}, (m, n, \lambda, k, \mathcal{H}), \mathsf{H})$. Both $CGBF$ and $BORL.state$ are stored locally on the node.

C. Certificate Revocation

To revoke certificates, a CA generates revocation information (Algorithm 1), and a blockchain node records the information in the blockchain as a transaction (Algorithm 2).

CA. Let $certs = \{cert_0, \ldots, cert_{\gamma-1}\}$ be the set of certificates issued by a CA C that will be revoked, and γ be the number of certificates in $certs$. To revoke those certificates (Line 2), the CA extracts the expiration time (i.e., NotAfter) $date_{cert_l}$ from the certificate $cert_l (0 \le l < \gamma)$ and generates revocation status $rs_{cert_l} \leftarrow \mathsf{H}(cert_l)$, where $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ is the cryptographic hash function defined in $BORL$. The revocation information is $revo = \{(date_{cert_0}, rs_{cert_0}), \ldots, (date_{cert_{\gamma-1}}, rs_{cert_{\gamma-1}})\}$.

Then, the CA signs the revocation information $revo$ along with its identity C with its private key $sk_C^{\mathcal{DS}}$ to obtain a

signature sig (Line 3). Finally, the CA sends $((C, revo), sig)$ to any blockchain node.

Blockchain Node. When a blockchain node receives $((C, revo), sig)$ from the CA C, it verifies that whether the revocation information is valid (Line 2). Then, the node packages $((C, revo), sig)$ into a transaction (Line 3) and broadcasts the transaction in the blockchain network. Finally, the blockchain node returns the address of the transaction to CA for revocation transparency [20] once the transaction is recorded on the blockchain.

D. BORL Update

Transactions of revocation information in the certificate revocation phase are not directly used for certificate status query, because they are inefficient and may leak clients' privacy. Instead, all necessary information is maintained in BORL in this phase, which includes three kinds of operations: *element insertion*, *element deletion*, and *update confirmation*.

1) Element Insertion: To support certificate status query, blockchain nodes collect revocation information from transactions on the blockchain, and insert the revocation information into the CGBF instance by generating a BORL node of insertion type. This operation is shown in Algorithm 3.

According to transaction address $addr_{ins}$ in $BORL.state$, a blockchain node N obtains the $height$ in the BORL node at $addr_{ins}$, and $height$ is the block height where last insertion stops. The blockchain node traverses the transactions on the blockchain bc from $height$-th block, and retrieves the revocation information $revo = \{(date_{cert_0}, rs_{cert_0}), \dots\}$ that contains at most u revocation statuses. It also records the block height $height'$ where the retrieval stops (Line 2).

Before proceeding, the blockchain node takes a snapshot of current BORL state and CGBF instance by computing the hash values of $BOLR.state$ (Line 3)/$CGBF$ (Line 4), respectively. The following steps generate the CGBF update (Line 5-22), and are performed on a temporary duplicate of $BORL.state$ and $CGBF$ rather than modifying them (see the update confirmation operation later in this section). More specifically, each rs_{cert} in $revo$ is mapped into k locations of $CGBF$ (Line 7-20). If some locations are never used, the blockchain node keeps one location for generating a special share (Line 12-13 and 20) and fills others with random shares (Line 15). If there is no unused location (i.e., $empty = -1$ in Line 19), the element insertion fails and $CGBF$ needs to be reconstructed. Then, the CGBF update is $update = (height', list)$.

To complete the BORL node, the blockchain node obtains the transaction address of previous BORL node, which is either $addr_{ins}$ or $addr_{del}$ (Line 23), and computes the random value of chameleon-hash function (Line 25). Finally, the blockchain node packages $(insertion, update, addr_p, \perp, h_s, h_c, r)$ into a transaction (Line 26) and broadcasts the transaction in the blockchain network.

2) Element Deletion: To reduce the frequency of CGBF reconstruction, blockchain nodes delete the expired certificates from the CGBF instance by generating a BORL node of deletion type. This operation is shown in Algorithm 4.

Algorithm 3 Element Insertion

```
1:  procedure INSERTION(bc, BORL.state, CGBF, pk_N^CH)
2:      revo, height' ← GetRS(bc, BORL.state, u)
3:      h_s ← H(BORL.state)
4:      h_c ← H(CGBF)
5:      list ← {}
6:      for each item (date_cert, rs_cert) in revo do
7:          empty ← −1, share ← 0^λ, sset ← {}
8:          for j from 0 to k − 1 do
9:              i ← H_j(rs_cert)
10:             if CGBF[i].counter = 0 then
11:                 if empty = −1 then
12:                     empty ← i
13:                     CGBF[i].share ← 0^λ
14:                 else
15:                     CGBF[i].share ←$ {0,1}^λ
16:             CGBF[i].counter ← CGBF[i].counter + 1
17:             share ← share ⊕ CGBF[i].share
18:             sset ← sset ∪ {(i, CGBF[i].share)}
19:         if empty ≠ −1 then
20:             CGBF[empty].share ← share ⊕ rs_cert
21:         list ← list ∪ {(sset, date_cert)}
22:     update ← (height', list)
23:     addr_p ← GetPrev(addr_ins, addr_del)
24:     ins ← (insertion, update, addr_p, ⊥, h_s, h_c)
25:     r ← CH.Hash(pk_N^CH, ins)
26:     return TransGen(ins, r)
```

According to transaction address $addr_{del}$ in $BORL.state$, a blockchain node N obtains the time that deletion operation has been processed which is recorded as $time$ in the BORL node at $addr_{del}$. The blockchain node extracts an array $array = \{(date_l, rs_l, addr_{rs_l}), \dots\}$ that contains at most d revocation statuses from $BORL.state$. It also records the expired time $time'$ where the deletion stops (Line 2).

Before proceeding, the blockchain node takes a snapshot of current BORL state and CGBF instance as in the element insertion operation (Line 3-4). The following operations generate the CGBF update (Line 5-13) that are also performed on a temporary duplicate of $BORL.state$ and $CGBF$. More specifically, for each item in $array$, the blockchain node computes k locations and reduces the counter at each location (Line 8-11). Then, the CGBF update is $update = (time', list)$.

To complete the BORL node, the blockchain node performs similarly to the last four steps of the element insertion operation, where $(deletion, update, addr_p, \perp, h_s, h_c, r)$ is finally packaged into a transaction and broadcasted in the blockchain network (Line 14-17).

3) Update Confirmation: In element insertion and deletion, all operations are performed on the temporary duplicate of the BORL state and CGBF instance, since the update (i.e., insertion and deletion) packaged in the BORL node is not confirmed in the blockchain network yet. Let the BORL node containing unconfirmed update be $node = (type, update,$

Algorithm 4 Element Deletion

1: **procedure** DELETION($BORL.state, CGBF, pk_N^{\mathcal{CH}}$)
2: $time', array \leftarrow$GetDS($BORL.state, d$)
3: $h_s \leftarrow \mathsf{H}(BORL.state)$
4: $h_c \leftarrow \mathsf{H}(CGBF)$
5: $list \leftarrow \{\}$
6: **for** each item $(date_l, rs_l, addr_{rs_l})$ in $array$ **do**
7: $iset \leftarrow \{\}$
8: **for** j from 0 to $k-1$ **do**
9: $i \leftarrow \mathsf{H}_j(rs_l)$
10: $CGBF[i].counter \leftarrow CGBF[i].counter - 1$
11: $iset \leftarrow iset \cup \{i\}$
12: $list \leftarrow list \cup \{(iset, addr_{rs_l})\}$)
13: $update \leftarrow (time', list)$
14: $addr_p \leftarrow \mathsf{GetPrev}(addr_{ins}, addr_{del})$
15: $del \leftarrow (deletion, update, addr_p, \perp, h_s, h_c)$
16: $r \leftarrow \mathcal{CH}.\mathsf{Hash}(pk_N^{\mathcal{CH}}, del)$
17: **return** $\mathsf{TransGen}(del, r)$

$addr_p, \perp, h_s, h_c, r)$ at the transaction addressed by $addr_n$, and the BORL node at the transaction addressed by $addr_p$ be $node^* = (type^*, update^*, addr_p^*, \perp, h_s^*, h_c^*, r^*)$. To confirm the update, $addr_n$ needs to be recorded in $node^*$. Then, all of the blockchain nodes can apply the update in $node$ to obtain the latest BORL state and CGBF instant.

To record $addr_n$ in $node^*$ (see Algorithm 5), the blockchain node N who generated $node^*$ first verifies whether $node$ is valid (Line 2). This process is similar to Algorithm 3 and 4 except for GetRS and GetDS. Instead of invoking GetRS and GetDS, the blockchain node extracts revocation statuses from the blockchain bc based on the height range $(height, height')$ if $type = insertion$, where $height$ is in the BORL node at $addr_{ins}$ and $height'$ is in $node$, and extracts $array$ from $BORL.state$ based on $(time, time')$ if $type = deletion$, where $time$ is in the BORL node at $addr_{del}$ and $time'$ is in $node$. Then, the blockchain node replaces $\perp$ in $node^*$ with the transaction address $addr_n$, and calculates new randomness r' (Line 3-6). As a result, the BORL node $node^* = (upd^*, r^*)$ is changed to $node' = (upd', r')$, which is broadcasted in the blockchain network.

For all of the blockchain nodes, they need to verify $node'$ before applying the update. To this end, they perform the smart contract $\mathsf{Confirmation}(pk_N^{\mathcal{CH}}, \cdot, \cdot)$ with the parameters $(node^*, node')$ (Algorithm 6). First, it calculates the chameleon hash h of $node^*$ before modification (Line 2), and then verifies whether the modified r' is correct (Line 3). If the smart contract execution result is 1, all of the blockchain nodes replace $node^*$ with $node'$, and update the state of BORL $BORL.state$ and $CGBF$ with $node$. Specifically, they update shares and counters in $CGBF$ and $(addr_{ins}, addr_{del}, \{date, rs, addr_{rs}\})$ according to $node$. Otherwise they ignore $node$ and the modification of $node^*$.

Algorithm 5 Node Modification

1: **procedure** MODIFICATION($bc, BORL.state, CGBF, sk_N^{\mathcal{CH}}, node, addr_n, node^*$)
2: **if** $\mathsf{Verify}(bc, BORL.state, CGBF, node) = 1$ **then**
3: $upd^* \leftarrow (type^*, update^*, addr_p^*, \perp, h_s^*, h_c^*)$
4: $upd' \leftarrow (type^*, update^*, addr_p^*, addr_n, h_s^*, h_c^*)$
5: $h \leftarrow \mathcal{CH}.\mathsf{ReHash}(pk_N^{\mathcal{CH}}, upd^*, r^*)$
6: $r' \leftarrow \mathcal{CH}.\mathsf{Adapt}(sk_N^{\mathcal{CH}}, upd^*, upd', r^*, h)$
7: **return** (upd', r')

Algorithm 6 Update Confirmation

1: **procedure** CONFIRMATION($pk_N^{\mathcal{CH}}, node^*, node'$)
2: $h \leftarrow \mathcal{CH}.\mathsf{ReHash}(pk_N^{\mathcal{CH}}, upd^*, r^*)$
3: **return** $\mathcal{CH}.\mathsf{Check}(pk_{CH}, upd', r', h)$

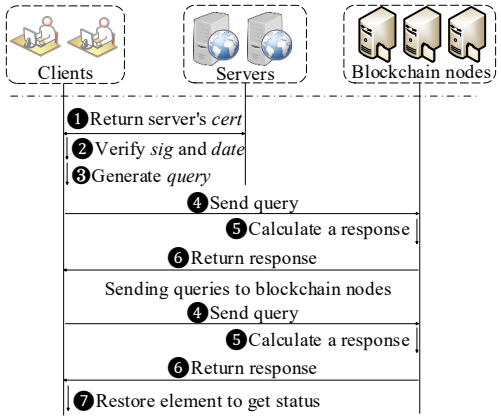

Fig. 3. Certificate status check protocol

E. Certificate Status Check

Before a client establishes the secure connection with a server, the client checks whether the certificate provided by the server is valid, as shown in Fig. 3.

When a client accesses a server, the server provides the client with its certificate $cert$ (Step 1). The client verifies the signature and expired time of $cert$ (Step 2). If the signature is valid and $cert$ is not expired, the client invokes the revocation status query protocol (Step 3-7).

In the revocation status query protocol, the client first generates the query of $cert$ (Step 3). According to the privacy and latency requirements, the client sets the length of index range len when querying a share and the number of blockchain nodes num when answering for each share, and performs Algorithm 7. For each index i, the client chooses len successive indexes that contain i (Line 7-8) and computes for each blockchain node the shares it should retrieve (Line 9-13). Specifically, the query is $\{(i_j - pos_j), len, \{pad_{j,l}\}_{l=0}^{num-1}\}_{j=0}^{k-1}$ and each $((i_j - pos_j), len, pad_{j,l})$ is sent to a random (better not repetitive) blockchain node (Step 4).

When a blockchain node receives $((i_j - pos_j), len, pad_{j,l})$, it computes $res_{j,l} = \bigoplus_{i=i_j-pos_j, pad_{j,l}[i]=1}^{i_j-pos_j+len-1} CGBF[i].share$ (Step 5) and sends $res_{j,l}$ back to the client (Step 6).

Algorithm 7 Query Generation

procedure QUERYGEN($m, \lambda, k, \mathcal{H}, \mathsf{H}, cert, len, num$)
  $rs_{cert} \leftarrow \mathsf{H}(cert)$
  $query \leftarrow \{\}$
  for $j = 0$ to $k-1$ **do**
    $i \leftarrow \mathsf{H}_j(rs_{cert})$
    $target \leftarrow 0^{len}, pad \leftarrow 0^{len}, strings \leftarrow \{\}$
    $pos \xleftarrow{\$} \{0, \ldots, len-1\}$
    $target[pos] \leftarrow 1$
    for l from 0 to $num-2$ **do**
      $pad \xleftarrow{\$} \{0,1\}^{len}$
      $strings \leftarrow strings \cup \{pad\}$
      $target \leftarrow target \oplus pad$
    $strings \leftarrow strings \cup target$
    $query \leftarrow query \cup \{((i-pos), len, strings)\}$
  return $query$

After receiving the responses $\{res_{j,l}\}_{l=0}^{num-1}$ for one index, the client can restore $CGBF[\mathsf{H}_j(rs_{cert})].share = \bigoplus_{l=0}^{num-1} res_{j,l}$. Then, the client combines the k shares by $rs'_{cert} = \bigoplus_{j=0}^{k-1} CGBF[\mathsf{H}_j(rs_{cert})].share$ and compares it with rs_{cert} obtained in Algorithm 7 (Step 7). If rs'_{cert} is equal to rs_{cert}, the certificate $cert$ is revoked, otherwise the certificate is valid.

V. Security Analysis

We show that PROCESS solves the problems in Section I (i.e., trust concentration and privacy leakage) through the following theorems. Especially, Theorem 1 claims that the certificate status cannot be compromised and Theorem 2 states that the blockchain node cannot learn the certificate queried by the client.

Theorem 1: Under the threat model in section II-B, CAs and blockchain nodes can ensure the integrity and authenticity of revocation information and BORL, respectively.

Proof: (sketch) This theorem can be directly derived through the distributed storage feature of the blockchain technology. Note that the revocation information and BORL nodes are packaged into transactions and verified by every blockchain node before being confirmed on the blockchain.

From CAs' perspective, the adversary may tamper or forge the revocation information. However, revocation information are signed by CAs' private keys and all blockchain nodes can verify the signatures stored on blockchain. Even if CAs' private keys are leaked, they can discover the invalid revocation information through the transaction addresses returned by blockchain nodes. Therefore, the integrity and authenticity of revocation information is guaranteed.

From blockchain nodes' perspective, the adversary may update BORL incorrectly. However, all BORL nodes, including insertion and deletion types, are verified in the update confirmation operation. More specifically, a BORL node is first verified by the blockchain node who generated the preceding BORL node and then this verification is verified by all other blockchain nodes. Therefore, the integrity and authenticity of BORL is guaranteed. ■

Theorem 2: If $\mathcal{H}$ are modeled as random oracles and there is at least one honest blockchain node, then the probability of any adversary that correctly determines the server accessed by the client in the certificate status check protocol is not greater than $m/(N \cdot len)$, where m is the length of CGBF, N is the number of total certificates, and len is the length of index range. More specifically, let P be the probability of insertion failures in CGBF and n be the upper bound of revoked certificates, then the successful probability of any adversary is less than $n \ln P/(-0.48 \cdot N \cdot len)$.

Proof: (sketch) In Algorithm 7, each position mapped from H_j is hidden in num strings of length len, where $num \geq 2$ is the number of blockchain nodes that response for one position. Since there is at least one honest blockchain node, even the collusion of $num-1$ blockchain nodes cannot distinguish the position from the len bits.

To compute the number of certificates that can be mapped to any of the len bits, we first calculate the probability that a certificate is mapped into the string of length len, which is $p = 1-(1-\frac{len}{m})^k$ since $\mathcal{H}$ are modeled as random oracles. Then, the expectation of the number of certificates mapped to the string is Np. As a result, we have the successful probability of any adversary $1/(Np) = 1/(N \cdot (1-(1-\frac{len}{m})^k)) \leq 1/(N \cdot \frac{len}{m}) = m/(N \cdot len)$ since $k \geq 1$.

Given P and n in advance, we can determine $m \approx -n \ln P/(\ln 2)^2$ [21], where $k = \ln 2 \frac{m}{n}$ minimizes the possibility of insertion failures $P \approx \left(1-e^{-kn/m}\right)^k$ in CGBF. Therefore, we have the successful probability of any adversary $m/(N \cdot len) = n \ln P/(-(\ln 2)^2 \cdot N \cdot len) < n \ln P/(-0.48 \cdot N \cdot len)$. ■

According to Liu et al. [16], 8% of all valid certificates are revoked, and the successful probability in Theorem 2 is then $-1.67 \ln P/len$. By adjusting len, the client can control the privacy protection level.

VI. Implementation and Evaluation

In this section, we conduct experiments and compare PROCESS with CertChain [8] to evaluate the performance of our solution in terms of space cost, privacy protection, and processing time (Section VI-B).

A. Implementation

Our prototype is mainly written in Javascript (node.js) and Python (version 3.6.9). CAs are implemented through OpenSSL and blockchain is based on Ethereum [22]. Clients and CAs are deployed on Intel Core i5-7400U CPU @3.0GHz, 8G RAM, and Ubuntu 18.04 64bit operating system. Blockchain nodes are deployed on Intel Xeon E5-2680 v4 CPU @2.4GHz, 32GB DDR, and Ubuntu 18.04 64bit operating system. Our dataset is from CRLite [23], which provides some basic information (e.g., domains, issuer's name, fingerprint (160 bits) used to mark each certificate) of 10.7M valid certificates, and we use the fingerprints of these certificates to conduct experiments.

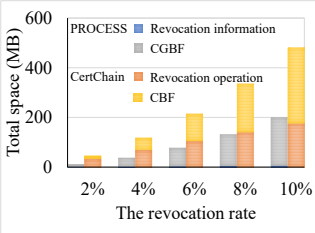

Fig. 4. Space cost in different revocation rates

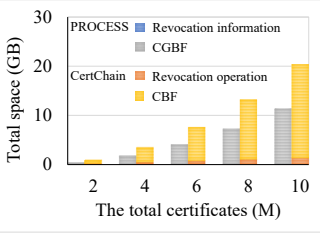

Fig. 5. Space cost in different numbers of total certificates

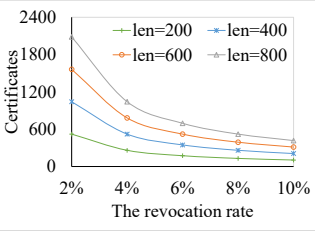

Fig. 6. Privacy protection in different revocation rates and query ranges

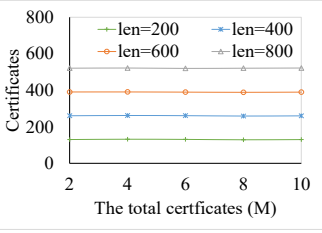

Fig. 7. Privacy protection in different total certificates and query ranges

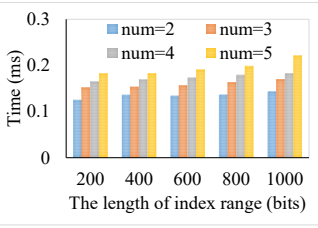

Fig. 8. Time of generating the query on the client side in different query ranges and the number of blockchain nodes

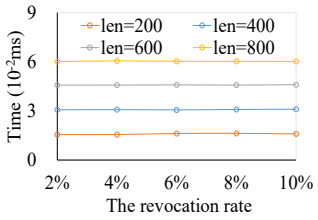

Fig. 9. Time of calculating the response on the blockchain node side in different revocation rates and query ranges

B. Result Analysis

For the parameters $(m, n, \lambda, k, \mathcal{H})$ of CGBF, we fixed $\lambda = 160$, $k = 3$, and $\mathcal{H} = \{hashlib.sha1(), hashlib.sha384(), hashlib.sha512()\}$. We also set the possibility of insertion failures $P = (1 - e^{-kn/m})^k = 0.0001$. Then, we adjust the revocation rate and the number of total certificates to determine various n and corresponding m, where n is product of the revocation rate and the number of total certificates.

1) Space cost: We conduct experiments to study the space cost of revocation statuses in CertChain and PROCESS, where the former consists of the revocation operations and CBF, and the latter consists of the revocation information and CGBF.

First, we randomly select 1M certificates [8] and adjust the revocation rate. To study the space cost in the blockchain, we revoke 500 certificates once a time, where 500 is the upper bound of the revocation operations in a block in CertChain. In addition, we set the number of reconstructions of CGBF with the product of the possibility of insertion failures and the number of revoked certificates. As shown in Fig. 4, CertChain consumes more space than PROCESS. On one hand, the revocation operations in CertChain take up much more space than the revocation information in PROCESS, since PROCESS supports batch processing and only contains necessary information for updating global statuses. On the other hand, although a single CBF is much smaller than CGBF, the entire CBF needs to be stored in every block while only a few CGBFs are stored on the blockchain. More specifically, the number of CGBFs stored on the blockchain is equal to the number of reconstructions, which is rare in practice.

Second, we set the revocation rate as 8% [16] and adjust the number of total certificates. As shown in Fig. 5, the space cost increases as the number of total certificates increases, but the cost of CertChain grows much faster than PROCESS. Moreover, the space costs of CBF and CGBF take up most of the space in CertChain and PROCESS, respectively.

In conclusion, the way that PROCESS updates global revocation statuses is more practical than CertChain in the blockchain architecture in terms of the space cost. In addition, the possibility of false positives in CBF is 2^{λ} times to the one in CGBF, which means PROCESS has a higher accuracy in checking the status of a certificate.

2) Privacy protection: Since CertChain does not protect clients' privacy, we only study the factors that influence the privacy protection in PROCESS.

First, we set the number of total certificates as 1M and adjust the revocation rate and the length of index range len. As shown in Fig. 6, the number of certificates mapped into the range of a query decreases with the increase of the revocation rate, which means the privacy protection level degrades as the revocation rate grows. That is because the increase of the revocation rate leads to the increase of m (i.e., the length of CGBF), which means the probability len/m that certificates are mapped into the index range decreases.

Second, we set the revocation rate as 8% and adjust the number of total certificates and the length of index range len. As shown in Fig. 7, the number of certificates mapped into the range of a query remains stable when the number of total certificates increases, which means that PROCESS can accommodate many CAs without affecting the privacy protection level.

Moreover, from Fig. 6 and 7, the number of certificates mapped into the range of a query decreases with the decrease of len. Therefore, the client should set a larger len if it needs better privacy protection.

3) Processing time: We set the number of total certificates as 1M to study the processing time in the on-chain part of the certificate status check protocol, which includes generating a query and calculating a response. The processing time of restoring an element in the on-chain part is omitted since it is almost the same as generating the query. In addition, we evaluate the processing time of the complete certificate status check protocol in PROCESS and CertChain.

First, we set the revocation rate as 8% and adjust the length of index range len and the number of blockchain nodes num to perform Algorithm 7 on the client side. As shown in Fig. 8, the processing time of generating a query increases with the increase of len and num, which corresponds to the time

TABLE I
PROCESSING TIME (MILLISECONDS) OF EACH STEP FOR CHECKING A CERTIFICATE'S STATUS

	Step 1 *Signature*	**Step 2** *Expiration*	**Step 3** *On-chain Check*	*Total*
CertChain [8]	9.39	0.60	9.05	19.04
PROCESS	9.39	0.60	3.75	13.74

complexity of Algorithm 7 (i.e., $O(len \cdot num)$). The client can appropriately reduce num to decrease the processing time.

Second, we adjust the revocation rate and the length of index range len to study the time of calculating a response on the blockchain side. As shown in Fig. 9, the processing time only increases with the increase of len, since the response is calculated from len shares in CGBF and is independent of the number of total/revoked certificates. Therefore, the client can choose appropriate len to balance the privacy protection and response latency requirements.

Finally, we compare PROCESS with CertChain, as shown in Table I. Similar to PROCESS, CertChain takes three steps to check a certificate's status: (1) verify the signature; (2) check the expiration date; (3) run an on-chain check. The first two steps are the same in both solutions. In step 3, PROCESS consumes less time than CertChain because the latter needs to perform more accesses to the blockchain, such as locating and retrieving the revocation operation. In conclusion, PROCESS is more efficient than CertChain in terms of the processing time of certificate status check.

VII. Related Work

According to whether the revocation information is released in time, the revocation mechanism can be divided into two categories: *periodic push* and *online query*.

A. Periodic Push Mechanism

In periodic push mechanism, the CA periodically pushes revocation information to the client.

CRL [2] puts the serial number of the revoked certificate in the list. The query efficiency of CRL is low, and it occupies a lot of space as the number of revoked certificates increases. Existing solutions are dedicated to reduce bandwidth consumption. On one hand, they reduce the coverage of the list. Google's CRLSet [24] pre-selects a subset of all revoked certificates, and Firefox's OneCRL [25] pushes lists of revoked intermediate certificates to browsers. On the other hand, they design efficient storage structures. Rabieh et al. [26] adopted Bloom filter, Larisch et al. [23] proposed *filter cascade*, and Smith et al. [27] used *bit vector*. Although these solutions can reduce the bandwidth consumption, it cannot fundamentally solve the latency of update.

B. Online Query Mechanism

In online query mechanism, a client runs an on-demand query for a single certificate's status to a responder, which returns the status to that client.

In *OCSP* [4], when a client attempts to access a server, it sends a request for certificate status to a responder, which provides a timestamp and signed revocation status of the server for the client. However, the client will expose its access behaviors to responders.

OCSP Staple [11] and *OCSP Must Staple* [12] were proposed to alleviate the privacy concerns. It is the server (e.g., a website) rather than the client to download an OCSP response from a staple responder. However, it requires an X.509 extension to notify the client that the server will send an OCSP response to the client. Moreover, they put too much pressures on staple responders, which sign revocation information for certificates every few seconds. Chariton et al. [28] proposed *CCSP*, the responder packages multiple revocations into a response, and compresses it with bitmap and other compression algorithms. However, it still requires an X.509 extension, and these solutions are based on centralized model.

To avoid single-point failure in centralized model, Chen et al. proposed *CertChain* [8] and *PBCert* [29], which are based on blockchain. CAs store the operations of certificates and double counting Bloom filters on blockchain to check certificates' statuses. In CertChain [8], a client sends a certificate's hash to a responder for checking the certificate's status, which will leak the client's accessed behavior. In PBCert [29], a client omits the first few bits of the certificate's hash to obtain multiple certificates' statuses. PBCert has a high possibility of false positives, which may misjudge an invalid certificate as valid. Kubilay et al. [30] proposed *CertLedger* to store the MHT composed of the certificate status at the head of each block. However, the construction and verification of the MHT cause high latency as certificates increase.

In summary, the existing solutions either have a long update circle, or they cannot solve the problems of latency and privacy concerns meanwhile.

VIII. Conclusion

To solve the problems of trust concentration and privacy leakage in the OCSP mechanism, we propose a privacy-preserving on-chain certificate status service, called PROCESS. Our solution is based on dedicated data structures, CGBF and BORL, and a certificate status check protocol, which is flexible in terms of privacy protection and latency requirements. The security analysis and experimental results show that PROCESS provides efficient query services for clients under the premise of privacy protection.

IX. Acknowledgments

This research was supported in part by the National Natural Science Foundation ofChina under grants No. 61772383, 61702379, U1836202; bythe China Postdoctoral Science Foundation under grant No. 2019T120685; by the Joint Fund of Ministry of Education of China for Equipment Pre-research under grant No. 6141A02033341; by the HUAWEI TECHNOLOGIES CO., LTD. The corresponding authors are Kun He and Jing Chen.

REFERENCES

[1] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," RFC Editor, Tech. Rep. 8446, 2018.

[2] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. T. Polk *et al.*, "Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile." RFC Editor, Tech. Rep. 5280, 2008.

[3] S. Han, H. Kwon, C. Hahn, D. Koo, and J. Hur, "A survey on MITM and its countermeasures in the TLS handshake protocol," in *Proceedings of ICUFN*, 2016.

[4] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "X.509 internet public key infrastructure online certificate status protocol - OCSP," RFC Editor, Tech. Rep. 6960, 2013.

[5] J. Chen, S. Yao, Q. Yuan, R. Du, and G. Xue, "Checks and balances: A tripartite public key infrastructure for secure web-based connections," in *Proceedings of INFOCOM*, 2017.

[6] M. T. Hammi, P. Bellot, and A. Serhrouchni, "BCTrust: A decentralized authentication blockchain-based mechanism," in *Proceedings of WCNC*, 2018.

[7] Y. Xu, S. Zhao, L. Kong, Y. Zheng, S. Zhang, and Q. Li, "ECBC: A high performance educational certificate blockchain with efficient query," in *Proceedings of ICTAC*, 2017.

[8] J. Chen, S. Yao, Q. Yuan, K. He, S. Ji, and R. Du, "CertChain: Public and efficient certificate audit based on blockchain for TLS connections," in *Proceedings of INFOCOM*, 2018.

[9] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-López, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, S. D. Schoen, and B. Warren, "Let's Encrypt: An automated certificate authority to encrypt the entire web," in *Proceedings of CCS*, 2019.

[10] C. Brunner, F. Knirsch, A. Unterweger, and D. Engel, "A comparison of blockchain-based PKI implementations," in *Proceedings of ICISSP*, 2020.

[11] D. Eastlake *et al.*, "Transport layer security (TLS) extensions: Extension definitions," RFC Editor, Tech. Rep. 6066, 2011.

[12] P. Hallam-Baker, "X.509v3 transport layer security (TLS) feature extension," RFC Editor, Tech. Rep. 7633, 2015.

[13] T. Chung, J. Lok, B. Chandrasekaran, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, J. P. Rula, N. Sullivan, and C. Wilson, "Is the web ready for OCSP must-staple?" in *Proceedings of IMC*, 2018.

[14] N. Ahituv, Y. Lapid, and S. Neumann, "Protecting statistical databases against retrieval of private information," *Computers & Security*, 1988.

[15] K. He, J. Chen, Q. Zhou, R. Du, and Y. Xiang, "Secure dynamic searchable symmetric encryption with constant client storage cost," *IEEE Transactions on Information Forensics and Security*, 2021.

[16] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. Maggs, A. Mislove, A. Schulman, and C. Wilson, "An end-to-end measurement of certificate revocation in the web's PKI," in *Proceedings of IMC*, 2015.

[17] D. Derler, K. Samelin, D. Slamanig, and C. Striecks, "Fine-grained and controlled rewriting in blockchains: Chameleon-hashing gone attribute-based," in *Proceedings of NDSS*, 2019.

[18] J. Camenisch, D. Derler, S. Krenn, H. C. Pöhls, K. Samelin, and D. Slamanig, "Chameleon-hashes with ephemeral trapdoors - and applications to invisible sanitizable signatures," in *Proceedings of PKC*, 2017.

[19] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: an efficient and scalable protocol," in *Proceedings of CCS*, 2013.

[20] Z. Wang, J. Lin, Q. Cai, Q. Wang, D. Zha, and J. Jing, "Blockchain-based certificate transparency and revocation transparency," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[21] G. . Kayaturan and A. Vernitski, "Routing in hexagonal computer networks: How to present paths by Bloom filters without false positives," in *Proceedings of CEEC*, 2016.

[22] W. Ethereum, "A secure decentralised generalised transaction ledger," Ethereum project, Tech. Rep., 2014.

[23] J. Larisch, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "CRLite: A scalable system for pushing all TLS revocations to all browsers," in *Proceedings of SP*, 2017.

[24] A. Langley, "Revocation checking and chromes CRL," 2012. [Online]. Available: https://www.imperialviolet.org/2012/02/05/crlsets.html

[25] M. Goodwin, "Revoking intermediate certificates: Introducing OneCRL," 2015. [Online]. Available: https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/

[26] K. Rabieh, M. M. E. A. Mahmoud, K. Akkaya, and S. Tonyali, "Scalable certificate revocation schemes for smart grid AMI networks using Bloom filters," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[27] T. Smith, L. Dickinson, and K. Seamons, "Lets Revoke: Scalable global certificate revocation," in *Proceedings of NDSS*, 2020.

[28] A. A. Chariton, E. Degkleri, P. Papadopoulos, P. Ilia, and E. P. Markatos, "CCSP: A compressed certificate status protocol," in *Proceedings of INFOCOM*, 2017.

[29] S. Yao, J. Chen, K. He, R. Du, T. Zhu, and X. Chen, "PBCert: Privacy-preserving blockchain-based certificate status validation toward mass storage management," *IEEE Access*, 2019.

[30] M. Y. Kubilay, M. S. Kiraz, and H. A. Mantar, "CertLedger: A new PKI model with certificate transparency based on blockchain," *Computers & Security*, 2019.