The Deferred Byzantine Generals Problem

Xuyang Liu, *Graduate Student Member, IEEE*, Zijian Zhang, *Senior Member, IEEE*, Zhen Li, Peng Jiang, Yajie Wang, Meng Li, *Senior Member, IEEE*, and Liehuang Zhu, *Senior Member, IEEE*

***Abstract*—This paper introduces the Deferred Byzantine Generals Problem, a variant of the Byzantine Generals Problem which focuses on ensuring replicas maintain consistency over timed-release secret operations (operations that can only be known after a specified time or event). The solution to the problem is called the Deferred Byzantine Fault Tolerant (DBFT) consensus. DBFT can operate exclusive or be interleave with BFTs to handle specific tasks at designated sequence numbers or views, thereby facilitating the implementation of certain system-desirable features or supporting novel applications. It does not rely on existing timed-release primitives, but instead ensures its timed-release property through voting interactions. We presents the system model of DBFT SMR under partial synchronization using Threshold Public Key Encryption (TPKE) as the cryptographic primitives, highlighting the core issues. Then we design and implement the DBFT protocol using PBFT notations, focusing on the unique parts to facilitate expansions to other paradigms. Through experimental results, we show the impact of different executing modes and parameter choices on performance and discuss potential optimizations.**

***Index Terms*—Distributed consensus, Byzantine fault tolerance, timed-release operations, threshold public key encryption.**

I. Introduction

FAULT-TOLERANT consensus are fundamental to distributed computing, where the reliability and consistency of the system are paramount. Ensuring that a network of replicas can agree on a common value or a sequence of operations, despite failures, remains a profound challenge that underpins the operation of distributed systems. These mechanisms are mainly classified into Crash Fault Tolerance (CFT) [1], which handles benign crashes (i.e., stop working entirely, but do not behave maliciously), and Byzantine Fault Tolerance (BFT) [2], which tackles the Byzantine Generals Problem, covering a broader range of fault modes, including malicious behaviors that can undermine the consensus process.

With the rising of blockchain technology [3], time-based primitives [4], [5] like Time-Lock puzzles [5], [6] and timed-release encryption [7], [8] are finding widespread use in practical protocols. These primitives are perceived as solutions to many challenges, such as scheduled confidential transactions [9], and potential applications like decentralized smart contracts [10]. In this paper, we abstract a timed-release variant of the Byzantine Generals Problem, called the Deferred Byzantine Generals Problem. The problem focuses on ensuring replicas maintain consistency on a sequence of timed-release secret operations (operations that can only be known after a specified time or event) in the presence of Byzantine failures. The corresponding consensus protocol is referred to as the Deferred Byzantine Fault Tolerant (DBFT) consensus.

DBFT can be run as the exclusive consensus protocol for a system or interleaved with a BFT protocol. In the latter case, it can be used to handle specific tasks assigned to certain sequence numbers, views, or rounds (or is triggered on demand), thereby facilitating the implementation of certain system-desirable features, such as periodically agreeing on globally consistent scheduled operations (e.g., coin toss [11], reconfiguration [12]) or supporting novel applications such as scheduled confidential transactions [9] and sealed-bid auctions [13]. Additionally, the Deferred Byzantine Generals Problem encompasses the interactive consistency conditions of the original problem, meaning that when DBFT is interleaved with a BFT protocol, the operations of the two parts can be indiscriminate in terms of the total order.

Cryptographic primitives most closely related to DBFT include timed-release computational secret sharing [8] and threshold timed-release encryption [10]. Analogous to how BFT relates to normal threshold cryptography [14], [15], DBFT ensures consistency in timed-release secret operations among replicas (i.e., State Machine Replication, SMR [16]), rather than generating signatures, encrypting messages, or recovering secrets under timed-release mechanisms. For example, DBFT and timed-release computational secret sharing share key commonalities: both utilize threshold mechanisms requiring a designated number of shares to reconstruct a secret, and both involve distributing information inaccessible until certain conditions are met (typically a timing event). However,

Received 21 November 2024; revised 23 May 2025 and 25 June 2025; accepted 9 July 2025. Date of publication 24 July 2025; date of current version 31 July 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFF0905300; in part by the National Natural Science Foundation of China (NSFC) under Grant U2468205, Grant 62372149, and Grant U23A20303; in part by China Scholarship Council (CSC); and in part by the Key Laboratory of Knowledge Engineering with Big Data (Ministry of Education of China) under Grant BigKEOpen2025-04. The associate editor coordinating the review of this article and approving it for publication was Prof. Haibo Hu. *(Corresponding authors: Zijian Zhang; Meng Li.)*

Xuyang Liu is with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China, and also with the School of Computer Science, The University of Auckland, Auckland 1010, New Zealand (e-mail: liuxuyang@bit.edu.cn).

Zijian Zhang, Zhen Li, Peng Jiang, Yajie Wang, and Liehuang Zhu are with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: zhangzijian@bit.edu.cn; zhen.li@bit.edu.cn; pengjiang@bit.edu.cn; wangyajie0312@foxmail.com; liehuangz@bit.edu.cn).

Meng Li is with the Key Laboratory of Knowledge Engineering with Big Data, Ministry of Education, the School of Computer Science and Information Engineering, and the Intelligent Interconnected Systems Laboratory of Anhui Province, Hefei University of Technology, Hefei 230601, China, and also with the HIT Center, University of Padua, 35131 Padua, Italy (e-mail: mengli@hfut.edu.cn).

Data is available on-line at https://ieeexplore.ieee.org

This article has supplementary downloadable material available at https://doi.org/10.1109/TIFS.2025.3592566, provided by the authors.

Digital Object Identifier 10.1109/TIFS.2025.3592566

DBFT specifically focuses on ensuring consistency among all non-faulty replicas (participants) for timed-release secret operations, while timed-release secret sharing is only concerned with privacy (all participants can obtain no information on the secret before the designated time), security (at most $k-1$ participants can obtain no information on the secret) and consistency (at least k valid shares from distinct participants must correctly reconstruct the secret after the designated time) of secret recovery. To illustrate this distinction, consider a $(k = 2m + 1, n = 3m + 1)$-timed-release secret sharing scheme with m malicious participants (n participants in total), where a secret can be reconstructed by at least k shares after the designated release time specified by the dealer. This threshold is consistent with that in DBFT. In this scenario, for example:

1) A malicious dealer (who is also a malicious participant) could create two conflicting secret operations with the same release time - e.g., in the context of decentralized scheduled confidential transactions: op_1: "transfer 5 tokens to p_1" and op_2: "transfer 5 tokens to p_2" (when the dealer only has 6 tokens). The dealer distributes op_1 shares to m malicious participants (set R_m) and $m + 1$ non-faulty participants (set R_1 where $p_1 \in R_1$), and similarly distributes op_2 shares to m malicious participants (set R_m) and $m + 1$ non-faulty participants (set R_2 where $p_2 \in R_2$, with $R_1 \cap R_2 = \{p^*\}$). If the m malicious participants only send their op_1 shares to set R_1 and op_2 shares to set R_2 (while non-faulty participants broadcast their shares), then after the designated time, inconsistency occurs - some non-faulty participants recognize the payment to p_1, others to p_2, which may result in a double-spending attack.
2) Similarly, a malicious dealer could create a secret operation op_1 ("transfer 5 tokens to p_1") but distribute shares only to m malicious participants and $m + 1$ non-faulty participants (set R_1 including p_1). If the m malicious participants only share their portions with p_1 (while non-faulty participants broadcast their shares), then after the designated time, among non-faulty participants, only p_1 would believe it received tokens, creating a situation where the payment appears successful to p_1 but p_1 cannot effectively use these tokens with other participants.

DBFT, as long as no more than m out of $3m + 1$ replicas (participants) are faulty, prevents these inconsistencies by ensuring all non-faulty replicas maintain consensus, collectively recognizing either a single valid operation or no operation at all in the above examples. This consistency guarantee extends beyond what timed-release computational secret sharing schemes can provide alone.

DBFT is also highly extensible, allowing modifications in the design of the secret operation maker or the type of timed events to fit different paradigms. For example, it can be extended to the "schedule and reveal" paradigm (it should be noted, however, that in the Deferred Byzantine Generals Problem, each secret-generating general must be a non-essential participant in the subsequent knowledge of its respective parts of the operation; that is to say, their revelation is non-essential). It can also be extended to a paradigm similar to Time-Lock Puzzles without the need for complex puzzles, instead leveraging any timed event.[1] This is because DBFT, as a timed-release consensus protocol, does not rely on existing timed-release cryptographic primitives (although such primitives could be employed). The timed-release feature essentially depends on voting interactions without requiring replicas to dedicate substantial computational resources to solving puzzles during the waiting period, which is guaranteed with faults constrained by the tolerance (and cannot be solved by leveraging more computational resources). While Time-Lock Puzzles require significant computational resources for puzzle solving, potentially impacting system performance for other tasks. In our implementation, we employ threshold public key encryption (TPKE) [17], [18] as the cryptographic primitives. It should be noted, however, this is not the only possible approach.

We construct a system model for DBFT under partial synchronization using TPKE and formalize it in the state machine replication (SMR) [16] form, highlighting the core issues DBFT addresses. We then present the DBFT protocol using PBFT [19] notations, focusing on the modifications and unique challenges in comparison to the original PBFT protocol. In doing so, we seek to demonstrate how existing BFT systems can be modified to support the DBFT protocol. Given the foundational nature of PBFT in BFT protocols, we believe our solution can be easily extended to most BFT paradigms if using the same cryptographic primitives. We also prove the correctness of the DBFT protocol.

To evaluate our solution, we implemented a system that supports the DBFT protocol based on an existing PBFT codebase. The experiments were conducted on up to six servers, with throughput and latency being the main metrics. We assessed the system's baseline performance and scalabil ity when running DBFT exclusively under various network conditions. Additionally, we measured the performance of DBFT when interleaved with the BFT protocol under different secret operation frequencies and time event lengths. The results demonstrate the impact of parameter selection (e.g., the trade-offs between secret operation frequency and throughput) on systems supporting DBFT. We also discuss potential optimizations for improving performance.

Thus, the paper makes the following contributions:

- It abstracts the Deferred Byzantine Generals Problem, using a formulation similar to the original problem, with a focus on the deferred aspect, and defines the interactive consistency conditions. To the best of our knowledge, we

[1]Furthermore, despite their similarities in paradigm, several critical distinctions between them remain. First, similar to the distinctions between DBFT and timed-release secret sharing (as discussed above), DBFT specifically focuses on ensuring consistency in timed-release operations among non-faulty replicas in distributed consensus environments, whereas Time-Lock Puzzles is only concerned with sequentiality and efficiency of individual puzzles without addressing distributed consensus (as well as Byzantine fault tolerance). Time-Lock Puzzles alone cannot prevent inconsistencies where malicious replicas distribute different secret operations at the same sequence position. Additionally, since DBFT encompasses the interactive consistency conditions of the original Problem, it can easily interleave with existing BFT protocols, and the service operations of the two parts can be indiscriminate in terms of the total order - functionality that would be complex to implement with Time-Lock Puzzles due to the Universal Composability security requirements.

are the first to formalize this timed-release variant of the original Byzantine consensus problem.
- It formalizes a system model of DBFT SMR (using TPKE as the cryptographic primitive) under partial synchronization, highlighting core issues addressed by DBFT. Our approach addresses distributed consensus challenges in timed-release contexts that existing timed-release mechanisms cannot adequately solve. Since the deferred problem encompasses the interactive consistency conditions of the original Problem, DBFT can easily interleave with existing BFT protocols, with service operations from both parts indiscriminately in terms of the total order.
- It designs the DBFT protocol using PBFT notations, focusing on the modifications and unique challenges. Given the foundational nature of PBFT in BFTs, we believe our solution can be easily extended to most BFT paradigms.
- It implements a DBFT system and experimentally evaluates its performance under various conditions. Our experiments assess both baseline performance when running DBFT exclusively and the impact when interleaving DBFT with standard BFT under different parameter settings. The results demonstrate that existing BFT systems can be feasibly modified to support DBFT for performing timed-release operations periodically without significantly compromising normal operation processing, and provide valuable reference into parameter selection trade-offs when implementing DBFT functionality within existing BFT systems for specific application requirements.

II. Problem Statement

In keeping with the tradition established by Lamport et al. [2] in the original Byzantine Generals Problem, we present our deferred variant using the same military metaphor. This approach allows us to clearly demonstrate how our problem extends the original formulation while maintaining conceptual continuity. After establishing the abstract problem definition, we'll subsequently map it to concrete computer science applications through State Machine Replication in Section IV-B.

We consider a scenario where several divisions of the Byzantine army are encamped outside an enemy city. Each division is commanded by its own general, and the generals can communicate with one another only through messengers. After observing the enemy, they must decide on a common plan of action in advance. Before executing the plan, each general is required to wait for a certain period (which may be long, for example, to complete necessary preparations) or until a specific moment in time. However, some of the generals may be traitors, who, during the waiting period, could send information to the enemy (with the time to inform the enemy being shorter than the waiting period), potentially leaking the plan or causing the enemy to raise its defenses. Additionally, some loyal generals, though involved in deciding on the common plan, may be temporarily recalled during the waiting period and may only return some (uncertain) time after the period has ended. The recalled generals cannot communicate with other generals until they return.

Thus, the plan of action must remain secret: each general, upon receiving the secret plan, can know at most only a partial share of the full plan and whether it is decryptable. Each secret-generating general is a non-essential participant in the subsequent knowledge of his part of the plan. Loyal generals will attempt to know the plan after the waiting period ends. The traitors, however, will still try to know the plan before the waiting period ends, or prevent the loyal generals from knowing a common plan or reaching agreement. The generals must have an algorithm to guarantee that:

(I) All loyal generals will know the same secret plan of action before waiting.

(II) If the secret plan of action is decryptable, all loyal generals (except those temporarily recalled) will have consistent knowledge of the common plan after the waiting period ends. No general should know the plan before the waiting period is over. Otherwise all loyal generals will not enter the waiting period.
The loyal generals will all do what the algorithm says they should, while traitors may do anything they wish. Regardless of what traitors do, the algorithm must ensure Condition I and II is satisfied. For simplification, henceforth, References to loyal generals after the waiting period ends will exclude any that are temporarily recalled.

(III) A small number of traitors or temporarily recalled generals cannot cause loyal generals to know a bad secret plan or plan.

Similar to the original Byzantine Generals Problem [2], we do not attempt to formalize Condition III, as it depends on a precise definition of a "bad plan". Instead, we consider how the generals can know a plan. Before waiting, each general observe the secret plans he receives and communicates his observations to the others. Let $w(i)$ be the information communicated by general i. The set $W_i \subseteq \{w(1), \ldots, w(n)\}$ contains observations received by general i, who use some method for converting W_i in to a secret plan. After the waiting period ends (if decryptable), each general communicates the partial share of the plan they know to the other generals. Let $s(i)$ be the partial share communicated by general i, and let n be the number of generals. The set $S = \{s(1), \ldots, s(n)\}$ contains all partial shares communicated by the generals. General i, using some method, is able to reconstruct the plan s after obtaining a subset $S_i \subseteq S$. Condition I is achieved by having all generals use the same method for transferring the observations and Condition II is achieved by ensuring all generals use the same method and follow the same rules to combine the partial shares. Condition III requires robustness of the methods. Note that this approach may not be the only way to satisfy conditions I-III, it is just the one we know.

We omit the conditions that if the generals need to make a decision (such as to attack or retreat) after knowing the plan, the conditions to be satisfied for reaching agreement subsequently, as it is identical to the original Problem. As a variant, the Deferred Byzantine Generals Problem builds upon the original problem. Since we use similar formulations, if the subsequent decision is necessary, the above can be considered one half, while the original problem forms the other

half, which together define the complete Deferred Byzantine Generals Problem. In other words, our focus will be on the deferred aspect of the problem.

To satisfy Condition I-II, the following must hold:

1. Every loyal general must obtain some consistent observations in W before waiting.
2. If decryptable, any two loyal generals i, j combine the same plan s using S_i, S_j after the waiting period ends.

In Condition 1-2, a general may not necessarily use the plan share $s(i)$ (observation $w(i)$) obtained directly from general i when reconstructing the plan (converting the secret), as a traitorous general may send different or incorrect shares (observations) to different generals. To satisfy Condition III, we cannot permit that even if general i is loyal, the generals may use a plan share $s(i)$ (observation $w(i)$) different from the one send by general i which may be not usable in the reconstruction of s (may lead to other secret). Therefore, we require the following for each i:

3. If general i is loyal, the observation he sends must be used by every loyal general as the value $w(i)$.
4. If general i is loyal, the plan share he sends must can be used in the reconstruction of s, and must be used by every loyal general as the value $s(i)$ in their combination of the plan.

We first simplify the process of obtaining the secret plan of action by restating the problem as one where a commanding general sends a secret order to his lieutenant generals. This leads to the following problem definition:

Definition 1 (Deferred Byzantine Generals Problem): A commanding general must send a secret order to his $n-1$ lieutenants, which requires a period of waiting, such that:

IC1. All loyal lieutenants know the same secret order.
IC2. If the secret order is decryptable, all loyal lieutenants know the same order after the waiting period. No generals know the order before the waiting period ends. Otherwise all loyal generals do not enter the waiting period.
IC3. If the commanding general is loyal, then each loyal lieutenant knows the order in the secret he sends after the waiting period, even if the commanding general and general involved in generating the secret were recalled during the waiting period.

IC1, IC2, and IC3 serve as the interactive consistency conditions for the Deferred Byzantine Generals Problem. If the commanding general is loyal, then IC1 and IC2 follow from IC3. This simplified problem does not consider who is reponsible for generating the secret order - a separate entity (e.g. a general other than the commanding general and his lieutenants) who generate a secret but does not disclose the order to any general. A loyal commanding general will first verify the secret's decryptability, and will not send it to lieutenants if it is not.

We can also consider the case where the secret order is generated by the commanding general. In this case, IC2 must be split into the following two conditions:

IC2.1 *If the secret order is decryptable, all loyal lieutenants know the same order after the waiting period. Otherwise, no loyal general will enter the waiting period.*
IC2.2 *If the commanding general is loyal, no traitor will know the order before the waiting period ends.*

Since the commanding general generates the secret, he can know the order before the waiting period ends, which seemingly contradicts the second clause of Condition II. However, even if a secret action plan s is decided by all generals (e.g., by combining orders $s_1, s_2, \ldots, s_n$, where s_i is the order from general i), the generals can first communicate secret orders, then communicate observations about the received secrets. After the waiting period ends, the s_i of general i (not $s(i)$) can be knowed using a solution to the Deferred Byzantine Generals Problem, with the other generals acting as lieutenants (thus IC2 is applicable and Condition II is satisfied).

If the generals must obey or make a decision after knowing the order, then it also has:

IC4. *All loyal lieutenants decide on (obey) the same order.*
IC5. *If the commanding general is loyal, then every loyal lieutenant decides on (obeys) the order he sends.*

IC4 follows from IC2 and IC5 follows from IC3. The two conditions can also be seen as directly inheriting the final goal of the original problem.

Since the Deferred Byzantine Generals Problem derived from the original problem, it is feasible to extend the model and setup used in Byzantine Fault Tolerance (BFT) consensus to construct a solution. In the subsequent sections, we will focus on a solution that works for $3m+1$ or more generals to cope with m traitors and temporarily recalled generals. For simplicity, let the total number of generals be $3m+1$. We use the Bracha reliable broadcast paradigm [20] and PBFT [19] notations–two classical primitives used in solving the original problem–to present a Deferred Byzantine Fault Tolerant (DBFT) consensus, which allows for easy extension to other paradigms. We employ Threshold Public Key Encryption (TPKE) as the cryptographic primitives.

III. Preliminaries

A. Cryptographic Primitives

A "k-of-n" Threshold Public Key Encryption (TPKE) [14], [15] distributes the private key across n decryption parties, requiring at least k parties to decrypt a ciphertext. The system involves an entity known as the combiner who holds a ciphertext and collects partial decryption shares from the parties to produce the cleartext. Following the notation from Shoup and Genarro [21], a TPKE scheme consists of five algorithms.

$\mathsf{Setup}(n, k, \Lambda)$: Takes n, k ($1 \le k \le n$), and a security parameter $\Lambda \in \mathbb{Z}$. Outputs $(\mathsf{PK}, \mathsf{VK}, \mathsf{SK})$, where PK is the public key, VK is the verification key, and $\mathsf{SK} = (\mathsf{SK}_1, \ldots, \mathsf{SK}_n)$ is a vector of n private key shares where party i receives (i, SK_i). VK is used to check validity of responses from decryption parties.

$\mathsf{Encrypt}(\mathsf{PK}, M)$: Takes PK and a message M, and outputs a ciphertext.

$\mathsf{ShareDecrypt}(\mathsf{PK}, i, \mathsf{SK}_i, C)$: Takes PK, a ciphertext C, and a private key share SK_i. Outputs a decryption share $\mu = (i, \hat{\mu})$

of the enciphered message if it is well-formed (valid and decryptable) or $(i, \perp)$ otherwise.

ShareVerify(PK, VK, C, μ): Takes PK, VK, ciphertext C, and a decryption share μ. Outputs valid or invalid, indicating if μ is a correct partial decryption of C.

Combine(PK, VK, $C, \{\mu_1, \ldots, \mu_k\}$): Takes PK, VK, ciphertext C, and k decryption shares $\{\mu_1, \ldots, \mu_k\}$. Outputs the cleartext M or $\perp$.

1) Consistency Requirements: Given (PK, VK, SK) from Setup(n, k, Λ), it requires: (1) Any valid decryption share μ = ShareDecrypt(PK, i, SK$_i$, C) must be verified as valid by ShareVerify(PK, VK, C, μ). (2) Any set of k valid decryption shares of C from distinct parties must correctly recover the original message through Combine.

2) Security: TPKE's security relies on two properties:

(1) *Chosen Ciphertext Security.* An adversary corrupting up to $k-1$ parties cannot distinguish between the encryptions of two equal length messages of their choice, given that they can ask for decryption shares of any other ciphertexts.

(2) *Decryption Consistency.* Different sets of k valid decryption shares from distinct parties of a ciphertext will produce the same decryption result.

A TPKE system is secure if both properties hold against polynomial-time adversaries. Ideally, no further interaction is needed between the parties during decryption (i.e., parties locally produce decryption shares that can be publicly combined to recover the cleartext), making the system non-interactive.

We use full non-interactive chosen ciphertext secure (CCA2) TPKE schemes [17], [18] in our solution, which additionally include a collision resistant hash function H and a strong existential unforgeability [22] one-time signature scheme (Gen, Sig, Verify) as part of PK and can be run independently by all parties. The advantage is that it does not require two distinct key systems, although it is also not a mandatory requirement. An alternative solution is to utilize a TPKE scheme in conjunction with a public key signing scheme.

B. Full TPKE Scheme

We give specific algorithms of the full non-interactive CCA2-secure TPKE scheme by Boneh et al. [17] (with minor modifications: the signing and verification processes have been removed from the scheme and incorporated into the inter-replica communication messages within our protocol), which is used by default for the subsequent solution construction. As shown in Fig. 1, the scheme is CCA2-secure assuming the Bilinear Diffie-Hellman assumption [17] holds for $\mathcal{GG}$, Σ is existentially unforgeable [22] and H is collision resistant.

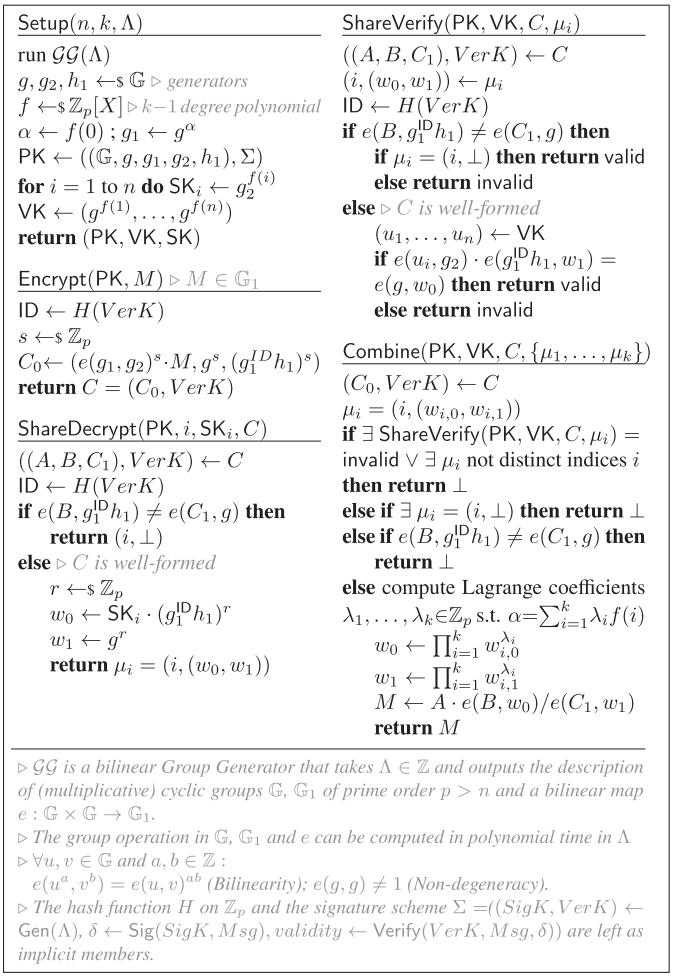


Fig. 1. The full TPKE scheme by Boneh et al. (with minor modifications).

TABLE I
ROLE CROSS-REFERENCE TABLE

Original Role	Role in SMR	
General (+ decryption party)	Replica	
Commanding general	Primary	
Lieutenant general	Backup	
Loyal general (not recalled)	Correct replica (non-faulty)	
Traitor general	Malicious replica	Fault
Loyal general (recalled)	Correct replica (crash)	
Adversary	Adversary	
Secet maker	Client(s) / Replica(s) /...	

*Replicas that crash when waiting correspond to temporarily recalled generals.

IV. MODEL

A. Communication Model

We consider a peer to peer communication and partial synchrony model [23]. There is a known bound Δ and an unknown global stabilization time (GST). After GST, all messages transmitted between loyal generals are arrived within Δ. The traitors or adversary are computationally bound who can eavesdrop on, delay, duplicate communications between loyal generals but lack the ability to impede, alter messages or to subvert the cryptographic primitives. They must cause GST eventually happen after some unknown finite time. The aforementioned model offers support for Conditions 3-4. When refer to "broadcast", it means a general, if loyal, sending the same peer to peer message to all generals, including himself.

B. DBFT SMR

We then transform the roles from the abstract Deferred Byzantine Generals Problem into the State Machine Replication (SMR) form [16], creating a direct mapping to practical distributed systems. In this transformation, generals correspond to replicas in a distributed system. The secret plan of

action represents operations that must remain confidential until a specific time or event occurs, enabling practical applications such as scheduled confidential transactions [9], sealed-bid auctions [13] and global coin tossing [11], etc. (depending on the specific scenario). The service is represented as a state machine replicated across different replicas, where each state machine replica maintains the service state and implements the service operations. We assume a total of $n = 3m + 1$ replicas $R = \{0, 1, \ldots, n\}$, where m is the maximum number of replicas that may be faulty. Each replica $i \in R$ possesses a triple $(\mathsf{PK}, \mathsf{VK}, \mathsf{SK}_i)$ including a public key, a verification key and a private key share, generated through $\mathsf{Setup}(n, 2m+1, \Lambda)$ (different methods can be used to create this setup, e.g., a suitable distributed DKG [24] protocol or a trusted dealer). Within a period referred to as a view v, a replica in R serves as the primary and the others serve as backups. The system functions in a series of views, each with a monotonically increasing number. Table I lists the cross-references for roles.

Due to space limitations, we focus primarily on the construction of a solution to the simplified Deferred Byzantine Generals Problem (as described in Section II), which provides the basis for this class of problems. In this discussion, we omit the details of who generates the secret, which could be state machine operations requested by a replica or client (clients) (not restricted to transactions, reconfiguration operations, reads/writes of portions of service state, etc.), a secret request generated by a primary, or even a globally deferred operation decided by all replicas. Therefore, we omit the client from the discussion in most parts, and continue to use the term "secret maker" to denote the entity responsible for generating the secret. We also omit the discussion of how the waiting period is determined, such as whether it is derived from specific reference rules, or based on an event agreed upon in a previous view.[2] You may also refer to the design of timed-release cryptographic primitives [5], [7], [8], [10], [25] to design the two mechanisms above. Additionally, we leave out discussions of replica crashes and how they recover, as well as how primary is elected (default having a unique dedicated primary known to all in each view). In summary, these aspects are not central to the DBFT SMR problem and can be adapted based on different scenarios.

It is worth noting that DBFT can also be interleaved within BFT to perform tasks specific to a particular view. Since it can achieve the final objective of the original problem, the service operations of the two parts can be indiscriminate in terms of the total order.

V. Algorithm

We now present the algorithm, focusing on the DBFT SMR problem. It ensures the first sentence of IC2 by guaranteeing that all non-faulty replicas agree on the same operations in secrets with the same sequence numbers, each is known (or commit locally) after the waiting period. Replicas move through a succession of views, each with a monotonically increasing number and has a unique dedicated primary known to all. View changes are carried out when it appears that the primary has failed. Throughout this section, we assume each message *msg* is unique.

Before presenting the detailed algorithm, we first explain the key design intuition behind our DBFT protocol. The Deferred Byzantine Generals Problem introduces unique challenges beyond those in the original problem, particularly related to the timed-release aspect and ensuring consistency across all non-faulty replicas when learning secret operations. Our design is guided by four key principles:

- Separating the consensus on the existence and ordering of secret operations from the actual learning of them, which ensures all non-faulty replicas agree on which secrets to process and in what order before any secret is learned.
- Addressing IC1-IC3 through a multi-phase consensus protocol. Since we design our protocol using PBFT [19] notations, we similarly divide the protocol into two parts: normal-case operation and view change mechanism. The former functions before timeout and allows us to:
  (1) Establish agreement on the secret's existence and sequence number.
  (2) Ensure consistent knowledge of which secrets are prepared.
  (3) Coordinate the knowing of operations only after the waiting period has elapsed.

  The latter is triggered when timeout occurs to ensure progress when the primary fails, while preserving consistency across views.
- Preventing, to the greatest extent possible, the scenario where malicious replicas could know the operation after the waiting period within a view, while non-faulty replicas cannot, even if non-faulty replicas would eventually know the operation after a view change. Note such a scenario may not be regarded as a vulnerability in standard BFTs.
- Ensuring timed-release through voting interactions.

The following subsections detail each component of our solution.

A. Secret Maker

Fig. 2 presents the pseudocode for how a secret maker submits a secret operation. The secret maker first encrypts the operation *op* using $\mathsf{Encrypt}$ and generates a signature δ for the secret. It then submits *op* by sending the message $\langle \text{SUBMIT}, C_0, sid, VerK, \sigma \rangle$ to what it believes is the current primary, and starts a timer. This process works when $op \in \mathbb{G}_1$. Since the bilinear map e is one-way, it is impossible to map $op \notin \mathbb{G}_1$ into $\mathbb{G}_1$ and subsequently recover it. However, there are additional mechanisms that can be employed to adapt any message type for *op*, such as Key Wrap or Key Encapsulation Mechanism [26].

Here, we provide an example using a symmetric encryption algorithm (such as AES [27], which has minimal impact on overall performance due to its efficiency), as indicated in the

[2]Note that DBFT can be extended to the "schedule and reveal" paradigm, in which case it would have explicit schedule or reveal processes. For example, there could be an independent phase where replicas collectively schedule an encrypted operation they agree on. The period before the scheduled time would constitute the waiting period. Alternatively, the secret maker (or a specific designated revealer) could decide when to reveal (e.g., by broadcasting corresponding messages). Non-faulty replicas would only attempt to decrypt and execute the operation upon later receiving valid reveal instructions.

Initialization:
v, sid ▷ *view number, id of the secret maker*
PK ▷ *public key:* $((G, g, g_1, g_2, h_1), \Sigma)$
$(SigK, VerK) \leftarrow \mathsf{Gen}(\Lambda)$ ▷ *signing key, signature verification key*

1: **function** *submit*(op)
2: $(C_0, VerK) \leftarrow \mathsf{Encrypt}(\mathsf{PK}, op)$ ▷ $op \in \mathbb{G}_1$
3: $\sigma \leftarrow \mathsf{Sign}(SigK, (C_0, sid))$
4: $msg \leftarrow \langle \textsc{submit}, C_0, sid, VerK, \sigma \rangle$
▷ *if* $op \notin \mathbb{G}_1$, *some additional mechanisms will be needed, e.g.:*
5: // $r \leftarrow_\$ \mathbb{Z}_p$, $K \leftarrow e(g_1, g_2)^r$ ▷ $K \in \mathbb{G}_1$
6: // $K_{sym} \leftarrow H(K)$ ▷ *use* H *to convert* K *to a symmetric key*
7: // $C \leftarrow SymEncrypt(K_{sym}, op)$ ▷ *symmetric encryption*
8: // $(C_0, VerK) \leftarrow \mathsf{Encrypt}(\mathsf{PK}, K)$
9: // $\sigma \leftarrow \mathsf{Sign}(SigK, (C, C_0, sid))$
10: // $msg \leftarrow \langle \textsc{submit}, (C, C_0), sid, VerK, \sigma \rangle$
11: send msg to $\textsc{primary}(v)$, start timer Δ
12: **upon** receiving replies from $m+1$ different replicas: $MSG \leftarrow \{\langle \textsc{reply}, op, rst, i, VerK, \sigma \rangle \mid \mathsf{verify}(VerK, (op, rst, i), \sigma)\}$ **do**
13: stop Δ, complete op and **return** ▷ rst *is result of the operation*
14: **upon** $timeout(\Delta)$ **do**
15: broadcast msg, reset Δ

Fig. 2. Pseudocode for a secret maker submitting secret operations.

pseudocode's comments (Line 5-10). First, a random r is chosen to compute an element K in $\mathbb{G}_1$, which is then transformed into a symmetric key K_{sym} using the hash function H. The op is then symmetrically encrypted with K_{sym}, resulting in an additional ciphertext C. The secret maker then encrypts K using Encrypt. At this point, the message sent by the secret maker becomes $\langle \text{SUBMIT}, (C, C_0), sid, VerK, \sigma \rangle$. Once the replicas know K through DBFT consensus, they can use the symmetric decryption algorithm $SymDecrypt(H(K), C)$ to recover op. For simplicity, we will not discuss this case separately in the subsequent pseudocode.

The secret maker then waits for $m+1$ valid replies (since at most m are faulty) from distinct replicas to complete op. If it does not receive enough replies before timeout, it broadcasts the message and resets the timer, repeating the process until the operation is completed. In this paper, we assume the secret maker waits for the one operation to complete before submitting the next one. However, we can allow the secret maker to issue asynchronous operations. The primary can create a queue to temporarily store the operations and then process them sequentially, or pack them into a batch and broadcast, which replicas could process in parallel without violating the order constraints on them.

B. Normal-Case Operation

The pseudocode for replica i's normal-case operation is shown in Fig. 3. Each replica maintains a message log that stores all messages it has accepted. Upon receiving a message from the secret maker containing the secret operation C_0, and assuming the signature is valid, the replica behaves as follows: if the operation has been already completed (known or locally committed), the replica simply re-sends the pertinent reply (replicas remember the last reply message they sent to each secret maker); otherwise, if the replica is not the primary, it forwards the message to the primary.

When the primary receives the secret C_0, it first checks whether it is well-formed. If so, it immediately initiates the consensus protocol, which consists of four phases:

Initialization:
v, i, log ▷ *view number, replica number, message log*
$seq_{min}, seq_{max}, freq$ ▷ *low, high water mark, checkpoint frequency*
$\mathsf{PK}, \mathsf{VK}, \mathsf{SK}_i$ ▷ *public key, verification key, private key share*
$(SigK, VerK) \leftarrow \mathsf{Gen}(\Lambda)$ ▷ *signing key, signature verification key*
/*the process of adding messages to log is omitted from the pseudocode*/
1: **upon** receiving $msg = \langle \textsc{submit}, C_0, sid, VerK, \sigma \rangle$ from sid and $\mathsf{verify}(VerK, (C_0, sid), \sigma)$ **do**
2: **if** $\exists v'$ and seq' such that $known(v', seq', C_0, op) \leftarrow true$ **then**
3: resend the reply to C_0 to sid /*Non-faulty replicas ignore undecryptable
4: **else if** $\textsc{primary}(v)$=i **then** C_0 with sources identified as malicious (Line
5: **if** $\mathsf{ShareDecrypt}(\mathsf{PK}, i, \mathsf{SK}_i, (C_0, VerK)) \neq (i, \perp)$ **then** 5,9)*/
6: $seq \leftarrow assign(C_0)$ ▷ *assign a sequence number*
/*since C_0 can be generated using only PK, if to ensure source or prevent forgery, σ and sid can be attached to the pre-prepare message*/
7: broadcast $\langle \textsc{pre-prepare}, v, seq, C_0, \mathsf{Sign}(SigK, (v, seq, C_0)) \rangle$
8: **else** send msg to $\textsc{primary}(v)$ ▷ *forward* msg *to the current primary*
9: **upon** receiving $msg = \langle \textsc{pre-prepare}, v, seq, C_0, VerK, \sigma \rangle$ from $\textsc{primary}(v)$ such that $(\mathsf{verify}(VerK, (v, seq, C_0), \sigma)) \wedge$ (no PRE-PREPARE message has been accepted for v and seq) $\wedge$ $(\mu_i \leftarrow \mathsf{ShareDecrypt}(\mathsf{PK}, i, \mathsf{SK}_i, (C_0, VerK)), \mu_i \neq (i, \perp)) \wedge$ $(seq_{min} \leq seq \leq seq_{max})$ **do** ▷ *accept the pre-prepare message*
10: $\sigma_{prepare} \leftarrow \mathsf{Sign}(SigK, (v, seq, H(C_0), i))$
11: broadcast $\langle \textsc{prepare}, v, seq, H(C_0), i, VerK, \sigma_{prepare} \rangle$
/*sequence number checking is omitted in the subsequent pseudocode*/
12: **upon** receiving $2m+1$ matching $\langle \textsc{prepare}, v, seq, H(C_0), j, VerK, \sigma \rangle$ msg_j each with $\mathsf{verify}(VerK, (v, seq, H(C_0), j), \sigma)$ from j **do**
13: $prepared(v, seq, C_0) \leftarrow true$
14: $PCert_{seq} \leftarrow \{msg_j\}_{j \in R}$ ▷ *prepare certificate*
15: $\sigma_p \leftarrow \mathsf{Sign}(SigK, (v, seq, H(C_0), i, PCert_{seq}))$
16: broadcast $\langle \langle \textsc{pre-combine}, v, seq, H(C_0), i, PCert_{seq}, VerK, \sigma_p \rangle, C_0 \rangle$
17: $event(C_0).await$ ▷ *wait for the event associated with* C_0 *to complete*
18: $\sigma_{share} \leftarrow \mathsf{Sign}(SigK, (v, seq, \mu_i, H(C_0))$
19: broadcast $\langle \textsc{combine}, v, seq, \mu_i, H(C_0), VerK, \sigma_{share} \rangle$
20: **upon** receiving $\langle \langle \textsc{pre-combine}, v, seq, H(C_0), j, PCert_{seq}, VerK, \sigma \rangle, C_0 \rangle$ such that $\mathsf{verify}(VerK, (v, seq, H(C_0), j, PCert_{seq}), \sigma)$ from j **do**
21: **if** $(\nexists C_0'$ such that $prepared(v, seq, C_0') = true) \wedge (PCert_{seq}$ is valid) **then** $prepared(v, seq, C_0) \leftarrow true$
22: save $PCert_{seq}$ as its own
23: $\sigma_p \leftarrow \mathsf{Sign}(SigK, (v, seq, H(C_0), i, PCert_{seq}))$
24: broadcast $\langle \langle \textsc{pre-combine}, v, seq, H(C_0), i, PCert_{seq}, VerK, \sigma_p \rangle, C_0 \rangle$
25: $event(C_0).await$ ▷ *wait for the event to complete*
26: $\mu_i \leftarrow \mathsf{ShareDecrypt}(\mathsf{PK}, i, \mathsf{SK}_i, (C_0, VerK))$
27: $\sigma_{share} \leftarrow \mathsf{Sign}(SigK, (v, seq, \mu_i, H(C_0))$
28: broadcast $\langle \textsc{combine}, v, seq, \mu_i, H(C_0), VerK, \sigma_{share} \rangle$
▷ *matching means matching with* $prepared(v, seq, C_0)$*here*
29: **upon** receiving $2m+1$ matching $\langle \textsc{combine}, v, seq, \mu_j, H(C_0), VerK, \sigma \rangle$ msg_j each with $(\mathsf{verify}(VerK, (v, seq, \mu_j, H(C_0)), \sigma)$ from $j) \wedge$ $(\mathsf{ShareVerify}(\mathsf{PK}, \mathsf{VK}, (C_0, VerK), \mu_j)$ **do** $shares \leftarrow \{\mu_j\}_{j \in R})$
30: $op \leftarrow \mathsf{Combine}(\mathsf{PK}, \mathsf{VK}, (C_0, VerK), shares)$
31: $rst \leftarrow$ /*further actions after knowing op if necessary*/
32: $known(v, seq, C_0, op) \leftarrow true$
33: send $\langle \textsc{reply}, op, rst, i, VerK, \mathsf{Sign}(SigK, (op, rst, i)) \rangle$ to sid
34: **if** $seq \% freq = 0$ **then** ▷ *handling the checkpoint*
35: **if** $CCert.seq < \forall sq \leq seq$ such that $prepared(\exists v^*, sq, \exists C_0^*) = true$, $\exists v''$ and op^* to have $known(v'', sq, C_0^*, op^*) \leftarrow true$ **then**
36: broadcast $\langle \textsc{checkpoint}, seq, i, VerK, \mathsf{Sign}(SigK, (seq, i)) \rangle$
37: **upon** receiving $2m+1$ matching $\langle \textsc{checkpoint}, seq, j, VerK, \sigma \rangle$ msg_j each with $\mathsf{verify}(VerK, (seq, j), \sigma)$ from j **do** ▷ *stable checkpoint*
38: $seq_{max} \leftarrow seq + seq_{max} - seq_{min}, seq_{min} \leftarrow seq$
39: $CCert \leftarrow (seq, \{msg_j\}_{j \in R})$ ▷ *checkpoint certificate*
40: $log.retain(msg.seq_{msg \in log} > seq)$ ▷ *discard* msg *with* $seq' \leq seq$

Fig. 3. Normal-case operation at replica i.

PRE-PREPARE, PREPARE, PRE-COMBINE, and COMBINE. The PRE-COMBINE phase can be considered an extension of the PREPARE phase, effectively reducing the protocol to three phases. As a result, the protocol can be viewed in terms of the Bracha broadcast paradigm [20] or using PBFT terminology [19]. The first three phases ensure that non-faulty

replicas agree on the same secrets with the same sequence numbers within the same view, establishing consistency before decryption (IC1). The COMBINE phase ensures consistency in the decryption result. Except for PRE-PREPARE, the remaining three phases also construct predicates or certificates to prevent conflicts across views and ensure consistency.

In the PRE-PREPARE phase, the primary assigns a sequence number *seq* to C_0, and packages it along with the current view v, C_0, and the signature into a pre-prepare message, which is then broadcast. If C_0 is too large, it may be decoupled from the pre-prepare message to reduce communication costs, as the pre-prepare message in the log may be re-engage in communication during view-change.

Upon receiving the pre-prepare message, replicas perform the following validations: (1) whether the signature is valid, (2) whether the secret is decryptable, as a malicious primary might broadcast a non-well-formed secret, (3) whether a pre-prepare message has already been accepted for the current sequence number and view, and (4) whether the sequence number is within the low and high watermarks, because a malicious primary might try to force the sequence number to grow rapidly by choosing an excessively large one, leading to space of sequence number exhaustion.[3] If the validation passes, the replica accepts the pre-prepare message and enters the PREPARE phase.

Once the replica has knowledge of C_0, it can use the hash of C_0 in subsequent communications to reduce the size of messages. In the PREPARE phase, the replica packages *seq*, v, $H(C_0)$, its own number, and the signature into a prepare message, which is broadcast to all other replicas. Upon receiving a prepare message, the replica accepts it if it is in the current view and has a valid signature. After accepting $2m+1$ prepare messages from distinct replicas for the same *seq*, v, and $H(C_0)$ matching the pre-prepare message, the replica packages them into a prepare certificate and sets the predicate $prepared(v, seq, C_0)$ to true.

If $prepared(v, seq, C_0) = true$ for any non-faulty replica, it means at least $m + 1$ non-faulty replicas have sent prepare messages for C_0. Since at most m replicas are malicious, it is impossible for another non-faulty replica to have $prepared(v, seq, C'_0) = true$ for $C'_0 \neq C_0$, as this would require one of the non-faulty replicas sending a prepare message for C_0 to also send a prepare message for C'_0 with the same sequence number and view, which is not possible.

Once the replica set $prepared(v, seq, C_0)$ to true in the PREPARE phase, it broadcasts a pre-combine message, which, in addition to the prepare message, includes the matching prepare certificate to prove that the replica's $prepared(v, seq, C_0) = true$. This informs other non-faulty replicas that, within v and *seq*, no other secret C'_0 can have $prepared(v, seq, C'_0) = true$, thus ensuring uniform knowledge of the secret across non-faulty replicas. The replica then proceeds to execute the event associated with C_0 ($event(C_0)$) and waits for it to complete. For any received pre-combine message with a valid signature, if the replica does not already have the *prepared* predicate set to true for v and *seq*, and if the certificate is valid (the validation process is similar to that for prepare messages, Line 9), the replica accepts C_0 and sets $prepared(v, seq, C_0)$ to true. It accepts the received certificate as its own, also broadcasts a corresponding pre-combine message, then similarly executes $event(C_0)$ and waits for it to complete.

[3]Replica will not consider the acceptance of any message with a sequence number outside of the high and low water marks, and we omit this in the subsequent pseudocode and description for the sake of brevity.

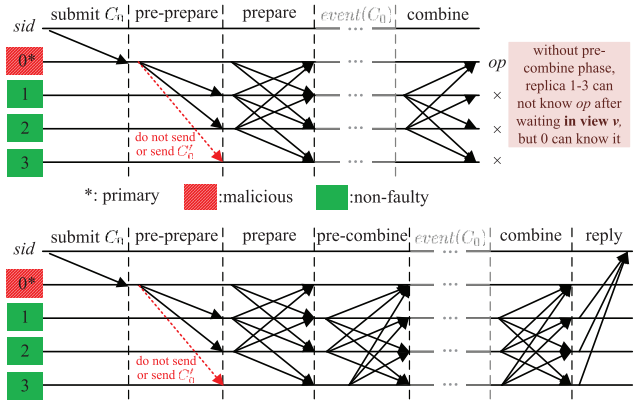


Fig. 4. Scenario (under v and *seq*) illustrating the significance of PRE-COMBINE phase in normal-case operation (assuming C_0 can be completed in view v).

The significance of the PRE-COMBINE phase lies in ensuring, to the greatest extent possible, within a given view, no situation arises where malicious replicas know the operation while non-faulty replicas do not after the waiting period. Consider a scenario where the primary is malicious and, within v, selects a valid *seq* but only sends the pre-prepare message containing C_0 to $m+1$ non-faulty replicas and m malicious replicas (including itself), while proceeding normally in the PREPARE phase. Without the PRE-COMBINE phase, the remaining m non-faulty replicas, which did not receive C_0, would not broadcast their decryption shares in the subsequent COMBINE phase. If the m malicious replicas do not send their shares to the $m + 1$ non-faulty replicas that completed $event(C_0)$, none of the non-faulty replicas would know the operation *op* after the waiting period ends, while the malicious replicas would know it. This is visually described in Fig. 4.

The prepare certificate proves the replicas included within the certificate sent prepare messages for C_0 under v and *seq*, with at least $m + 1$ of them being non-faulty, matching the message they accepted in the PRE-PREPARE phase. The PRE-COMBINE phase ensures when any non-faulty replica has the $prepared(v, seq, C_0)$ predicate set to true under v and *seq*, all non-faulty replicas (even those that accepted conflicting pre-prepare messages or didn't receive pre-prepare messages under v and *seq*, since a malicious primary can send conflicting pre-prepare messages to different replicas) will be able to have the same prepared predicate set to true under v and *seq*. Since the prepared predicate can be set to true in either the PREPARE or PRE-COMBINE phase in the normal-case operation, non-faulty replicas need to broadcast a corresponding pre-combine message whenever they set this predicate to true (but the algorithm ensures non-faulty replicas won't broadcast more than once for the same v and *seq* in a single view).

A similar phase is not required for PBFT because a malicious replica having an early local commitment does not have

significant affections. If a similar situation occurs in PBFT, the request for *seq* that non-faulty replicas prepared but failed to commit is simply reconsensused after the next view change, and the non-faulty replicas will complete it if the primary of the new view is non-faulty. However, in DBFT, we generally assume the situation is more serious and should be avoided to the greatest extent possible where malicious replicas can know the operation after the waiting period within a view while non-faulty replicas cannot, even if non-faulty replicas would eventually know the operation after a view change. If it can be determined that such a situation poses no potential risk in a specific scenario, eliminating the PRE-COMBINE phase could be a viable option.

Once the replica completes $event(C_0)$, it packages *seq*, v, $H(C_0)$, its own decryption share μ_i, and its signature into a combine message, which it then broadcasts, entering the COMBINE phase. For any received combine message in the current view, if the signature is valid and the decryption share passes ShareVerify, the message is accepted. After accepting $2m + 1$ combine messages from distinct replicas for C_0, the replica performs the decryption and recovers *op*, setting the predicate $known(v, seq, C_0, op)$ to true. If a replica has $known(v, seq, C_0, op) = true$, it implies that at least $m + 1$ non-faulty replicas have $prepared(v, seq, C_0) = true$ and have completed $event(C_0)$. We treat this as the sign that the waiting period has ended. Since the primary has the ability to assign sequence numbers to operations, the predicates and certificates are used to ensure that after a view change, the new primary cannot assign an already-used sequence number to an operation (whether intentionally or through malicious replicas' interference) thus preventing conflicts, and coping with the case of completing an operation across views. This ensures IC2, which we will discuss further in the next subsection on the view change protocol.

Finally, the replica send a reply to *sid* containing, *op*, a possible result *rst* of *op*, its own number and the signature.

1) Garbage Collection: Garbage collection is used to periodically discard unnecessary messages from the message log to reduce storage overhead. We adopt a checkpoint mechanism similar to that in PBFT. Any accepted message must be stored in the replica's log until its corresponding operation has been completed and an appropriate certificate has been constructed to prove this to a new primary during a view change. Specifically, we consider a preset constant checkpoint frequency *freq*. Whenever a replica completes an operation with a sequence number *seq* that is a multiple of *freq*, it enters the checkpoint mechanism. For all sequence numbers *sq* between the latest stable checkpoint and *seq*, if there is a *prepared* predicate set to true for each *sq*, and each *prepared* predicate can be matched with a *known* predicate set to true, then a checkpoint is generated for *seq*. The replica packages *seq*, its replica number and signature into a checkpoint message and broadcasts it.

When a replica accepts $2m+1$ valid checkpoint messages for the same *seq* from distinct replicas, it packages these messages into the new checkpoint certificate, and *seq* becomes the latest stable checkpoint. The replica then discards all messages in the log with sequence numbers less than or equal to the stable checkpoint.

Note that our checkpoint messages do not include data reflecting the replica's state. If the operation has been completed or a reply has been sent to be reflected in the state, the state hash may also be included in the checkpoint message, similar to PBFT. In this case, the replica only needs to collect $m + 1$ valid checkpoint messages (PBFT's OSDI version [19] requires $2m + 1$ for this process, as it is in MAC mode rather than a digital signature mode) from different replicas with the same state hash and sequence number to establish a stable checkpoint and generate a corresponding checkpoint certificate. Since the senders of these messages cannot deny their possession of the same checkpoint (proven by the signature and state hash), which confirms that the operation has been completed or the replies have been sent, and at least one message must come from a non-faulty replica, proving that the state is reachable.

Replicas use the latest stable checkpoint to update the watermarks. The low watermark is equal to the latest stable checkpoint. The gap between the high and low watermarks can be adjusted based on requirements, balancing the storage overhead of maintaining the message log and the communication overhead of broadcasting checkpoint messages. For example, it can be set to a constant greater than the checkpoint frequency.

C. View Change

Generally speaking, (partially) synchronous fault tolerant protocols rely on view changes to make progress, which are triggered either when consensus can't make further progress (e.g. a primary fails) [19], [28] or periodically according to some strategies [29], [30]. We follow a paradigm similar to the former. In DBFT, the view change protocol allows the system to switch the primary and view when the current primary fails, preventing replicas from waiting indefinitely and enabling consensus to continue, thereby ensuring IC3 eventually holds to provide liveness. During normal-case operation, each replica maintains a timer, which is reset after completing an operation (secret). If replica i's timer expires in view v for the *num*th consecutive time or it receives a set R_1 of $m + 1$ valid view-change messages from other replicas for greater views, it enters the view change protocol to transition the system to $v+num$ or the smallest view in R_1, as shown in the pseudocode in Fig. 5. The replica first resets its timer. It stops accepting all messages from normal-case operation phases (except for checkpoint messages, which are still processed). The replica then iterates through its message log, creating the sets *PP* and *Pset*. *PP* contains all accepted pre-prepare messages, while *Pset* is a set of prepare certificates for all sequence numbers *seq* where a corresponding *prepared* predicate is true, but no matching *known* predicate is true. The replica packages the target view number, the current *CCert* (checkpoint certificate), *PP*, *Pset*, its replica number, and its signature into a view-change message and broadcasts it.

When the new primary receives $2m + 1$ valid view-change messages, it packages these messages into a new view-change certificate and selects the highest sequence number from all valid checkpoint certificates included in the messages as *seq*. Based on *seq*, the primary creates a set *Oset*, which includes

Initialization:
v, i, log ▷ *view number, replica number, message log*
$seq_{min}, seq_{max}, CCert$ ▷ *low, high water mark, checkpoint certificate*
$\mathsf{PK}, \mathsf{VK}, \mathsf{SK}_i$ ▷ *public key, verification key, private key share*
$(SigK, VerK) \leftarrow \mathsf{Gen}(\Lambda)$ ▷ *signing key, signature verification key*
▷ *Certificate validation is similar to validating its containment messages*
1: **upon** timer timeout (case 1) or receiving $m+1$ ▷ *a PRIMARY is also a replica* $\langle$VIEW-CHANGE, $v_j \geq v, CCert, Pset, PP, j, \sigma\rangle$, each with $\mathsf{verify}(VerK, (v_j, CCert, Pset, PP, j), \sigma)\}$ from j (case 2) **do**
2: **as** a replica ▷ *num is the number of consecutive timeouts in v.*
3: $v_c \leftarrow v + num$ (case 1) or $v_c \leftarrow argmin\{v_j\}$ (case 2), reset timer
4: stop accepting phase messages from normal-case operation
5: $PP \leftarrow \{\langle msg \mid (msg \in log) \wedge (msg.type = \text{PRE-PREPARE})\rangle\}$
6: $Pset \leftarrow \{PCert_{seq} \mid prepared(\exists v^*, seq, \exists C_0^*) = true,\ \nexists v''$ and op^* to have $known(v'', seq, C_0^*, op^*) = true\}$
7: $\sigma_{vc} \leftarrow \mathsf{Sign}(SigK, (v_c, CCert, Pset, PP, i))$
8: broadcast $\langle$VIEW-CHANGE, $v_c, CCert, Pset, PP, i, \sigma_{vc}\rangle$
9: **upon** receiving $\langle$NEW-VIEW, $v', VCert, Oset, \sigma\rangle$ from PRIMARY(v') with $(\mathsf{verify}(VerK, (v', VCert, Oset), \sigma)) \wedge (VCert$ is valid) **do**
10: **if** $v' > v, v \leftarrow v'$ **then** ▷ *update the latest stable checkpoint*
11: $CCert \leftarrow (\ argmax_{(msg_j \in VCert) \wedge (msg_j.CCert \text{ is valid})}\{msg_j.CCert.seq\}\ ).CCert$
12: $seq_{max} \leftarrow seq + seq_{max} - seq_{min}, seq_{min} \leftarrow seq$
13: **if** $Oset$ is validated using $VCert$ **then**
14: **for** $msg \in Oset$, set the PRE-PREPARE message for v and $msg.seq$ has been accepted, **If** $prepared(\exists v^*, msg.seq, msg.C_0)$ **do**
15: **if** $(known(\exists v'', msg.seq, msg.C_0, \exists op^*)) \vee (event(msg.C_0)$ is completed) **then** broadcast $\langle$COMBINE, $v, msg.seq, \mu_i, H(msg.C_0), VerK, \sigma_{share}\rangle$ /*$\mu_i, \sigma_{share}, \sigma_p \leftarrow$ identical to Figure 3, lines 24,15,18*/
16: **else** broadcast$\langle\langle$PRE-COMBINE, $v, msg.seq, H(msg.C_0)$ $i, PCert_{msg.seq}, VerK, \sigma_p\rangle, msg.C_0\rangle$
17: $prepared(v, msg.seq, msg.C_0) \leftarrow true$
18: return to normal-case operation
19: **as** PRIMARY($v' > v$) and **upon** receiving $2m+1$ matching $\langle$VIEW-CHANGE, $v', CCert, Pset, PP, j, \sigma\rangle\ msg_j$, each with $\mathsf{verify}(VerK, (v', CCert, Pset, PP, j), \sigma)\}$ from j
20: $VCert_{v'} \leftarrow \{msg_j\}_{j \in R}$ ▷ *view-change certificate*
21: $seq \leftarrow (argmax_{(msg_j.CCert \text{ is valid})}\{msg_j.CCert.seq\}).seq$ ▷ *latest stable checkpoint*
22: $Oset \leftarrow \{msg \mid (\exists msg_j, msg \in msg_j.PP) \wedge (msg.seq > seq) \wedge (\exists msg'_j, PCert \in msg'_j.Pset$ such that ($PCert$ is valid $\wedge$ ($PCert$ is for $msg.C_0$ under $msg.seq$))$\}$
23: $\sigma_{nv} \leftarrow \mathsf{Sign}(SigK, (v', VCert_{v'}, Oset)$
24: broadcast $\langle$NEW-VIEW, $v', VCert_{v'}, Oset, \sigma_{nv}\rangle$

Fig. 5. View change protocol at replica *i*.

a unique set of pre-prepare messages with the following properties: (1) the sequence number is greater than *seq*, (2) at least one valid view-change message's *PP* set contains the message, and (3) at least one valid view-change message's *Pset* contains a matching prepare certificate. The new primary then packages the new view number, the view-change certificate, *Oset*, and its signature into a new-view message and broadcasts it.

Upon receiving a new-view message, if it is from the primary of the new view and contains a valid view-change certificate and signature (the validation of the certificate is similar to validating the messages it contains), the replica accepts it and enter the new view. The replica then updates its *CCert* and watermarks using the highest sequence number from the valid checkpoint certificates in the view-change certificate (similar to the process in normal-case operation). If *Oset* is valid (the replica can verify this by performing the same computation as the primary when creating the *Oset*), for each pre-prepare message *msg* in the *Oset*, the replica first sets the PRE-PREPARE message for the new view and *msg.seq* has been accepted, and then checks whether it has a corresponding *prepared* predicate to be true and a corresponding *known* predicate to be true, and based on this, decides whether to broadcast a matching combine or pre-combine message for the new view. The replica then sets the matching prepare predicate to true in the new view and returns to normal-case operation after traversing *Oset*.

The view change protocol prevents conflicts after a view change, ensuring that if a non-faulty replica has completed the secret C_0 (or operation *op*) at sequence number *seq*, no other non-faulty replica will complete C_0 at a different *seq'*. Regardless of how malicious replicas interfere, the new primary will always be able to obtain the latest stable checkpoint and all pre-prepare messages after that checkpoint from at least $m+1$ non-faulty replicas. The worst-case scenario is that some already completed operations will need to have their PRE-COMBINE or COMBINE phases re-executed. This re-execution mechanism is necessary to ensure cross-view consistency in decryption, as we cannot guarantee secrets which have been successfully prepared (i.e., the corresponding *prepared* predicate is true) will always be completed by non-faulty replicas before timeout. For example, due to network partitions or malicious interference, *event*(C_0) might not be completed before the timer expires. The replica can continue the PRE-COMBINE or COMBINE phase of C_0 after the view change protocol completes.

Replicas missing logs or state can also retrieve the missing information after the view change. The pseudocode explicitly outlines the process by which replicas missing a stable checkpoint state can use the view-change certificate to update to the latest stable checkpoint (Line 11-12), and other missing information can be obtained in a similar manner. Even if the new primary fails as well, replicas can initiate another view change after next timeout.

1) Complexity: In the normal-case operation, the protocol involves four phases. In the PRE-PREPARE phase, the primary broadcasts to all n replicas. In the PREPARE phase, each replica broadcasts to all other replicas, resulting in $O(n^2)$ messages, which is repeated in the PRE-COMBINE and COMBINE phases. Thus, the overall communication complexity in the normal case is $O(n^2)$ per consensus instance. When timeout occurs, replicas broadcast view-change messages. After receiving $2m+1$ valid view-change messages, the new primary broadcasts a new-view message, followed by the possible re-execution of the PRE-COMBINE or COMBINE phases, which also incur $O(n^2)$ complexity. Both the normal-case operation and view-change protocol have a communication complexity of $O(n^2)$. The latter adds overhead due to state synchronization, but the overall complexity remains quadratic in both phases.

Our solution's correctness is shown in Appendix A in the supplementary material.

VI. Implementation and Evaluation

A. Implementation

We implemented DBFT on top of an existing production-ready Rust-based PBFT codebase (since the Deferred Byzantine Generals Problem is derived from the original problem,

it is recommended to extend existing BFT systems to construct DBFT solutions), which uses the tokio library for asynchronous IO and TCP connections for communication between replicas. The core consensus logic of DBFT consists of approximately 800 lines of code. The modified system can either run the DBFT protocol exclusively or interleave DBFT with the original PBFT protocol (i.e., only using DBFT for consensus on secret operations at specific sequence numbers). The client, also serving as the secret maker, is implemented as a separate single-threaded process that sends (secret) operations to the replicas at a specified rate.

We used the bls12_381_plus, ed25519_dalek, and aes_gcm libraries to implement cryptographic primitives, totaling around 400 lines of code. This includes the full non-interactive CCA2-secure TPKE scheme and symmetric encryption/decryption (used to allow the Secret Maker to encrypt arbitrary operation types as secret operations, as detailed in Section V-A). These cryptographic operations replaced the original cryptographic modules in the codebase.

Since, to the best of our knowledge, the Deferred Byzantine Generals Problem and DBFT protocol has not been previously discussed in the literature, and there are no existing BFT protocols with similar timed-release properties that could serve as direct comparisons, our experiments instead follow a two-pronged approach: (1) We first assess the baseline performance and scalability of DBFT running exclusively under various network conditions to comprehensively demonstrate its fundamental characteristics. (2) We then measure the performance impact when DBFT is interleaved with the PBFT protocol under different secret operation frequencies and time event lengths. This set of experiments serves two purposes: first, to demonstrate that existing BFT systems can be feasibly modified to support DBFT for performing specific tasks (e.g., timed-release secret operations [9], [11], [13]) periodically without significantly compromising the processing of normal operations; and second, to illustrate how parameter selection (e.g., the trade-off between secret operation frequency and throughput) affects system performance. We hope these insights will be a valuable reference for system designers looking to implement DBFT functionality within existing BFT systems for specific application requirements.

B. Setup

We conducted experiments on a cluster of 6 Amazon EC2 m5.8xlarge instances, each with 32 virtual CPUs on a 3.1 GHz, Intel Xeon Platinum 8175 processor, 128GB memory, running Ubuntu 22.04 LTS and deploying rustc 1.67.0-nightly. Replicas were distributed across those instances as equally as possible. We use throughput and latency as our main metrics, where throughput is measured as the number of operations completed per second and latency is measured from the time an operation is generated to the time it is completed [31].

We evaluated the baseline performance and scalability of the system when running the DBFT protocol exclusively. Additionally, we measured the performance of DBFT when interleaved with the BFT protocol. In the following sections, each measurement in the figures represents the average of 10 independent runs. Unless stated otherwise, the batch size for normal operations is set to 400. **Secret operations are not batched** (the secret maker waits for one to complete before submitting the next) and are encrypted from normal operations. The checkpoint mechanism is triggered every 100 sequence numbers. The timeout for each replica is set to 5s, and each normal operation is 128 bytes in size. In order to facilitate evaluation, the length of waiting period per secret operation was set to be the same.

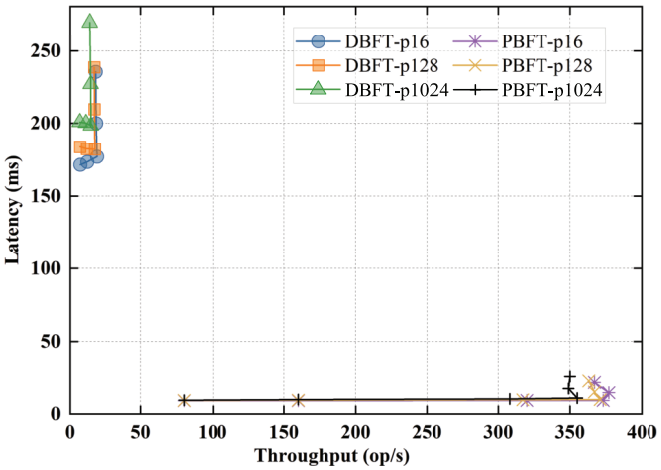


Fig. 6. Throughput vs. latency of DBFT (exclusively) and PBFT with different choices of payload sizes, 4 replicas, under the local (1ms RTT and 1Gb/s bandwidth) network setup.

C. Benchmark Under Different Choices of Payload Size

We first measured the throughput and latency of DBFT (running exclusively) and unmodified PBFT (i.e., without incorporating timed-release properties) in a setting commonly seen in evaluation of other BFT systems [12], [29] to test their L-graphs. This comparison helps understand the trade-offs involved when extending BFT protocols to use TPKE primitives for processing timed-release secret operations versus relying on conventional public key signing schemes for normal operations. The experiments were conducted in a local network environment (round-trip time (RTT) of 1ms and 1Gb/s bandwidth) with 4 replicas, with normal operation (payload) sizes varying from 16, 128, and 1024 bytes (denoted as "p16", "p128", "p1024"). We varied the operation request rate until the system saturated. Compared to PBFT, the secret maker (client) in DBFT need to additionally encrypt normal operations into secret operations, with 0ms waiting period per secret operation to exclude the impact of waiting time on system performance. Since secret operations are processed individually (not batched) in DBFT, PBFT's batch size was also set to 1 for fair comparison.

As shown in Fig. 6, achieving consensus on timed-release secret operations using DBFT introduces significant overhead through two primary factors: (Factor 1) an additional phase beyond the standard three-phase PBFT protocol, and (Factor 2) additional TPKE and symmetric encryption operations including encryption by the secret maker, partial decryption share generation, share verification, and final combination and decryption by replicas. This results in a substantial

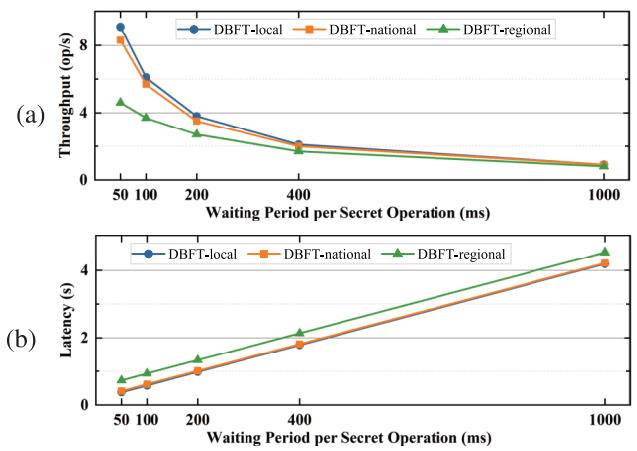


Fig. 7. Throughput and Latency of DBFT for varying waiting periods per secret operation from 50 to 1000 ms. The experiments were conducted in three different network environments: local (1ms RTT and 1Gb/s bandwidth), national (10ms RTT and 1Gb/s bandwidth), and regional (100ms RTT and 100Mb/s bandwidth) with a total of 4 replicas.

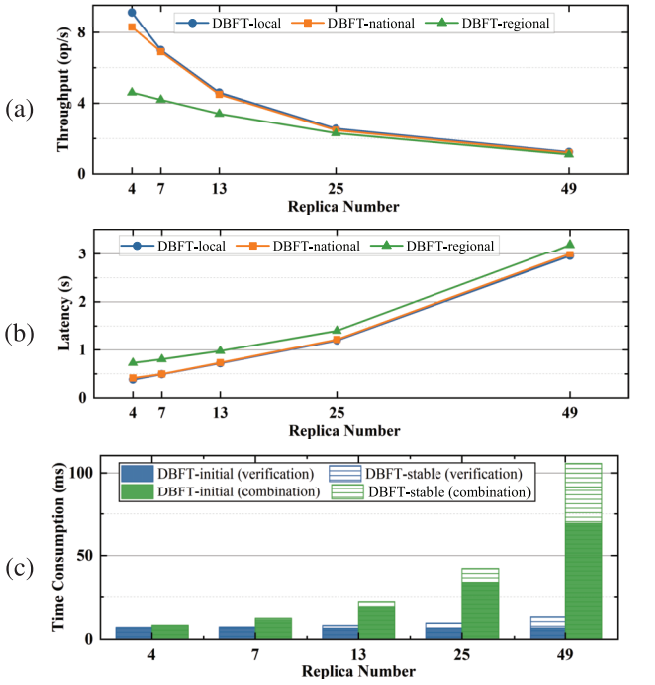


Fig. 8. Throughput, Latency and (average) Time Consumption of cryptographic operations of DBFT for varying number of replicas from 4 to 49. The experiments were also conducted in local, national, and regional network environments, with a waiting period of 50ms per secret operation.

performance gap between DBFT and conventional PBFT. DBFT achieves a peak throughput of approximately 17.1 op/s, which is only about 1/22 of PBFT's peak throughput (371 op/s), while experiencing a latency of approximately 183ms compared to PBFT's 9.5ms - a 19-fold increase. The impact of cryptographic overhead (Factor 2) becomes more pronounced with larger payload sizes (primarily the rise in the overhead of symmetric encryption operations, see Fig. 2): at 1024 bytes, DBFT's throughput decreases by approximately 22.6% compared to 16 bytes, while PBFT experiences only a 5.9% decrease. These results quantify the performance cost of incorporating timed-release capabilities on a per-operation basis in BFT systems.

D. Baseline Performance Under Different Waiting Periods Per Secret Operation

We evaluated the baseline performance of the DBFT under different waiting periods per secret operation, with the replicas set to $n = 4$. The experiments were conducted in three network environments: local (1ms RTT and 1Gbps bandwidth), national (10ms RTT and 1Gbps bandwidth), and regional (100ms RTT and 100Mbps bandwidth). Similar settings can also be found in [32]. As illustrated in Figures 7(a) and 7(b), since secret operations are processed one at a time, the system (when running DBFT protocol exclusively) does not achieve high throughput. The RTT becomes the limiting factor when it exceeds the waiting period: with a 50ms waiting period, the system achieves a throughput of approximately 4.6 op/s in the regional network, which is slightly more than half of the throughput in the national (8.3 op/s) or local (9.1 op/s) networks, with a latency approaching 1s. When the waiting period is increased to 1000ms, the RTT is no longer the limiting factor, and the throughput across all three network environments converges to around 0.9 op/s, with the latency in the regional network reaching approximately 4.5 seconds, less than 10% higher than in the other two networks.

E. Scalability

We evaluated the system's performance with varying numbers of replicas across the three network environments, with the waiting period per secret operation set to 50ms. As shown in Figures 8(a) and 8(b), compared to conventional BFTs, DBFT experiences a more significant decrease in throughput and a greater increase in latency as the number of replicas increases, particularly in networks with lower delay. In the local network, throughput drops from 9.1 op/s at 4 replicas to 1.23 op/s at 49 replicas, a decline of over 85% (similar declines of around 80% are observed in the other two network environments), while latency increases by more than 7.7 times, from 0.38s to 3s. This is closely related to the cryptographic primitives we employed (the reliance on pairing operations). As Fig. 8(c) illustrates, regardless of the network environment, the average time for the TPKE scheme to combine decryption shares increases significantly with the replicas. At 49 replicas, the average time for share combination reaches 70ms (the verification of a single decryption share takes about 7ms initially, independent of the network size). Furthermore, we observed that, during stable state, the time taken for verifying and combining decryption shares consistently exceeds that of the system's initial state. This difference is less noticeable with 4 or 7 replicas (less than a 10% increase), but becomes more pronounced as replica number rises. For example, the combination takes around 106ms at 49 replicas after stabilization, a 50% increase over the initial 70ms.

To improve system throughput when running DBFT protocol exclusively one potential approach is to batch secret operations and synchronize the execution of their associated events as much as possible, thereby parallelizing their

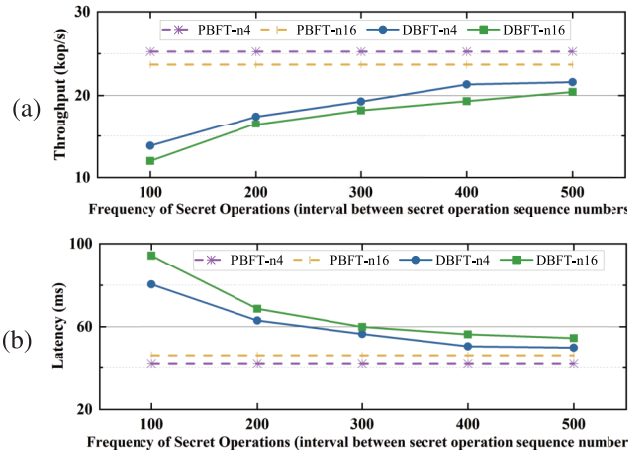


Fig. 9. Throughput and Latency of DBFT when interleaved with the PBFT protocol for varying interval between secret operation sequence numbers (secret operation frequencies) from 100 to 500. The experiments were conducted under the national (10ms RTT and 1Gb/s bandwidth) network setup with a 1000ms waiting period and a total of 4 or 16 replicas.

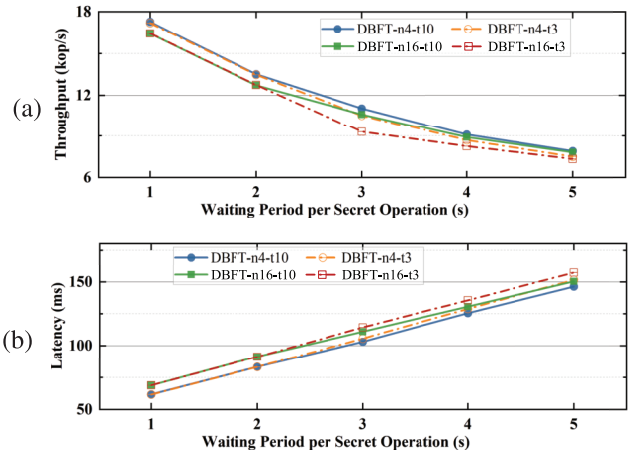


Fig. 10. Throughput and Latency of DBFT when interleaved with the PBFT protocol for varying waiting periods per secret operation from 1s to 5s. The experiments were conducted with a secret operation frequency of 200 under the national (10ms RTT and 1Gb/s bandwidth) network setup, a total of 4 or 16 replicas and a timeout of 3s or 10s.

waiting periods. However, such an approach has risks, including potential conflicts or incorrect causalities. For instance, some events may not be able to execute in parallel due to conflicts (or parallel execution may introduce new conflicts such as deadlocks). Additionally, a secret operation with a shorter waiting period but later in the sequence may not execute before an earlier operation with a longer waiting period, which may otherwise introduce incorrect causalities [33]. To resolve the challenges, additional mechanisms would need to be introduced to avoid these risks. Due to space constraints, we do not further explore this topic in this paper.

We next evaluate the performance of DBFT when interleaved with the BFT protocol.

F. Performance Under Different Secret Operation Frequencies (Mixed Mode)

The experiments were conducted with 4 replicas ('n4') and 16 replicas ('n16') under the national network setup, with a 1000ms waiting period per secret operation. We also included the performance of the unmodified PBFT system (running PBFT protocol exclusively) under the same setup as a reference. When referring to the frequency of secret operations, we mean the interval between two secret operation sequence numbers. As shown in Figures 9(a) and 9(b), when the interval is 100 sequence numbers, the throughput is only half that of PBFT. As the frequency decreases (i.e., the interval increases), the system performance gradually approaches that of the unmodified PBFT system. When the interval is 500 sequence numbers, the system achieves a throughput of 22 kop/s with 4 replicas, which is only about 10% lower than PBFT's 25.3 kop/s (latency of 49.8ms, which is about 20% higher than PBFT's 41.7ms). Additionally, as discussed in the Scalability paragraph, the impact of increasing the number of replicas on system performance becomes more pronounced when DBFT is supported (particularly at higher secret operation frequencies). For instance, PBFT experiences a throughput drop of around 5% when increasing the number of replicas from 4 to 16 (25.3 kop/s to 23.7 kop/s), while in mixed mode, with a secret operation interval of 100 sequence numbers, the drop is 14% (from 13.8 kop/s to 11.98 kop/s).

G. Performance Under Different Waiting Periods Per Secret Operation (Mixed Mode)

The experiment was conducted with 4 ('n4') and 16 ('n16') replicas under two timeout settings: 3s ('t3') and 10s ('t10'), with a secret operation frequency of 200 in the national network setup. As shown in Figures 10(a) and 10(b), system performance degrades when the timeout ≤ the waiting period. For a 3s waiting period and 4 replicas, the system achieves a throughput of 11.1 kop/s with a 10s timeout, but only 10.5 kop/s with a 3s timeout. This effect is even more pronounced with more replicas (16 replicas), where throughput drops from 10.6 kop/s to 9.3 kop/s. Additionally, latency increases by 3%-5%.

This degradation is primarily due to the timer expiring before the end of the waiting period, causing operations to be incomplete within the view and triggering unnecessary view-change protocols (not due to primary failure), leading to additional overhead. The corresponding operations must undergo a re-execution of their PRE-COMBINE or COMBINE phases after view changes to completion. We can expect to observe more significant performance drops when the timeout duration is more shorter, or when the waiting period is several times longer than the timeout, especially at higher secret operation frequencies. Therefore, it is important to configure the timeout properly to avoid view changes during the waiting period (network delays should also be considered in high-latency environments), or to make the waiting period non-blocking (though this would still require addressing potential conflicts and inconsistencies).

VII. Related Work

A. Fault Tolerant Consensus

Fault-tolerant consensus in distributed systems can be mainly categorized into CFT and BFT, with CFT handling

benign errors. Most CFTs stem from a few key protocols [1], [28], [34], [35], which influenced many subsequent designs, including BFT, and underpin various distributed coordination and data management systems.

In 1982, Lamport et al. [2] introduced the Byzantine Generals Problem, aiming to achieve consensus in the presence of Byzantine faults. Tolerating Byzantine faults requires a more complex protocol with cryptographic authentication and additional interaction phases. The initial solutions [2] in synchronous settings were communication-intensive. Shortly after, Fischer et al. [36] proved reaching consensus deterministically is impossible with even a single fault. Dwork et al. [23] later proposed the partially synchronous model, offering a new approach. Building on this, Castro and Liskov [19] introduced the practical solution under partial synchrony in 1999 (PBFT), combining ideas from Viewstamped Replication [35] and Paxos [1] to use view changes for selecting a new primary (but do not select a different set of replicas to form the new view), laying groundwork for many future BFTs. Recent advancements include using block locking [29], [37] to simplify view-change protocols and pipelined approaches to further reduce the average communication overhead per request [38]. Most state-of-the-art BFT paradigms [11], [29], [39], [40] minimize replicas communicating with each other, instead relying on various cryptographic operations to achieve indirect mutual authentication, which significantly improve throughput, though at the cost of increased latency.

The Deferred Byzantine Generals Problem extends the Byzantine Generals Problem [2] and can encompass the consistency conditions of the original problem. We use PBFT [19] notations to construct the DBFT protocol, aiming to highlight the unique challenges and modifications required for DBFT in comparison to BFTs. Given the foundational nature of PBFT, we believe DBFT can be easily extended to most BFT paradigms if using same cryptographic primitives.

B. Timed-Release Mechanism

Timed-release cryptographic primitives are finding widespread use in practical protocols, partially due to the rise of blockchain technology [3], [41]. The goal is to ensure certain ciphertext can only be decrypted after a specified time, even if one possesses the decryption key beforehand. The concept of timed-release cryptography was first proposed by May in 1993 [4], and shortly thereafter, Rivest et al. [5] developed it in a systematic and formal way. Research in timed-release cryptography has primarily focused on signature [42] and encryption [7], [8], with existing constructions broadly divided into two categories: Time-Lock Puzzles [5], [6] and Trusted Time Servers [8], [43]. The former relies on computationally intensive puzzles that require a precise amount of time to solve and cannot be solved substantially faster with large investments in hardware (unlike Proof-of-Work). However, Time-Lock Puzzles are inherently not Universal Composability secure, making them complex to use in a modular fashion [25]. The latter approach depends on trusted servers and lacks non-interactive properties.

From this perspective, The Deferred Byzantine Generals Problem can be viewed as a timed-release variant of the Byzantine Generals Problem, providing high scalability through the design of Secret Maker and waiting period. For example, it can be extended to the "schedule and reveal" paradigm (but it should be noted that in the deferred problem, each secret-generating general must be a non-essential participant in the subsequent knowledge of its respective parts of the operation; i.e., their revelation is non-essential). It can also be extended to a paradigm similar to Time-Lock Puzzles without the need to introduce complex puzzles, instead leveraging any timed event (the timed-release feature essentially depends on voting interactions). Faults not exceeding the tolerance cannot decrypt the secret before the designated time, and they cannot solve it by leveraging more computational resources.

Cryptographic primitives most closely related to our work include timed-release computational secret sharing [8] and threshold timed-release encryption [10]. DBFT differs from them in that it does not focus on individual signature generation, message encryption or secret recovery under a timed-release mechanism, but rather ensures non-faulty replicas maintain consistency in timed-release secret operations.

C. Threshold Cryptography

Threshold cryptography [14], [15] is a general technique used to protect a cryptographic secret by splitting it into multiple shares. Many signature and encryption schemes have been thresholdized [17], [44], [45]. Research in threshold cryptography, beyond improving the efficiency of encryption or signing, primarily focuses on optimizing the expensive Distributed Key Generation (DKG) protocols required for almost all threshold schemes [24], [46] (i.e., to sample a correlated partial key for each party), making decryption or signing processes semi-interactive [45] or even non-interactive [17], [18] (where parties locally produce partial shares that can be publicly aggregated to recover the message or generate the signature), and supporting dynamic thresholds [47] (allowing parties to choose a different threshold without the need to repeat the interactive setup).

We employ fully non-interactive CCA2-secure TPKE schemes [17], [18] as the cryptographic primitives to construct the DBFT protocol. However, it is the only method for constructing the solution. For instance, using timed-release cryptographic primitives would also be feasible, although the construction path or assumptions might be different.

VIII. Conclusion

We abstract the Deferred Byzantine Generals Problem, a timed-release variant of the Byzantine Generals Problem, focusing on maintaining consistency on a sequence of timed-release secret operations among replicas in the presence of Byzantine failures and define its interactive consistency conditions. We study the DBFT consensus, presenting the system model of DBFT SMR under partial synchronization using TPKE as the cryptographic primitives and highlighting core issues addressed by DBFT. We describe the DBFT protocol using PBFT notations, focusing on the modifications and

unique challenges to facilitate extensions to other paradigms, and prove it satisfies the interactive consistency conditions required by the Deferred Byzantine Generals Problem. We implement a DBFT system based on an existing PBFT codebase and evaluate its baseline performance and scalability when running DBFT exclusively. We also measured system's performance when DBFT is interleaved with the BFT protocol. Through experimental results, we show the impact of different executing modes and parameter choices on performance and discuss potential optimizations.

REFERENCES

[1] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Comput. Column)*, vol. 32, pp. 51–58, Dec. 2001.

[2] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," in *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: ACM Press, 2019, pp. 203–226.

[3] S. Nakamoto and A. Bitcoin, "Bitcoin: A peer-to-peer electronic cash system," *Bitcoin*, vol. 4, no. 2, p. 15, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[4] T. C. May. (Feb. 10, 1993). *Timed-Release Crypto*. Cypherpunks Mailing List. Accessed: Jul. 26, 2025. [Online]. Available: https://cypherpunks.venona.com/date/1993/02/msg00129.html

[5] R. L. Rivest, A. Shamir, and D. A. Wagner, *Time-lock Puzzles and Timed-release Crypto*. Cambridge, MA, USA: MIT Press, 1996.

[6] S. Agrawalr, G. Malavolta, and T. Zhang, "Time-lock puzzles from lattices," in *Proc. Annu. Int. Cryptol. Conf. (CRYPTO)*, 2024, pp. 425–456.

[7] J. H. Cheon, N. Hopper, Y. Kim, and I. Osipkov, "Provably secure timed-release public key encryption," *ACM Trans. Inf. Syst. Secur.*, vol. 11, no. 2, pp. 1–44, Mar. 2008.

[8] Y. Watanabe and J. Shikata, "Timed-release computational secret sharing and threshold encryption," *Designs, Codes Cryptography*, vol. 86, no. 1, pp. 17–54, Jan. 2018.

[9] I. Sheff, T. Magrino, J. Liu, A. C. Myers, and R. van Renesse, "Safe serializable secure scheduling: Transactions and the trade-off between security and consistency," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 229–241.

[10] L. Baird, P. Mukherjee, and R. Sinha, "I-TiRE: Incremental timed-release encryption or how to use timed-release encryption on blockchains?," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 235–248.

[11] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2021, pp. 165–175.

[12] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with BFT-SMART," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, Feb. 2014, pp. 355–362.

[13] R. Alvarez and M. Nojoumian, "Comprehensive survey on privacy-preserving protocols for sealed-bid auctions," *Comput. Secur.*, vol. 88, Jan. 2020, Art. no. 101502.

[14] Y. Frankel, "A practical protocol for large group oriented networks," in *Proc. Workshop Theory Appl. Cryptograph. Techn.*, Belgium, Apr. 2007, pp. 56–61.

[15] Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *Proc. Int. Workshop Theory Appl. Cryptograph. Techn. (ISC)*, 2007, pp. 307–315.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: ACM, 2019, pp. 179–196.

[17] D. Boneh, X. Boyen, and S. Halevi, "Chosen ciphertext secure public key threshold encryption without random oracles," in *Proc. Cryptographers' Track RSA Conf.*, 2005, pp. 226–243.

[18] B. Libert and M. Yung, "Non-interactive CCA-secure threshold cryptosystems with adaptive security: New framework and constructions," in *Proc. 9th Theory Cryptography Conf.*, Sicily, Italy, Mar. 2012, pp. 75–93.

[19] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," *OsDI*, vol. 99, pp. 173–186, Mar. 1999.

[20] G. Bracha, "Asynchronous Byzantine agreement protocols," *Inf. Comput.*, vol. 75, no. 2, pp. 130–143, Nov. 1987.

[21] V. Shoup and R. Gennaro, "Securing threshold cryptosystems against chosen ciphertext attack," *J. Cryptol.*, vol. 15, no. 2, pp. 75–96, Jan. 2002.

[22] J. H. An, Y. Dodis, and T. Rabin, "On the security of joint signature and encryption," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn.* Cham, Switzerland: Springer, 2002, pp. 83–107.

[23] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[24] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 2518–2534.

[25] C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner, "TARDIS: A foundation of time-lock puzzles in UC," in *Proc. 40th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, vol. 12698, 2021, pp. 429–459.

[26] V. Shoup, "Using hash functions as a Hedge against chosen ciphertext attack," in *Proc. Int. Conf. theory Appl. Cryptograph. Techn. (EUROCRYPT)*, 2000, pp. 275–288.

[27] V. Rijmen and J. Daemen, "Advanced encryption standard," *Federal Inf. Process. Standards Publications, Nat. Inst. Standards Technol.*, vol. 19, p. 22, Jun. 2001.

[28] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.

[29] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2019, pp. 347–356.

[30] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin One's wheels? Byzantine fault tolerance with a spinning primary," in *Proc. 28th IEEE Int. Symp. Reliable Distrib. Syst.*, Sep. 2009, pp. 135–144.

[31] X. Liu et al., "ABSE: Adaptive baseline score-based election for leader-based BFT systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 36, no. 8, pp. 1634–1650, Aug. 2025.

[32] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, Oct. 2021, pp. 35–48.

[33] A. Misra and A. D. Kshemkalyani, "Byzantine-tolerant causal ordering for unicasts, multicasts, and broadcasts," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 5, pp. 814–828, May 2024.

[34] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2011, pp. 245–256.

[35] B. Oki and B. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. 7th Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*, Aug. 1988, pp. 8–17.

[36] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[37] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, Dept. Master Appl. Sci. Eng. Syst. Comput., Univ. Guelph, Guelph, ON, Canada, 2016. [Online]. Available: http://hdl.handle.net/10214/9769

[38] X. Liu et al., "Dolphin: Efficient non-blocking consensus via concurrent block generation," *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 11824–11838, Dec. 2024.

[39] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," in *Proc. 17th Eur. Conf. Comput. Syst.*, Mar. 2022, pp. 17–33.

[40] Z. Zhang et al., "HCA: Hashchain-based consensus acceleration via re-voting," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 775–788, Mar. 2024.

[41] M. Li, Y. Chen, C. Lal, M. Conti, M. Alazab, and D. Hu, "Eunomia: Anonymous and secure vehicular digital forensics based on blockchain," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 1, pp. 225–241, Jan. 2023.

[42] J. A. Garay and M. Jakobsson, "Timed release of standard digital signatures," in *Proc. Int. Conf. Financial Cryptography (FC)*, 2003, pp. 168–182.

[43] A. C.-F. Chan and I. F. Blake, "Scalable, server-passive, user-anonymous timed release cryptography," in *Proc. 25th IEEE Int. Conf. Distrib. Comput. Syst.*, May 2005, pp. 504–513.

[44] Y. Lindell and A. Nof, "Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1837–1854.

[45] T. Ruffing, V. Ronge, E. Jin, J. Schneider-Bensch, and D. Schröder, "ROAST: Robust asynchronous Schnorr threshold signatures," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 2551–2564.

[46] S. Garg, D. Kolonelos, G.-V. Policharla, and M. Wang, "Threshold encryption with silent setup," in *Proc. Annu. Int. Cryptol. Conf. (CRYPTO)*, 2024, pp. 352–386.
[47] C. Delerablée and D. Pointcheval, "Dynamic threshold public-key encryption," in *Proc. Annu. Int. Cryptol. Conf.* Santa Barbara, CA, USA: Springer, Aug. 2008, pp. 317–334.

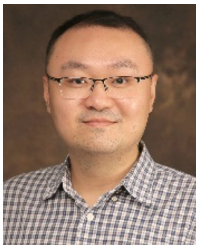
Xuyang Liu (Graduate Student Member, IEEE) received the B.E. degree in computer science and technology from Beijing Institute of Technology in 2022. He is currently pursuing the dual Ph.D. degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology, and the School of Computer Science, The University of Auckland. His research interests include applied cryptography, blockchain technology, and distributed consensus.

Zijian Zhang (Senior Member, IEEE) was a Visiting Scholar at the Computer Science and Engineering Department, State University of New York at Buffalo, in 2015. He is a Professor at the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is also a Research Fellow with the School of Computer Science, University of Auckland. His research interests include design of authentication and key agreement protocol and analysis of entity behavior and preference.

Zhen Li is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. His research interests include privacy computing, AI security, and blockchain.

Peng Jiang received the Ph.D. degree from Beijing University of Posts and Telecommunications in 2017. She is an Associate Professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing. She has edited three book/book chapters and authored or co-authored more than 50 refereed papers in international journals and conferences, including ESI Highly Cited Papers in 2020, ESI Hot Paper in 2021, and Editor's Choice Paper. Her research interests include cryptography, information security, and blockchain technology. She also serves as the Co-Chair of SIG for the IEEE TEMS's TC on Blockchain and Distributed Ledger Technologies. She served as an Associated Editor for *Computer Standards and Interfaces*, a guest editor for many journals, and a program committee member of many conferences.

Yajie Wang received the Ph.D. degree in Cyberspace Security from Beijing Institute of Technology, Beijing, China, in 2023. He is currently a Post-Doctoral Research Fellow at the School of Cyberspace Science and Technology, Beijing Institute of Technology. His research interests include data privacy protection and artificial intelligence security.



Meng Li (Senior Member, IEEE) received the Ph.D. degree in computer science and technology from the School of Computer Science and Technology, Beijing Institute of Technology (BIT), China, in 2019. He was a Post-Doctoral Researcher at the Department of Mathematics and HIT Center, University of Padua, Italy, where he is with the Security and PRIvacy Through Zeal (SPRITZ) Research Group led by Prof. Mauro Conti (IEEE Fellow). He is an Associate Professor and Personnel Secretary at the School of Computer Science and Information Engineering, Hefei University of Technology (HFUT), China. He was sponsored by ERCIM 'Alain Bensoussan' Fellowship Programme (from October 1, 2020 to March 31, 2021) to conduct Post-Doctoral Research supervised by Prof. Fabio Martinelli at CNR, Italy. He was sponsored by China Scholarship Council (CSC) as a Joint Ph.D. Student (from September 1, 2017 to August 31, 2018) supervised by Prof. Xiaodong Lin (IEEE Fellow) in the Broadband Communications Research (BBCR) Laboratory, University of Waterloo and Wilfrid Laurier University, Canada. He is supported by CSC as a Visiting Scholar (from March 1, 2025 to February 28, 2026) collaborating with Prof. Mauro Conti (IEEE Fellow) at the HIT Center, University of Padua, Italy. His research interests include security, privacy, applied cryptography, blockchain, TEE, and internet of vehicles. In this area, he has authored or co-authored 106 papers in topmost journals and conferences, including IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY (TIFS), IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING (TDSC), IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, *ACM Transactions on Database Systems*, TPDS, IEEE TRANSACTIONS ON SERVICES COMPUTING, COMST, S&P, MobiCom, and ISSTA. He is a Senior Member of CIE, CIC, and CCF. He has served as a TPC member for conferences, including ICDCS, TrustCom, ICC, Globecom, and HPCC. He was recipient of 2024 IEEE HITC Award for Excellence (Early Career Researcher). He is an Associate Editor for TIFS, TDSC, *EURASIP Journal on Information Security*, IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, and COMNET.

Liehuang Zhu (Senior Member, IEEE) is a Professor at the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is selected into the Program for New Century Excellent Talents in University from Ministry of Education, China. His research interests include cryptographic algorithms and secure protocols, the Internet of Things security, cloud computing security, big data privacy, mobile and Internet security, and trusted computing.