Impossibility of Finding Any Third Family of Server Protocols Integrating Byzantine Quorum Systems with Threshold Signature Schemes*

Jingqiang Lin[1,2], Peng Liu[2], Jiwu Jing[1], and Qiongxiao Wang[1]

[1] The State Key Laboratory of Information Security, Graduate University of CAS, Beijing 100049, China
[2] The Pennsylvania State University, University Park, PA 16802, USA

Abstract. In order to tolerate servers' Byzantine failures, a distributed storage service of self-verifying data (e.g., certificates) needs to make three security properties be Byzantine fault tolerant (BFT): data consistency, data availability, and confidentiality of the (signing service's) private key. Building such systems demands the integration of Byzantine quorum systems (BQS), which only make data consistency and availability be BFT, and threshold signature schemes (TSS), which only make confidentiality of the private key be BFT. Two families of correct or *valid* TSS-BQS systems (of which the server protocols carry all the design options) have been proposed in the literature. Motivated by the failures in finding a third family of valid server protocols, we study the reverse problem and formally prove that it is *impossible* to find any third family of valid TSS-BQS systems. To obtain this proof, we develop a *validity theory* on server protocols of TSS-BQS systems. It is shown that the only two families of valid server protocols, "predicted" (or deduced) by the validity theory, precisely match the existing protocols.

Keywords: Byzantine fault tolerance, Byzantine quorum systems, threshold signature schemes.

1 Introduction

Malicious codes, software bugs or operator mistakes can cause servers' Byzantine (or arbitrary) failures [12], and then compromise the services of network systems. As a result, BFT (Byzantine fault tolerant) systems, which run correctly in the presence of failures and do *not* have any assumptions about the behavior of faulty entities, are increasingly important. Several techniques [5,6,8,12,13] are proposed to provide BFT properties, such as integrity, consensus, consistency, availability and confidentiality. Of these techniques, BQS (Byzantine quorum systems) [13] and TSS (threshold signature schemes) [6] are remarkable.

* Jingqiang Lin, Jiwu Jing and Qiongxiao Wang were supported by National Natural Science Foundation of China grant 70890084/G021102 and National Science & Technology Pillar Program of China grant 2008BAH22B01. Peng Liu was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-0916469 and AFRL FA8750-08-C-0137.

S. Jajodia and J. Zhou (Eds.): SecureComm 2010, LNICST 50, pp. 307–325, 2010.

BQS provide distributed storage services by replicating data on multiple servers, despite the Byzantine failures of (a certain number of) servers. To ensure data consistency and data availability, each read/write operation is performed on some quorum of servers (every quorum is a subset of servers; any two quorums have at least one server in common). In particular, in order to tolerate up to f faulty servers, $3f + 1$ servers are needed to compose a *dissemination* BQS storing *self-verifying* data, and every quorum contains $2f + 1$ servers (i.e., each operation is performed on at least $2f + 1$ servers; the intersection of quorums masks the impact of faulty servers and enables clients to obtain the right replica).

TSS are also proposed to tolerate servers' Byzantine failures by distributing a private key among n servers. Each server holds a share (or partition) of the private key. A threshold number (denoted h, $1 < h \leq n$) of servers can cooperatively use the distributed private key to sign messages, while any subset of fewer than h servers cannot. Every server *partially* signs a message (i.e., uses its key share to generate a partial signature), and h partial signatures can be combined into a fully signed message.

Essentially, *a*) the goal of BQS is to make data consistency and data availability be BFT; and *b*) TSS can be viewed as a measure to make confidentiality of the (signing service's) private key be BFT. This key observation provides an intuitive understanding about the merits of *integrating* BQS with TSS, two BFT techniques holding different security properties. In a nutshell, traditional BQS [13,14] make two properties (i.e., data consistency and availability) be BFT, while the integration of BQS and TSS yields three BFT properties (i.e., confidentiality of the private key, data consistency and availability). More specifically, this integration benefits the services that demand Byzantine fault tolerance on all of the three properties instead of two.

Given these nice properties of TSS-BQS systems, people have been trying to design TSS-BQS systems. COCA [26] is a TSS-BQS system for BFT certificate query/update services, and its protocol is adopted in CODEX [15] to store secret data. [10] proposed another server protocol. Their study shows that when BQS (e.g., public key infrastructures or publish/subscribe systems) need to support write operations (e.g., create or update) on self-verifying data, the third property is much useful. From the user point of view, letting confidentiality of the signing service's private key BFT ensures non-repudiation of service signatures; so clients can have full trust in the service. Besides, proactive recovery, not implemented in traditional BQS, is enabled by the integration with TSS [15,26]: the service's private key keeps unchanged while servers are recovered periodically.

This integration provides more assurance than the "sum" of BQS and TSS. In particular, it is recognized that an *integrated* storage service in this manner shall and can ensure two "upgraded" security properties:

Service Availability. In the presence of Byzantine failures, a read/write request from authorized clients still gets a response to it, which is signed using the distributed service private key.

Service Integrity. Each fully signed response guarantees that the requested read/write operation has been performed on some quorum of servers.

These two integrated properties are beyond both BQS and TSS, and cannot be satisfied automatically even when confidentiality of the service private key, data consistency and availability are all ensured. Compared to traditional BQS [13,14], clients can directly update self-verifying data without the assistance of external signing services. Compared to TSS, the implication of a signed response is beyond asserting that the request has been processed by the signing service itself; it also asserts that the requested operation has been performed on a quorum of servers and that Byzantine failures won't affect the correctness.

At first glance, there appear to be lots of options in designing a *valid* TSS-BQS system. For example, *a*) the threshold to sign might be any number between 2 and $3f + 1$; and *b*) integrated with TSS, BQS might construct a response when less than $2f + 1$ servers are examined to have performed the requested operation. Moreover, the comparison of existing TSS-BQS systems [10,26] in terms of communication costs, computation costs and the ability to handle concurrent operations (see Section 6) shows that these options can affect performance of TSS-BQS systems, and suggests that there might be different optimized options or tradeoffs when TSS-BQS systems are applied for specific applications.

However, *only two* valid server protocols [10,26] are proposed in the literature (they have roughly the same client protocol). We had tried to design a different or better one, but the outcome is always similar to [10,26]. The failures suggest that there may *not* exist *any third* family of valid server protocols! This suggestion is hard to believe, but our research shows that this conjecture should be true.

Our main contribution is a *formalization* of this impossibility conjecture and a *proof* asserting that it is true. In particular, we propose a validity theory on server protocols of TSS-BQS systems. To the best of our knowledge, it is the first validity theory on this problem. Using this theory, we prove that there are *only two* families of valid server protocols of TSS-BQS systems. The representatives of these two families are that of COCA [26] (denoted *SP-I* in this paper) and that of [10] (denoted *SP-II*), respectively. The validity theory also shows that SP-I and SP-II are the only two *efficient* (and valid) protocols. Our conclusion advises researchers to *a*) apply TSS-BQS systems through one of these two protocols, and *b*) improve TSS-BQS systems by more efficient TSS, task schedulers or resource management of servers, but *not* by server protocol designs.

The rest of this paper is organized as follows. Section 2 describes the TSS-BQS system model. The problem of server protocol design is formulated in Section 3, followed by the main results of the validity theory in Section 4. In Section 5, we deduce the two existing server protocols based on the proposed theory. Performance is analyzed in Section 6, and related work is presented in Section 7. We conclude in Section 8.

2 System Model

A TSS-BQS system consists of n *servers*, and an arbitrary number of *clients* that are distinct from the servers. Servers can be *correct* or *faulty*. A correct

server always follows its protocol, while a faulty one can *arbitrarily* deviate from its protocol (i.e., Byzantine failure). Assume up to f servers can be faulty throughout this paper. We assume that clients always behave correctly.

Data Replication. TSS-BQS systems are demanded by the security requirements arising in maintaining *self-verifying* data (e.g., certificates), whose authenticity (or origin) and integrity can be verified by any entity (server or client). Before being replicated on servers of a TSS-BQS system, each data item is signed by the system to make it self-verifying; thus any modification (by faulty servers or other attackers) can be detected by any entity.

To tolerate servers' Byzantine failures, data are replicated on multiple servers, which can be regarded as variables supporting read/write operations. For a variable x, each server (denoted S_i, $1 \leq i \leq n$) independently stores its replica (consisting of a value v and a timestamp t, which are signed *together* as one replica), denoted $[x, v_i, t_i]$. Timestamps are assigned by a client when it writes the variable, and each client c has its own timestamp set TS_c, not intersecting other $TS_{c'}$ for any other client c' [13]. For example, timestamps can be formed as ascending sequence numbers appended with the name of clients [14].

Quorum. In order to tolerate up to f faulty servers, a dissemination BQS is composed of $n = 3f + 1$ servers, and each quorum contains $2f + 1$ servers [13]. Then, every pair of quorums intersect on at least $f + 1$ servers. This "pervasive intersection" feature enables BQS to achieve data consistency; i.e., a read operation returns the *right* replica, which is written by the most recent write operation. To leverage this feature, a new data item $[x, v, t]$ must be delivered to some quorum of $2f+1$ servers before the write operation ends. The right replica of variable x is obtained out of the (different) replicas stored on $2f + 1$ servers, by choosing the unmodified one with the highest timestamp [13].

Service Key. A TSS-BQS system holds one system-wide key pair, the *service private key* and the *service public key.* The service private key is split into *service key shares* based on TSS, and distributed among the *same* $3f + 1$ servers composing the dissemination BQS. Any h $(1 < h \leq n)$ servers can use their service key shares to sign messages cooperatively. Conspiracies by fewer than h servers cannot compromise the service private key or use it to sign any messages.

The service public key is known to every entity, and clients accept responses and replicas only if they are verifiable using the service public key. That is, servers can use the service private key to sign *a*) a response to clients, and *b*) each data item stored in TSS-BQS systems.

Server Key. To prevent outside attackers from impersonating servers of TSS-BQS systems and for secure communications among servers, each server also holds a key pair denoted *server key*, which has *nothing* to do with the service key pair. A server knows others' public keys, and server keys are used to sign and verify messages *only* among servers.

Clients only (need to) know the service public key but *not* any server keys. As a further benefit, they aren't disturbed by proactive recovery [8,27] (i.e., periodic

refreshment of service key shares and server keys) against mobile adversaries [19] (which attack and compromise one server for a limited period of time before moving on to another), because the service key pair keeps unchanged while service key shares are refreshed.

Uniformity. All servers are uniform. Given a TSS-BQS system, all (correct) servers follow one same protocol. If a message is delivered to two correct servers, they process it following the same protocol. Servers are *not* assumed to run identically. Each server runs independently according to its own state and messages received, and then they may run to *different* branches of the same protocol. However, there is no sentence with any special servers' identities in the protocol. Note that this uniformity assumption has *not* any constraints on faulty servers, which always run in an arbitrary way.

This assumption is generalized from usual threshold cryptographic schemes and quorum systems. Assuming that all servers are uniform, the threshold to sign and the size of a quorum can depict the condition to fully sign a messages by TSS and to finish a read/write operation in BQS, respectively.

Asynchronous Fair Link. A fair link [15,26] is a channel that doesn't deliver all messages sent, but if an entity sends infinitely many messages to another entity then infinitely many of these messages are correctly delivered. In addition, the link is asynchronous; i.e., there is no bound on message delivery delay or server execution speed.

We assume that only asynchronous fair links are provided among all entities. Adversaries may eavesdrop, delay, delete or alter messages in transmit, and replay or insert messages. However, a message sent sufficiently often by an entity to another will be delivered eventually.

3 Problem Formulation

The problem is to formally prove that it is impossible to design any third family of valid server protocols. To make this problem tangible, we need to define the notion of "validity" and firstly model the activities of clients and servers.

3.1 Client Protocol

When reading/writing variable x, an authorized client of TSS-BQS systems periodically sends a request to at least $f+1$ servers until it receives a signed response verifiable using the service public key. Because up to f servers could be faulty, sending a request to $f+1$ servers guarantees that at least one correct server receives it eventually, and starts the server protocol.

A credential is generated and included in the read/write request, authorizing this operation. To prevent attackers from replaying the past signed responses, a nonce (e.g., the name of the client and an ascending sequence number) is included in each request and also the corresponding response. For reading, only the right replica is returned in each fully signed response. For writing, a client

firstly reads the variable to obtain its current timestamp t', and chooses a higher timestamp $t > t'$ [13]. Then, the new value v and the timestamp t compose the write request, and its response is a signed acknowledgement to it.

3.2 General Model of Server Protocols

We present a general model of server protocols to enable the design space identification for TSS-BQS systems, showing both the flexible and the fixed parts. In our model, every server is abstracted to implement three functions as follows.

Storage. As a server of BQS firstly, S_i maintains its replica of variable x independently. On receiving a read request, S_i replies with its replica $[x, v_i, t_i]$. On receiving a request writing $[x, v, t]$, S_i acknowledges it, and only updates its replica if the data item being written is unmodified (i.e., verifiable using the service public key) and has a higher timestamp than its own (i.e., $t > t_i$).

Delegate. Since clients only know the service public key, some servers shall become *delegates* for each request, performing the requested read/write operation among servers on behalf of the client sending the request [26]. Note that a delegate is *not* a special or additional server; otherwise it will be a vulnerable component not tolerating failures. On receiving a request from clients, each (correct) server becomes a delegate for it. Since each request is sent to $f + 1$ servers, there may be more than one delegate for each operation. These delegates will return same responses, unless there are concurrent read/write operations.

On receiving a request from clients, a delegate constructs a response, cooperates with h servers to sign it, and then sends this signed response to clients. The response (to be signed) is constructed as below: *a*) in the case of read operations, the delegate lets the response be the right replica determined (or chosen) out of the $2f + 1$ replicas read from some quorum of servers; *b*) in the case of write operations, the delegate firstly cooperates with h servers to sign the data item being written to make it self-verifying, writes it to $2f + 1$ servers, and lets the response be an acknowledgement to this write request.

Partial-Signing. The service key pair is used to communicate with clients and create self-verifying data. So, servers can generate partial signatures for *a*) a response to clients, or *b*) a data item being written (to make it self-verifying).

S_i can use its service key share to partially sign a response, when receiving a partial-signing request for it. However, since the delegate sending this request may be faulty, the response (to be signed) may: *a*) be constructed when the requested read/write operation has not been performed on enough servers; sometimes, the corresponding request doesn't even exist; or *b*) return an out-of-date but self-verifying replica, even though the faulty delegate has read $2f + 1$ replicas. To avoid signing such a fake response, before partially signing each response, a (correct) server S_i shall process the corresponding read/write request by itself, and/or carry out some examinations. Messages that are signed using server keys, indicating that some servers have processed the read/write request, can be sent along with the partial-signing request as evidences to convince S_i to partially sign a response.

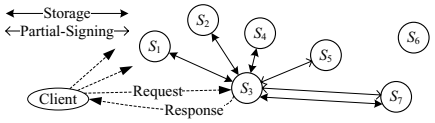

Fig. 1. General Model of Server Protocols

S_i can also generate a partial signature for each data item to make it self-verifying, before delegates write it to some quorum. This *data-signing* requires that: *a*) the service private key is available, and *b*) at least one correct server is involved in checking that the write request is generated by an authorized client (i.e., $h \geq f + 1$). For a *valid* server protocol, these two requirements of data-signing must be satisfied as well as service availability and service integrity (see Section 1). Fortunately, they are satisfied automatically when service availability and integrity are ensured. Otherwise, service availability isn't ensured if the service private key is unavailable, or service integrity isn't ensured if $h \leq f$ and then faulty servers can conspire to sign responses arbitrarily.

So, we skip this data-signing in the remainder due to limited space, focus on how to ensure service availability and integrity, and assume that each data item $[x, v, t]$ has been self-verifying when a client sends the write request.

Figure 1 shows the relationship of these three functions when $f = 2$ and $h = 3$, *not* showing the data-signing. A client sends a request to $f + 1$ servers. Then, S_3 assumes the role of delegate, performs the requested read/write operation on $2f + 1$ servers (including S_3), cooperates with h servers (also including S_3) to sign a response, and sends it to the client. It can be seen that in this flexible model, the server subset of partial-signing can *differ* from that of storage, though they may be the same in some instances or for some server protocols.

3.3 Defining the Validity of Server Protocols

As mentioned in Section 1, to provide BFT storage services as traditional BQS, an integrated TSS-BQS system must provide two upgraded security properties:

Service Availability. A read/write request from authorized clients gets a signed response to it, which is signed using the service private key.

Service Integrity. Each signed response guarantees that the requested operation has been performed on some quorum of servers. That is, a write response is signed only if the request has been delivered to at least $2f + 1$ servers; and a signed read response is derived from (different) replicas of $2f + 1$ servers, returning the right replica which is written by the most recent write operation.

These two security properties produce the definition of *valid* server protocols. On receiving a read/write request from clients, a delegate starts the valid server protocol to perform the requested operation on some quorum of servers, get a response fully signed, and return the signed response to clients.

Definition 1 (Valid Server Protocol). *A valid server protocol guarantees that a response is fully signed using the service private key* if and only if *a read/write request (generated by an authorized client) is delivered to some correct server (i.e., a delegate) and the requested read/write operation is performed on each server in some quorum.* □

3.4 Existing Valid Server Protocols

Two valid server protocols are proposed in [10,26], ensuring service integrity through different mechanisms. Based on these protocols, we can design similar ones, leading to two families of TSS-BQS systems.

SP-I. COCA [26] is the first system integrating BQS with TSS: *a*) $3f+1$ servers compose a dissemination BQS to provide certificate query/update services; *b*) the service private key is distributed among the exactly same $3f+1$ servers; *c*) the threshold to sign certificates or responses is $f+1$; and *d*) before using its service key share to partially sign a response, each server examines that the corresponding read/write operation has been performed on $2f+1$ servers. Since up to f servers can be faulty, at least one correct server carries out the necessary examinations before partially signing it, ensuring service integrity.

By increasing the threshold to sign of COCA, we can get a family of similar server protocols, where service integrity is still ensured through the examinations by the correct server(s) partially signing responses. For example, $h = f+2$ while all other features keep unchanged. Then, service integrity is ensured repeatedly, because there are at least two correct servers carrying out the examinations.

SP-II. Another valid protocol (SP-II) is suggested in [10]: *a*) the threshold to sign is equal to the size of a quorum (i.e., $h = 2f+1$); and *b*) each server processes the read/write request itself before partially signing a response. Thus, when a response is fully signed, $2f+1$ servers must have performed the requested operation; and service integrity is ensured through the threshold to sign.

By requiring servers of SP-II to carry out additional examinations, we can get another family of server protocols, where service integrity is ensured through the threshold to sign. For example, before partially signing the response, each server processes the read/write request itself and examines that the requested operation has been performed on d ($d < 2f+1$) servers. However, service integrity is still ensured through the threshold to sign (but not the examinations), because the d servers may be a subset of the h servers signing responses and multiple examinations together don't guarantee that the operation has been performed on more than h servers.

3.5 Is It Possible to Find Any Third Family of Valid Protocols?

SP-I and SP-II ensure service integrity of TSS-BQS systems through different mechanisms. Some questions appear when we analyze these protocols. Firstly, are there any valid server protocols ensuring service integrity through mechanisms *essentially* different from SP-I and SP-II? We had tried to design a different

one, but the outcome is always similar to SP-I or SP-II. Secondly, can we design valid server protocols with a *combined* mechanism requiring fewer examinations than SP-I while having a smaller threshold to sign than SP-II? Such combined protocols might have advantages of both SP-I and SP-II, and offer balanced performance. Finally, (if such a combined mechanism exists) can we discover the relationship of these two mechanisms to ensure service integrity? The relationship may lead to parameterized TSS-BQS systems with flexible configurations.

4 Main Results

In this section, we present the validity theory on TSS-BQS systems. The "soul" of this theory is to identify the "bonds" between the validity definition (Definition 1) and the design space for server protocols in a mathematically rigorous way so that a formal proof of our impossibility conjecture is derived. In particular, we prove that: *a*) there exist *only two* families of valid server protocols integrating BQS with TSS; *b*) service integrity is ensured through either the threshold to sign or the examinations by the correct server(s) partially signing responses; and *c*) *nobody* can design a combined mechanism, e.g., a server protocol requiring fewer examinations than SP-I and having a smaller threshold than SP-II.

4.1 Design Space for Server Protocols

Based on the general model in Section 3.2, it can be seen that the design flexibilities of server protocols are mainly associated with how servers validate the correctness of a response (to be signed) and partially sign it. We find that the design flexibilities can be "captured" by a rather simple concept called *signing-condition* (i.e., the condition to satisfy when a server partially signs a response).

For example, following some server protocol, a *correct* server S_i can partially sign a response *only if* it receives messages indicating that the corresponding read/write request has been processed by certain servers, e.g., replicas or acknowledgements signed using their server keys. So, when S_i partially signs a response, it is asserted that the operation has been performed on these servers. However, it doesn't mean that all these servers strictly serve the storage function. A read/write operation is defined to be performed on a server of BQS [13], if it receives the request and replies with a replica or acknowledgement.

Definition 2 (Signing-Condition). *Given a server protocol, whenever S_i (either correct or faulty) uses its service key share to partially sign a response, it is asserted that the corresponding read/write operation has been performed on some subset of servers. This specific subset is called one* signing-condition *of S_i, denoted $C(S_i)$.* □

Given one server protocol, the condition enabling a server to partially sign a response can be *not* unique. Rather, alternative signing-conditions exist. In Figure 1, for example, S_5 uses its service key share to partially sign a response

after it is convinced that $\{S_1, S_2, S_3, S_4, S_7\}$ have processed the read/write request. However, if the delegate S_3 performs the operation on another quorum of servers (e.g., $\{S_1, S_2, S_3, S_4, S_5\}$ or $\{S_2, S_3, S_4, S_5, S_6\}$), S_5 will also partially sign it. Thus, there are at least three alternative signing-conditions of S_5.

Definition 3 (Signing-Condition-Set). *Given a server protocol, the set of* all *signing-conditions $C(S_i)$ of S_i is called the* signing-condition-set $\mathscr{C}(S_i)$ *of S_i; i.e., $\mathscr{C}(S_i) = \{C(S_i)\}$.* □

Correct and faulty servers often have different signing-condition-sets, because faulty servers can partially sign either correct or fake responses (without satisfying the conditions as correct servers must follow).

4.2 Properties of Valid Server Protocols

Assumption 1. *For a* valid *server protocol, the read/write request from clients is included in the partial-signing request (sent by a delegate to servers).*

Justification. When requesting servers to partially sign a response, a delegate shall firstly convince them that an authorized client has sent the corresponding request. Otherwise, if a response is signed even when clients don't send the request, this response can be cached by faulty servers to launch attacks later. For example, if faulty servers can (convince others to) sign responses returning the *current* right replica with "potential" nonce when there doesn't exist a read request, these signed responses can be accepted by clients later, even when the returned "right" replica is updated by some write operation.

So, it is necessary for (correct) servers to check that the corresponding request exists before they partially sign a response. A safe and straightforward way is to include the intact read/write request in each partial-signing request, and then any server can authenticate it[1]. □

Assumption 2. *On receiving a read/write request, either forwarded by delegates directly or sent along with a partial-signing request, a* correct *server performs the requested read/write operation.*

Justification. Each *correct* server S_i of TSS-BQS systems is firstly a server of BQS and serves the basic storage function. On receiving a read/write request forwarded by delegates, S_i acts as a server of BQS [13,14]: it replies with its replica or an acknowledgement signed using its *server key*, and only updates its replica if the data item being written has a higher timestamp than its own.

On receiving a read/write request sent along with a partial-signing request (to sign a replica to be returned or an acknowledgement)[2], S_i uses its *service*

[1] We can require clients to sign the read/write request; then each server can use clients' public keys to authenticate it. And each client's public key (or certificate) can also be stored in the TSS-BQS system as a self-verifying variable and servers can (act as *read-only* clients to) read it, making the system self-contained.

[2] In the meantime, some signing-condition of S_i must be satisfied.

key share to partially sign a) the replica to be returned if it is identical with its own[3], or b) the acknowledgement. S_i also updates its replica if the data item being written has a higher timestamp than its own. In this case, S_i *actually* acts the *same* as that of BQS: serves the storage function and signs the same messages, except that one is signed using its server key and the other is done using its service key share, which are both held by S_i only. □

Because TSS-BQS systems assume asynchronous channels, this assumption doesn't harm the security. Moreover, it allows more flexible and efficient server protocols. Firstly, it doesn't require a strict order between the read/write operation and the partial-signing of each server. Secondly, the read/write request can be sent along with the partial-signing request to reduce communication costs. In addition, a correct delegate also "sends" the request to itself, then serves the storage function and performs the requested read/write operation.

Lemma 1. *For a* valid *server protocol,* $\forall C(S_i) \in \mathscr{C}(S_i) : S_i \in C(S_i)$.

Proof. This lemma can be directly concluded from Assumptions 1 and 2. When S_i (either correct or faulty) accepts a partial-signing request and replies with a partial signature, this partial-signing means that S_i has a) received the corresponding read/write request according to Assumption 1, and b) performed the requested operation according to Assumption 2. So, $S_i \in C(S_i)$. □

In BQS (and TSS-BQS systems), "an operation performed on each server in some quorum" doesn't mean that all these servers strictly follow the server protocol and serve the storage function. As long as $2f+1$ servers reply with their replicas or acknowledgements (and the *correct* ones of them have accepted and processed the read/write request), data consistency is ensured. Although up to f faulty servers may send fake replicas or acknowledge write requests without updating their replicas, the negative impact of faulty servers and their replies are masked by (the correct ones in) any quorum of $2f+1$ servers.

Lemma 2. *For a* valid *server protocol,* $\mathscr{C}(S_i) = \{C : S_i \in C\}$ *if* S_i *is faulty.*

Proof. If S_i is faulty, it can uses its service key share to partially sign a response *arbitrarily*, whether with any process or not. Then, a partial signature by S_i may ensure *no* operations on any other servers except *itself* according to Lemma 1. So, any subset C containing S_i ($S_i \in C$) can be a signing-condition of S_i. □

Theorem 41. $\mathscr{C}(\cdot)$ *of valid server protocols satisfies the following* signing-condition-inequality*:*
For any h*-server set* $H = \{S_{i_1}, S_{i_2}, \cdots, S_{i_h}\}$ ($|H| = h$), $\forall C(S_{i_e}) \in \mathscr{C}(S_{i_e} : 1 \leq e \leq h) : \bigcup_H C(S_{i_e}) = C(S_{i_1}) \cup C(S_{i_2}) \cup \cdots \cup C(S_{i_h})$ *contains some quorum of* $2f+1$ *servers; that is,* $|\bigcup_H C(S_{i_e})| \geq 2f+1$.

[3] If the replica to be returned is unmodified and has a higher timestamp than its own, S_i also partially signs it. This case can be explained as two steps: firstly S_i updates its replica with the one to be returned (which must be written by a more recent write operation), and then partially signs the response returning its replica.

Proof. To prove this theorem by contradiction, assume a server protocol *not* satisfying the inequality, and then we will find that service integrity is *not* ensured either. If the signing-condition-inequality is not satisfied, there must exist an h-server set (denoted H'), $\exists C'(S_{i_e}) \in \mathscr{C}(S_{i_e} : S_{i_e} \in H') : |\bigcup_{H'} C'(S_{i_e})| < 2f + 1$.

On receiving a read/write request, a faulty server can serve as a delegate:

1. Perform the requested operation on servers in $\bigcup_{H'} C'(S_{i_e})$, and construct a response to be signed;

2. Request $S_{i_1} \in H'$ to partially sign the response, and S_{i_1} (either correct or faulty) will partially sign it because the operation has been performed on servers in $C'(S_{i_1}) \subseteq \bigcup_{H'} C'(S_{i_e})$;

3. Request $S_{i_2}, S_{i_3}, \cdots, S_{i_h} \in H'$ to partially sign the response, and collect the partial signatures generated by these h servers; and

4. Combine these h partial signatures into a signed response.

Then, this signed response will be accepted by the client even when the requested operation is performed on fewer than $2f + 1$ servers, i.e., $\bigcup_{H'} C'(S_i)$. Hence, service integrity is *not* ensured. So, the signing-condition-inequality is a *necessary* condition of valid server protocols. □

Theorem 42. *A server protocol is valid* if and only if *the following conditions are satisfied:*
A. There exists an h-server set H^* ($|H^*| = h$) consisting of correct *servers only; and $\exists C^*(S_i) \in \mathscr{C}(S_i)$ for all $S_i \in H^* : |\bigcup_{H^*} C^*(S_i)| \leq n - f = 2f + 1$.*
B. Every signing-condition-set $\mathscr{C}(\cdot)$ satisfies the signing-condition-inequality.

Proof. Necessity. Firstly, an h-server set H^* consisting of correct servers only, is necessary to sign responses using the service private key, when faulty servers don't partially sign any messages. Secondly, the subset that performs the requested read/write operation (i.e., $\bigcup_{H^*} C^*(S_i)$), shall be available if f servers are crash, so it cannot contain more than n-f servers. Thus, Condition-A is a necessary condition as well as the signing-condition-inequality.

Sufficiency. On receiving a read/write request from clients, a (correct) delegate can perform the requested operation on servers in $\bigcup_{H^*} C^*(S_i)$, cooperate with the h correct servers in H^* to sign the response, and send it to clients. So, the service is available. Service integrity is also ensured because the signing-condition-inequality is satisfied and $\bigcup_{H^*} C^*(S_i)$ contains some quorum. □

Lemma 3. *For a valid server protocol, $f + 1 \leq h \leq 2f + 1$.*

Proof. Firstly, there exists an h-server set consisting of correct servers only according to Theorem 42, and up to f out of n servers can be faulty, so $h \leq n - f = 2f + 1$. Secondly, let's prove $f + 1 \leq h$ by contradiction. Assume $h < f + 1$; there exists an h-server set consisting of *faulty* servers only (denoted $\bar{H}$). For each $S_j \in \bar{H}$, $\{S_j\}$ is a signing-condition of S_j according to Lemma 2. Then, $|\bigcup_{\bar{H}} C(S_j)| = |\bigcup_{\bar{H}} \{S_j\}| = |\bar{H}| = h < f + 1$, and the signing-condition-inequality is *not* satisfied. Thus, $h \geq f + 1$; i.e., faulty servers cannot conspire to use the service private key to sign responses arbitrarily. □

4.3 Two Families of Valid Server Protocols

We investigate $\mathscr{C}(S_i)$ of valid server protocols under the *uniformity* assumption.

Lemma 4. *Assuming that servers are uniform and S_i is correct, if $C \in \mathscr{C}(S_i)$ and $S_j \in C$, then $C \in \mathscr{C}(S_j)$, where $j \neq i$.*

Proof. According to Lemma 2, C is a signing-condition of S_j ($S_j \in C$) if S_j is faulty. Let's assume S_j is correct. Since C is a signing-condition of S_i, S_i uses its service key share to partially sign a response after examining that the requested operation has been performed on servers in C ($S_i \in C$). Following the same server protocol as S_i, S_j also partially signs it after examining that the requested operation has been performed on the same subset C ($S_j \in C$; the relationship between C and S_i is the same as that between C and S_j). Thus, C is also a signing-condition of S_j; i.e., $C \in \mathscr{C}(S_j)$. □

Lemma 5. *Assuming that servers are uniform and S_i is correct, if $C \in \mathscr{C}(S_i)$ and $S_j \notin C$, then $\{S_j\} \cup C' \in \mathscr{C}(S_j)$, where $C' = C \setminus \{S_i\}$.*

Proof. According to Lemma 2, $\{S_j\} \cup C'$ is a signing-condition of S_j if S_j is faulty. Let's assume S_j is correct. Since $C = \{S_i\} \cup C'$ is a signing-condition of S_i, S_i uses its service key share to partially sign a response, after it processes the request itself and examines that the requested operation has been performed on servers in C' ($S_i \notin C'$). Following the same server protocol as S_i, S_j also partially signs the response, after it processes the request itself and examines that the requested operation has been performed on the same subset C' ($S_j \notin C'$; the relationship between C' and S_i is the same as that between C' and S_j). Thus, $\{S_j\} \cup C'$ is a signing-condition of S_j; i.e., $\{S_j\} \cup C' \in \mathscr{C}(S_j)$. □

Theorem 43. *Assuming that servers are uniform, there are* only two *families of valid server protocols as listed below, and it is* impossible *to find any third family:*
1. $2f + 1 > h \geq f + 1$, and for any h-server set H, $\exists S^* \in H : \mathscr{C}(S^*) = \{C : S^* \in C \wedge |C| \geq 2f + 1\}$.
2. $h = 2f + 1$, and $\forall C(S_i) \in \mathscr{C}(S_i) : S_i \in C(S_i)$.

Proof. Two cases are analyzed to find h and $\mathscr{C}(\cdot)$ of valid server protocols. Note that these complementary cases cover *all possible* scenarios.

1. For any h-server set H, $\exists S^* \in H, \forall C(S^*) \in \mathscr{C}(S^*) : |C(S^*)| \geq 2f + 1$.
The signing-condition-inequality is satisfied without additional constraints because $|\bigcup_H C(S_i)| \geq |C(S^*)| \geq 2f + 1$. Furthermore, since $S_i \in C(S_i)$ according to Lemma 1, $\mathscr{C}(S^*) = \{C : S^* \in C \wedge |C| \geq 2f + 1\}$.

2. There exists an h-server set (denoted $\widetilde{H}$), $\forall S_i \in \widetilde{H}, \exists \widetilde{C} \in \mathscr{C}(S_i) : |\widetilde{C}| < 2f + 1$.
According to Lemma 3, $h \geq f + 1$ and there is at least one *correct* server $\widetilde{S} \in \widetilde{H}$. There exists $\widetilde{C} \in \mathscr{C}(\widetilde{S}) : |\widetilde{C}| < 2f + 1$. For each $S_j \in \widetilde{C}$, $\widetilde{C}$ is a signing-condition of S_j according to Lemma 4. Select all servers in $\widetilde{C}$, and then

$|\bigcup_{\widetilde{C}} C(S_j)| = |\widetilde{C} \cup \cdots \cup \widetilde{C}| = |\widetilde{C}| < 2f+1$. So, in order to satisfy the signing-condition-inequality, more servers than $\widetilde{C}$ are needed to compose a valid h-server set, i.e., $h > |\widetilde{C}|$. Then, we can find an h-server set $\widetilde{H}^* \supset \widetilde{C}$.

For each $S_k \in \widetilde{H}^* \setminus \widetilde{C}$, $\{S_k\} \cup \widetilde{C}'$ is a signing-condition of S_k according to Lemma 5, where $\widetilde{C}' = \widetilde{C} \setminus \{\widetilde{S}\}$; and $|\bigcup_{\widetilde{H}^*} C(\cdot)| = |\bigcup_{\widetilde{C}} C(S_j) \bigcup_{\widetilde{H}^* \setminus \widetilde{C}} C(S_k)| = |\widetilde{C} \cup \cdots \cup \widetilde{C} \bigcup_{\widetilde{H}^* \setminus \widetilde{C}} (\{S_k\} \cup \widetilde{C}')| = |\widetilde{C} \bigcup_{\widetilde{H}^* \setminus \widetilde{C}} \{S_k\}| = |\widetilde{H}^*| = h$. So, $h \geq 2f+1$ to satisfy the signing-condition-inequality.

Thus, $h = 2f+1$ because $2f+1 \geq h$ according to Lemma 3; and the signing-condition-inequality is satisfied: $|\bigcup_H C(S_i)| \geq |\bigcup_H \{S_i\}| = |H| = h = 2f+1$.

These two solutions cover all possible scenarios and there is *no* other solution for the signing-condition-inequality (i.e., Condition-B of Theorem 42), and it can be verified that these solutions also satisfy Condition-A. Therefore, according to Theorem 42, they are the all solutions (or valid server protocols) of TSS-BQS systems, and no other valid protocol exists. □

These solutions correspond to two families of valid server protocols, respectively. The first family satisfies the signing-condition-inequality (i.e., ensures service integrity) through the examinations by correct server S^* ($|C(S^*)| \geq 2f+1$), and the second does through the threshold to sign ($h = 2f+1$). There is no valid protocol with combined mechanisms requiring fewer examinations than $2f+1$ while having a threshold $h < 2f+1$. Although we can design a protocol where $|C(\cdot)| \geq 2f+1$ for correct servers and $h = 2f+1$, service integrity is ensured *repeatedly* through each of these two mechanisms, instead of a *combined* one.

5 Efficient Server Protocols

In this section, two existing server protocols [10,26] are deduced by minimizing the computation costs of the solutions predicted in Theorem 43. Two types of computations are reflected in these solutions as follows:

Partial-signing using service key shares. The computation cost is measured by h: h partial signatures are needed to fully sign a response.

Examinations that the requested read/write operation has been performed on certain servers. The computation cost is measured by $|C(\cdot)|$: before partially signing a response, a (correct) server S_i verifies messages which are signed using server keys, to examine that servers in $C(S_i)$ have processed the read/write request. In fact, since $S_i \in C(S_i)$, it can verify only $|C(S_i)| - 1$ messages from other servers.

By minimizing the amount of computations (i.e., choosing the minimal h and $|C(\cdot)|$ allowable), we find two solutions of *efficient* (and valid) server protocols:

1. $h = f+1$, and for any h-server set H, there exists a (correct) server $S^* \in H : \mathscr{C}(S^*) = \{\{S^*\} \cup C' : |C'| = 2f+1\}$ (C' may contain S^* or not).
2. $h = 2f+1$, and $\mathscr{C}(S_i) = \{\{S_i\}\}$ if S_i is correct.

Based on the efficient solutions, we design two server protocols as below. They are essentially the same as SP-I [26] and SP-II [10], which may have additional design details for specific applications (e.g., the means of generating timestamps).

5.1 Server Protocol I

The service private key is shared by $3f + 1$ servers, and the threshold to sign is $f + 1$. Servers use the following protocol, and S_d is a delegate.

A. On receiving a read/write request from clients, S_d forwards it to all servers.

B. On receiving a read/write request from S_d, S_i uses its server key to sign a reply and sends it to S_d. The reply is its replica for reading. For writing, S_i replies with an acknowledgement, and only updates its replica if the data item being written has a higher timestamp than its own.

C. S_d repeats Step-A periodically until it receives replies from $2f + 1$ servers.

D. S_d generates a partial-signing request and sends it to all servers. The partial-signing request includes: the read/write request, the response (to be signed), and those $2f + 1$ replies collected in Step-C. For reading, the response is the right replica (i.e., the unmodified one with the highest timestamp out of those $2f + 1$ replicas). For writing, the response is an acknowledgement.

E. On receiving a partial-signing request from S_d, S_i uses its service key share to generate a partial signature for the response and sends it to S_d, after examining that those included replies are generated by $2f + 1$ servers for the included read/write request. For reading, S_i also examines that the replica to be returned a) is the unmodified one with the highest timestamp out of those included $2f + 1$ replicas, and b) has a timestamp higher than or identical with its own. Otherwise, S_i replies to S_d with a rejection. For writing, S_i also updates its replica if the data item being written has a higher timestamp than its own.

F. S_d repeats Step-D periodically until it receives partial signatures from $f + 1$ servers, or re-starts from Step-A if it receives $f + 1$ rejections for reading (happening when a write operation overlaps the read operation). S_d combines these $f + 1$ partial signatures into a fully signed response and sends it to clients.

5.2 Server Protocol II

The threshold to sign of SP-II is equal to the size of a quorum (i.e., $h = 2f + 1$). Steps for reading and writing are described separately, and S_d is a delegate.

Read

A. On receiving a read request from clients, S_d generates a partial-signing request and sends it to all servers. The partial-signing request includes: the read request, and the response (to be signed) which includes the right replica to be returned. S_d sets the "right" replica to its own replica *tentatively*.

B. On receiving a partial-signing request from S_d, S_i uses its service key share to generate a partial signature for the response and sends it to S_d, after checking that the replica (to be returned) is unmodified and has a timestamp higher than or identical with its own. Otherwise, S_i replies to S_d with a rejection.

C. S_d repeats Step-A periodically until it receives partial signatures from $2f + 1$ servers, or breaks to Step-D if it receives $f + 1$ rejections. S_d combines these $2f + 1$ partial signatures into a fully signed response and sends it to clients.

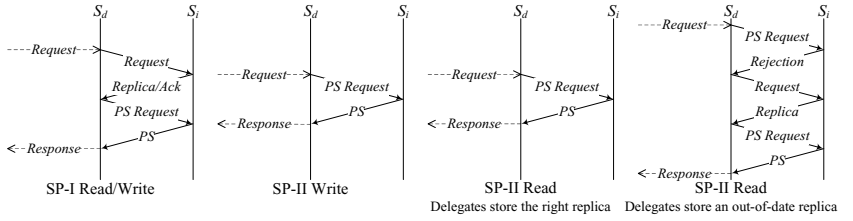

Fig. 2. Communication Costs of SP-I and SP-II

D. S_d forwards the read request (from clients) to all servers. Step-D is executed only if S_d receives $f + 1$ rejections, happening when it doesn't store the right replica and shall collects replicas from other servers to update its own.

E. On receiving a read request from S_d, S_i replies with its replica.

F. S_d repeats Step-D periodically until it receives replicas from $2f+1$ servers. S_d obtains the right replica out of these $2f + 1$ replicas, updates its own, and re-starts from Step-A.

Write

A. On receiving a write request from clients, S_d generates a partial-signing request and sends it to all servers. The partial-signing request includes: the write request, and the response (to be signed) which is an acknowledgement.

B. On receiving a partial-signing request from S_d, S_i uses its service key share to generate a partial signature for the response and sends it to S_d. S_i also updates its replica if the data item being written has a higher timestamp.

C. S_d repeats Step-A periodically until it receives partial signatures from $2f + 1$ servers. S_d combines these $2f + 1$ partial signatures into a fully signed response and sends it to clients.

6 Performance

In this section, SP-I and SP-II are compared in terms of communication costs, computation costs and the read responses on concurrent read/write operations.

6.1 Communication

Assuming that there are no concurrent read/write operations, Figure 2 shows the communication costs of SP-I and SP-II. It can be seen that SP-I always needs two rounds of communications among servers, while SP-II-write needs only one round, because SP-II doesn't need to collect process results from some quorum as SP-II does before delegates request other servers to partially sign responses.

The communication cost of SP-II-read varies whether delegates store the right replica or not. If S_d stores the right replica, one round is enough. Otherwise, three rounds are needed: after receiving $f+1$ rejections, S_d collects replicas to update its own, and requests other servers to partially sign the response again.

6.2 Computation

The major computation costs of TSS-BQS systems are public key cryptographic computations [15,26]: signing using server keys and partial-signing using service key shares. Firstly, the computation costs of server key depend on the communications among servers, because they are used to sign messages among servers.

Secondly, while SP-II always needs partial signatures by more f servers than SP-I, the cost of each partial-signing varies with different TSS. The following analysis is specific to the scheme used in [15,26,27]: each partial-signing includes $L_{n,h} = l(n-h+1)/n$ modular exponentiations of long integer (e.g., 1024 bits), where $l = \binom{n}{h-1}$. It can be verified that $L_{3f+1,f+1} = L_{3f+1,2f+1}$, i.e., each partial-signing of SP-I and SP-II costs approximately equal resources.

6.3 Concurrent Read/Write Operations

Since each read/write operation must be performed on $2f+1$ servers and may last a long time, concurrent operations can happen usually in TSS-BQS systems. We analyze the responses of the read operations overlapped by a concurrent write operation, and firstly define the windows of operations:

Read. A read operation returning $[x, v_r, t_r]$ from some quorum starts (denoted T_{rs}) when the first server in this quorum receives the read request and replies with its replica, and ends (denoted T_{re}) when the delegate determines to return $[x, v_r, t_r]$, which is eventually signed and sent to clients. Note that the delegate may determine and be rejected for several times before the operation ends.

Write. An operation writing $[x, v_w, t_w]$ on some quorum starts (denoted T_{ws}) when the first *correct* server in this quorum receives the write request and updates its replica, and ends (denoted T_{we}) when the last *correct* one in this quorum does. Variable x really starts to change only when a correct server receives the write request, because even if faulty servers receive the request before T_{ws}, they can drop it maliciously.

Assume that $[x, v_0, t_0]$ is the right replica before the concurrent operation writing $[x, v_w, t_w]$ and $t_w > t_0$. All situations of concurrent operations are analyzed as below.

1. $T_{ws} < T_{rs} < T_{we} < T_{re}$ or $T_{rs} < T_{ws} < T_{we} < T_{re}$

SP-I may return $[x, v_w, t_w]$ or $[x, v_0, t_0]$, while SP-II always returns $[x, v_w, t_w]$ at the cost of (possible) more rounds of communications among servers. Following SP-I, the $2f+1$ replicas sent along with the partial-signing request as evidences, may be collected before T_{we} and contain $[x, v_0, t_0]$ only (e.g., these replicas are collected when only one correct server has updated its replica). And the $f+1$ servers signing the read response, may contain only another correct server which doesn't receive the concurrent write request or update its replica.

Following SP-II, the $2f+1$ servers signing the read response, must contain one correct server which has received $[x, v_w, t_w]$ when T_{we}, because BQS guarantee that the intersection of any two quorums contains at least one correct server. This correct server partially signs the response only if the replica to be returned has a timestamp higher than or identical with its own (i.e., $t_r \geq t_w$); otherwise, it rejects to sign it, leading to two more rounds of communications.

2. $T_{ws} < T_{rs} < T_{re} < T_{we}$ or $T_{rs} < T_{ws} < T_{re} < T_{we}$

Both SP-I and SP-II may return $[x, v_w, t_w]$ or $[x, v_0, t_0]$. Although at least one correct server has received $[x, v_w, t_w]$ after T_{ws}, it may *not* be involved in the concurrent read operation at all. It is possible that all servers involved in the read operation, store $[x, v_0, t_0]$ only; and then the read response returns $[x, v_0, t_0]$.

7 Related Work

BQS of self-verifying data over asynchronous, authenticated and reliable channels are proposed in [13]; variations of other data or over different channels can be found in [4,13,17]. Dynamic BQS [1,11,16] can reconfigure the number of servers and faulty ones (i.e., dynamic n and f).

Several distributed storage systems [7,9,22,24,25] apply threshold cryptography (e.g., secret sharing, erasure code, etc.) to protect data integrity and confidentiality. In [18] and [23], secret sharing is integrated with quorum systems and BQS, respectively, to provide fault-tolerant storage services.

TSS are utilized to sign messages in state machine replication [2,3,20,21] and distributed storage systems [9,22]: signatures by TSS indicate that enough servers agree with the content of signed messages or have performed the requested operations, masking the impact of faulty servers. COCA [26] is the first work to integrate BQS with TSS, and its protocol is adopted in CODEX [15] to store secret data. [10] proposed another server protocol of TSS-BQS systems.

8 Conclusions

To provide *self-contained* BFT storage services of self-verifying data, traditional BQS are no longer sufficient. Achieving this goal demands the integration of BQS and TSS, and only two *valid* TSS-BQS systems have been proposed in the literature. Based on these two systems, we can find similar server protocols, leading to two families of TSS-BQS systems. We develop a *validity theory* on server protocols of TSS-BQS systems and formally prove that it is *impossible* to find any third family of valid TSS-BQS systems. It is also shown that the *only two* families of valid server protocols "predicted" (or deduced) by the proposed theory precisely match the existing protocols.

References

1. Alvisi, L., Dahlin, M., et al.: Dynamic Byzantine quorum systems. In: Int'l. Conf. Dependable Systems and Networks, pp. 283–292 (2000)
2. Amir, Y., Coan, B., et al.: Customizable fault tolerance for wide-area replication. In: IEEE Symp. Reliable Distributed Systems, pp. 65–82 (2007)
3. Amir, Y., Danilov, C., et al.: Scaling Byzantine fault-tolerant replication to wide area networks. In: Int'l. Conf. Dependable Systems and Networks, pp. 105–114 (2006)
4. Bazzi, R.: Synchronous Byzantine quorum systems. Distributed Computing 13(1), 45–52 (2000)

5. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Computer Systems 20(4), 398–461 (2002)
6. Desmedt, Y.: Society and group oriented cryptography: A new concept. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 120–127. Springer, Heidelberg (1988)
7. Goodson, G., Wylie, J., et al.: Efficient Byzantine-tolerant erasure-coded storage. In: Int'l. Conf. Dependable Systems and Networks, pp. 135–144 (2004)
8. Herzberg, A., Jakobsson, M., et al.: Proactive public key and signature systems. In: ACM Conf. Computer Communications Security, pp. 100–110 (1997)
9. Iyengar, A., Cahn, R., et al.: Design and implementation of a secure distributed data repository. In: IFIP Int'l. Information Security Conference, pp. 123–135 (1998)
10. Jing, J., Wang, J., et al.: Research on server protocols of Byzantine quorum systems implemented utilizing threshold signature schemes (accepted to appear). Chinese Journal of Software
11. Kong, L., Subbiah, A., et al.: A reconfigurable Byzantine quorum approach for the Agile Store. In: IEEE Symp. Reliable Distributed Systems, pp. 219–228 (2003)
12. Lamport, L., Shostak, R., et al.: The Byzantine generals problem. ACM Trans. Programming Languages and Systems 4(3), 382–401 (1982)
13. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11(4), 203–213 (1998)
14. Malkhi, D., Reiter, M.: Secure and scalable replication in Phalanx. In: IEEE Symp. Reliable Distributed Systems, pp. 51–60 (1998)
15. Marsh, M., Schneider, F.: CODEX: A robust and secure secret distribution system. IEEE Trans. Dependable and Secure Computing 1(1), 34–47 (2004)
16. Martin, J.-P., Alvisi, L.: A framework for dynamic Byzantine storage. In: Int'l. Conf. Dependable Systems and Networks, pp. 325–334 (2004)
17. Martin, J.-P., Alvisi, L., et al.: Small Byzantine quorum systems. In: Int'l. Conf. Dependable Systems and Networks, pp. 374–383 (2002)
18. Naor, M., Wool, A.: Access control and signatures via quorum secret sharing. IEEE Trans. Parallel and Distributed Systems 9(9), 909–922 (1998)
19. Ostrovsky, R., Yung, M.: How to withstand mobile virus attacks. In: ACM Symp. Principles of Distributed Computing, pp. 51–59 (1991)
20. Reiter, M., Birman, K.: How to securely replicate services. ACM Trans. Programming Languages and Systems 16(3), 986–1009 (1994)
21. Reiter, M., Franklin, M., et al.: The Ω key management service. In: ACM Conf. Computer and Communications Security, pp. 38–47 (1996)
22. Rhea, S., Eaton, P., et al.: Pond: the OceanStore prototype. In: USENIX Conf. File and Storage Technologies, pp. 1–14 (2003)
23. Subbiah, A., Ahamad, M., et al.: Using Byzantine quorum systems to manage confidential data. Technical Report GIT-CERCS-04-13, Georgia Institute of Technology (2004)
24. Subbiah, A., Blough, D.: An approach for fault tolerant and secure data storage in collaborative work environments. In: ACM Workshop on Storage Security and Survivability, pp. 84–93 (2005)
25. Wylie, J., Bigrigg, M., et al.: Survivable information storage systems. IEEE Computer 33(8), 61–68 (2000)
26. Zhou, L., Schneider, F., et al.: COCA: A secure on-line certification authority. ACM Trans. Computer Systems 20(4), 329–368 (2002)
27. Zhou, L., Schneider, F., et al.: APSS: Proactive secret sharing in asynchronous systems. ACM Trans. Information and System Security 8(3), 259–286 (2005)