Slarm: SLA-aware, Reliable and Efficient Transaction Dissemination for Permissioned Blockchains

PAPER # 45

The blockchain paradigm has attracted diverse smart contract applications to be deployed on a blockchain consisting of a P2P network. However, no service-level agreements (SLA) mechanism has been proposed to enforce the commit deadlines of blockchain transactions, although these applications are often interactive with clients via phones and desire stringent commit deadlines (e.g., tens of seconds). Existing P2P reliable multicast protocols for blockchains disseminate transactions regardless of their SLA deadlines and are too heavyweight, which incur significant traffic on blockchains' P2P network. Moreover, these protocols are vulnerable to faulty P2P nodes, and their specific protocol messages are vulnerable to targeted attacks.

This paper presents Slarm, the first SLA-aware and reliable transaction multicast protocol for permissioned blockchains. We leverage the strong integrity and confidentiality features of Intel SGX to invent a network-efficient and message-oblivious P2P multicast protocol, which efficiently guarantees transactions' end-to-end SLA requirements in a localized way and defends against both the integrity and targeted attacks. Evaluation of Slarm on Ethereum with five state-of-the-art P2P multicast protocols and five diverse real-world SLA-oriented applications shows that: (1) Slarm eliminates targeted attacks on its SLA-enforcing messages; and (2) even with the existence of transaction spikes and attacked nodes, Slarm achieves a much higher transaction SLA satisfaction rate with reasonable high throughput compared with the evaluated relevant protocols .

1 Introduction

A blockchain is an immutable ledger for recording transactions maintained by mutually untrusted nodes [29, 78]. Some of these nodes (i.e., consensus nodes) execute a *block consensus* protocol to agree on the order of transactions and commit transactions in blocks. A blockchain can be permissionless or permissioned. A permissionless blockchain (e.g., Bitcoin [78]) does not manage node identities, so it usually incites nodes to follow the blockchain's protocol using cryptocurrencies, which requires nodes to contribute either high computation resources (e.g., PoW [78]) or wealth (e.g., PoS [9]).

A permissioned blockchain (e.g., Ethereum private networks [8]), on the other hand, runs among a set of explicitly identified nodes [29, 33]. Due to nodes' explicit identities, a permissioned blockchain can identify faulty nodes that do not follow the blockchain's protocol and thus is decoupled from the cryptocurrency. Moreover, a permissiond blockchain can run a fast Byzantine-fault-tolerant (BFT) protocol (e.g., Clique [4] and IBFT [77]) for block consensus. Therefore, permissioned blockchains generally achieve much higher performance and energy efficiency than permissionless blockchains (§2.1), which is more suitable to deploy performance-sensitive applications (e.g., online auction [60, 88] and trading [67, 86]). This paper focuses on the permissioned blockchains.

A blockchain application is implemented with smart contracts: *stateful* programs stored on the blockchain that can be invoked by clients via transactions [37, 96]. Transactions invoking one smart contract must be committed onto the blockchain with the same complete (gap-free) order as the order issued by the client.

As more performance-sensitive applications have been deployed on permissioned blockchains, the service-level agreements (SLA) on transaction's commit latency is becoming increasingly important. For instance, Ethereum [96] allows a smart contract to specify an absolute time deadline on accepting transaction invocations [18], and such a deadline can be short (e.g., a blockchain-driven flash auction is opened for only a few minutes [25, 60, 88]). More importantly, different transactions may have different SLA latency requirements. In the online trading application [67, 86],

Author's address: Paper # 45.

the realtime payments need to be committed to the blockchain as soon as possible, while the non-realtime payments often tolerate extra delay. We call the transactions that desire commit latency requirement "SLA transactions."

The commit latencies of blockchain transactions are mainly determined by the transaction's dissemination latency to the entire network (§2.1). A permissioned blockchain may contain a large number of nodes (e.g., 10K [45]), thus it often adopts P2P multicast protocols such as the Gossip protocol [50, 66] to disseminate transactions.

Unfortunately, despite much effort, state-of-the-art still cannot meet the transactions' SLA requirement on commit latency. The main reason is that existing P2P multicast protocols are unidirectional. They disseminate all transactions with the same set of dissemination parameters, which determines a transaction's priority and speed during the dissemination, regardless of the transactions' SLA stringencies. Moreover, existing P2P multicast protocols lack a feedback mechanism for adapting the dissemination parameters. As a result, non-stringent SLA transactions may be disseminated throughout the network much faster than their latency requirements, while SLA-stringent transactions are delayed by these non-stringent transactions.

Worse, the smart contract transactions must be committed to the blockchain sequentially without gaps, but existing P2P multicast protocols lack a reliable and efficient mechanism to provide the gap-free guarantee. Existing P2P multicast protocols for blockchains can be divided into two categories: Gossip and reliable multicast protocols. Developers of blockchain applications face the dilemma between the low communication overhead but best-effort dissemination in Gossip, and the gap-free dissemination but high overhead afforded by the reliable multicast protocols.

To meet the SLA latency requirement, the former option is unacceptable because it is unreliable and introduces gaps in nodes' received transactions, which incurs high latency for smart contract transactions (must be committed without gaps) in normal case. Yet the latter option is also unacceptable due to the risk of performance collapse (high overhead) under unusual but not rare traffic spikes.

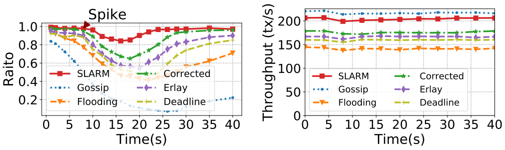
We ran the online trading application [67, 86] on Ethereum with flooding [10], Gossip [66], and three reliable multicast protocols [53, 59, 80]. All existing protocols exhibit low SLA satisfaction rates under traffic spikes (Figure 1a). For instance, in the Gossip protocol (Figure 2a), the commit latency of 87.3% SLA transactions is larger than their SLA deadline (32s) under traffic spikes.

In this study, we introduce a new design point: an SLA-aware and Reliable transaction Multicast (Slarm) protocol, which disseminates transactions according to their SLAs and ensures nodes receive gap-free transactions with low communication overhead in permissioned blockchains.

Slarm is a bi-directional multicast protocol. In the forward directional multicast, Slarm disseminates transactions individually by Gossip. During Gossip, nodes prioritize the disseminations of SLA-stringent (i.e., less remaining deadline) transactions, which prevents stringent transactions being delayed by non-stringent transactions. Upon detecting gaps in the received transactions, nodes conduct light-weight transaction gap-fillings.

In the backward directional multicast, Slarm disseminates transactions in committed blocks throughout the network. In typical blockchain systems, consensus nodes must disseminate all committed blocks throughout the network [78, 96]. Slarm leverages this block dissemination as a feedback to the forward transaction dissemination. Slarm adjusts the Gossip parameters of the forward directional multicast according to transactions' remaining deadlines in committed blocks, making SLA transactions' dissemination speeds match their SLA latency requirements.

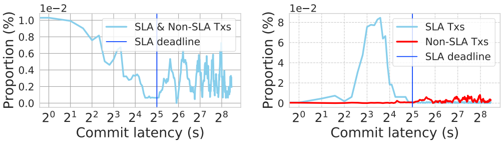
However, making this new SLA-aware P2P multicast protocol practical in the blockchain domain still faces two major challenges. First, P2P nodes must infer each transaction's SLA stringency (i.e., remaining deadline) in a decentralized way without a globally synchronized clock. Since the clocks of blockchain nodes are loosely synchronized on the Internet, each node's internal clock and transactions' timestamps may have time errors of several minutes [19], much larger than a



(a) SLA guarantees.

(b) Throughput.

Fig. 1. SLA guarantee for SLA transactions and throughputs for all transactions on the online trading application (details of evaluation settings are in §7). At 0s, all systems are at peak throughputs; at 8s, a spike of 200 tx/s SLA transactions lasts for 5s.



(a) Gossip.

(b) SLARM.

Fig. 2. The probability distribution of transactions' commit latencies in Gossip and SLARM's multicast protocol. We use the same setting with Figure 1 and only show the latency distribution of transactions submitted after 8s.

transaction's commit latency (e.g., tens of seconds as evaluated in §7).

The second challenge is the security issue in blockchains: SLARM's SLA mechanism is run by each node, where some nodes can be faulty and corrupt the mechanism. Moreover, specific protocol messages (e.g., SLA transactions) in SLARM's multicast protocol (also in existing P2P multicast protocols [34, 53, 59, 80]) are vulnerable to targeted attacks. For instance, in the online auction application [25, 60, 88], attackers can delay or drop a bidding transaction, causing the transaction to miss the auction deadline.

To tackle these two challenges, SLARM includes a *conservative scheduler* based on Intel SGX [74], an increasingly prevalent trusted execution environment. SGX has been pervasively used to protect the P2P layer [35, 49], the consensus layer [42, 75, 100], the contract layer [47, 63], and the application layer [70, 73] of many blockchain systems.

The conservative scheduler infers the stringency (i.e., remaining deadline) of SLA transactions on each node *conservatively*: the inferred remaining deadline of each transaction is more stringent than the transaction's actual remaining deadline. SGX provides strong confidentiality and integrity protection for code and data. SLARM leverages SGX to protect its scheduling mechanism on each node, preventing faulty nodes from corrupting the scheduling of transactions. To handle targeted attacks, the conservative scheduler also provides high obliviousity for all SLARM messages (§5).

We implemented SLARM on Ethereum [96] and evaluated it on both our cluster and AWS [11]. We compared SLARM with two traditional P2P multicast protocols (Gossip [50, 66] and flooding [26]) and three state-of-the-art reliable multicast protocols (Erlay [80], Corrected Gossip [59], and Deadline Gossip [53]). Specifically, Erlay [80] is the latest reliable multicast protocol for blockchains. We evaluated all protocols with diverse real-world applications. Our evaluation and analysis show that:

1. SLARM completely eliminates targeted attacks on its SLA-enforcing messages (§5.2);
2. Figure 1a and Figure 2b show that, even during transaction spikes, SLARM achieves much higher SLA satisfaction rate for SLA transactions than all the evaluated relevant protocols;
3. Figure 1b shows that, compared to Gossip, the most light-weight protocol among the five evaluated P2P multicast protocols, SLARM achieves a reasonable overhead of 6.9% on the throughput of all SLA and non-SLA transactions.

The major novelty of this paper is SLARM, the first P2P reliable multicast protocol that meets the important SLA requirements on commit latency in the blockchain domain. SLARM is secure, efficient, and can support both general blockchain block consensus protocols and applications. SLARM can enable people to develop even more interesting blockchain applications with heterogeneous SLA requirements (§7.5), and secure P2P protocols for safety-critical systems. SLARM source code and evaluation results are released on github.com/sigmetrics21/slarm.

The rest of the paper is as follows: §2 introduces the background. §3 gives an overview of SLARM. §4 introduces SLARM's protocol. §5 gives a security analysis, §6 presents implementation details, §7 shows our evaluation. §8 introduces related work, and §9 concludes the paper.

2 Background

2.1 Permissioned Blockchain

A permissioned blockchain runs a distributed block consensus protocol among a group of identified nodes (i.e., consensus nodes) to agree on which block can be committed in a total order onto the blockchain, where a block contains many client transactions. A transaction is defined as *committed* if a block containing this transaction is committed. We define the client-perceived commit latency (in short, commit latency) as the latency between the time a client submits a transaction and the time this transaction be committed.

Transaction life cycle. In a typical permissioned blockchain, the life cycle of a client transaction mainly contains two phases: (1) the dissemination of the transaction from the client to the entire network, and (2) the consensus and dissemination of a block containing this transaction.

In the first phase, it is essential for a permissioned blockchain to disseminate transactions to the **entire network**. The main reason is that transactions must reach the consensus nodes, which are almost unknown during the transaction dissemination. In a permissioned blockchain, some nodes are elected as consensus nodes to agree on committing which block onto the blockchain. In some consensus protocols (e.g., PoS[9] and PoET [42]), any node throughout the network has the right to propose a new block. Some consensus protocols (e.g., PoA [4]) periodically re-elect consensus nodes among all nodes in the P2P network, while some other protocols (e.g., Algorand [54]) make their consensus nodes be hidden from other nodes and clients. Prior work also shows that full converge of the entire network is important for security [3, 80].

The first part often takes a major portion of the time cost of a transaction's life cycle, so it is the major bottleneck and improvement spot of a single transaction's commit latency. A blockchain's consensus protocol only provides limited throughput (e.g., 200 tx/s in Clique consensus protocol), which is usually much lower than the number of transactions submitted by clients per second [12, 16]. A large portion of the client transactions must wait a long time for commitment in nodes' transaction pools [23], which causes a long time cost (Figure 2a). Our evaluation shows that the second part (block consensus and dissemination) often has stable latency and took about 5.7s (only 33.5% of the entire life cycle) in Ethereum with Gossip and the Proof-of-Authority (PoA) consensus protocol (§7).

Why permissioned blockchain? We focus on the permissioned blockchain for two reasons. First, a permissioned blockchain often has much higher performance than a permissionless blockchain, so a permissioned blockchain is more pervasively used in performance-sensitive applications (with SLA requirement on commit latency) and thus faces a more stressful P2P network.

Second, permissionless blockchains already have cryptocurrency incentive schemes motivating P2P nodes to prioritize the disseminations and the commitments of the more valuable transactions (a kind of SLA scheme), where the client can assign a transaction with a higher transaction fee to achieve lower commit latency. However, a permissioned blockchain lacks such a scheme.

We implemented Slarm on the codebase of Ethereum [96] and deployed Slarm in a permissioned blockchain network. Ethereum is one of the most notable open-source blockchain systems, which is fully tested and actively maintained. Ethereum can either be deployed as a permissionless blockchain (i.e., Ethereum Mainnet [15]) or a permissioned blockchain (i.e., Ethereum private networks [8]). A large number of permissioned blockchain systems [13, 20, 21, 24] and applications [14, 22, 28] have been built upon Ethereum.

2.2 Intel SGX

Intel Software Guard eXtension (SGX) [74] is a prevalent trusted execution hardware product. SGX provides a secure execution environment called enclave. Data and code in an enclave cannot be tampered with (integrity) or revealed (confidentiality) from outside. A process running outside the enclave can invoke an SGX `ECall` to switch its execution into the enclave and to execute a statically-shielded function in the enclave; a process running in an enclave can invoke an `OCall` to switch its execution outside the enclave.

Why does Slarm use SGX? Slarm uses SGX for two reasons. First, we implemented a fine-grained trusted timer based on the coarse-grained SGX trusted timer (§5.1) to measure the elapsed time of SLA transactions spent on nodes and during node-to-node transmissions. Second, Slarm uses the integrity feature of SGX to maintain the SLA metadata of SLA transactions. In Slarm, each SLA transaction is piggybacked with an SLA metadata indicating the remaining time to its deadline, which is updated by nodes in their SGX enclaves during the dissemination of the transaction.

3 Overview

3.1 Deployment Requirements and Threat Model

Deployment requirements. Slarm is designed for permissioned blockchains deployed in the Internet-wide WAN networks. In this blockchain's P2P network, all P2P nodes are granted members of the permissioned blockchain. Some P2P nodes can be elected as consensus nodes to agree on committing which block onto the blockchain. Slarm is designed to support general consensus protocols, where consensus nodes can be unknown, hidden, or constantly changing. Therefore, clients in Slarm submit transactions through P2P disseminations to the entire network using gossip-like P2P multicast protocols in order to reach consensus nodes.

The permissioned blockchains that deploy centralized consensus nodes are out of the scope of this paper. For instance, Hyperledger Fabric [29] requires all consensus nodes to be explicitly deployed in the centralized way (e.g., in a single datacenter), and clients directly send transactions to these consensus nodes without using P2P dissemination protocols.

Threat model. Same as typical permissioned blockchains' P2P networks, we consider Slarm's network an asynchronous, Internet-wide network. For nodes that have joined Slarm's permissioned blockchain (network), any components of these nodes running outside of SGX can be faulty. Any Slarm node with a faulty component is called a faulty node in this paper. Specifically, faulty nodes can randomly drop and delay clients' transactions as well as Slarm protocol messages. All nodes have only loosely synchronized clocks, and faulty nodes can manipulate their local clocks. Clients can submit transactions to the blockchain by sending the transactions to any blockchain nodes. All clients are not trusted. A faulty client can send corrupted or illegal transactions (e.g., invoking an invalid smart contract).

Each SLA transaction is piggybacked with an SLA metadata indicating its remaining time to deadline during dissemination (§4.1.1). To prevent faulty nodes from maliciously tampering a transaction's SLA metadata and disseminating the transaction to its peers, Slarm's runtime system updates only the SLA metadata within each node's SGX enclave launched by Slarm. Outside the Slarm's enclaves, all client transactions (including both SLA and non-SLA transactions) are encrypted and oblivious. In each node's enclave, transactions are decrypted (§4.1). In Slarm, all the firmware and hardware components of SGX are trusted. We assume that the threads in SGX are subject to a normal OS scheduling and can be scheduled out at any moment (§5.1). Hiding traffic endpoints (Karaoke [64] and Stadium [90]), SGX micro-architecture side-channels [95], and SGX Iago attacks [41] are out of the scope of this paper.

3.2 SLA in Slarm

SLA is an agreement [61, 82, 89] between a service provider and customers on the service quality (e.g., performance and reliability). Our Slarm system is the service provider, and a distributed blockchain application with its clients are the customers. Specifically, a distributed blockchain application for Slarm consists of two parts:

(1) Smart contracts that implement the application logic and manage the system configurations. A smart contract is a stateful blockchain program that can be invoked via client transactions submitted to the blockchain [7, 38, 48, 96]. Smart contract applications often desire the SLA requirement on client-perceived commit latency.
(2) *SLAs* that specify the SLAs of different transactions in this application (e.g., high-risk or low-risk events in the disease control application [5, 6, 99]) and assign clients with different permissions in submitting transactions (e.g., only eligible clients can report high-risk events). The SLA specification is recorded on the blockchain, and only the designated client has the permission to modify the SLA specification. Upon receiving a client transaction, **Slarm nodes** determine each transaction's SLA independently according to the transaction's content and the SLA specification on chain.

In Slarm, the SLA of a transaction is a 2-tuple *SLA:*$\{$*Deadline*, *Sequential*$\}$, where *Deadline* is the desired commit latency of the transaction (e.g., 16s), and *Sequential* indicates whether the transaction need be committed sequentially according to its sequence number issued by the client. For example, a client's transactions that invoke the same stateful smart contract must be sequentially committed by the consensus nodes without gaps (i.e., tx_2 in Figure 3). The client assigns the transactions that require sequential execution with continuous sequence numbers.

Suppose a gap exists in the received transactions of a consensus node. If these transactions require sequential execution, the consensus node can only commit the transactions after the gap until the gap is fixed. If these transactions do not require sequential execution, they can be executed out-of-order, so the node directly commits these transactions without waiting for the gap is fixed.

Suppose an uncongested network, and suppose the median commit latency of a transaction is T (e.g., 16s in the Ethereum-PoA system with 5000 nodes in Table 3). Given an application (e.g., online trading in Table 2), Slarm by default considers all smart contract transactions as SLA transactions, and by default sets the two tuples as $(c \times T, Yes)$ for all the SLA transactions, where c is a configurable constant among clients. Because all clients and P2P nodes have loosely synchronized clock in Slarm, and faulty nodes can manipulate their local clock, Slarm uses elapsed time instead of absolute clock time (§4.1).

In Slarm, the SLA satisfaction rate p is defined as the portion of SLA transactions meeting their SLA deadlines. Suppose clients submit n SLA transactions with deadline $c \times T$ per second to the blockchain network. If n does not exceed the maximum throughput of the consensus protocol deployed in Slarm and the network capacity of Slarm's P2P network, then, Slarm guarantees that all SLA transactions can meet their SLAs with high probability p (96.0% when $c = 2$, proved in §4.3). Note that, given the same network condition, all typical blockchains' Gossip [50, 66], flooding [10], and existing reliable multicast protocols [34, 53, 59, 80] enforce much lower SLA satisfaction rate than Slarm (Figure 1a) when they are integrated into blockchains (e.g., Erlay [80]).

3.3 Slarm Overview

Figure 3 shows Slarm's architecture, Slarm's components are in green. Each Slarm node, including both normal nodes and consensus nodes (§2.1), has two trusted modules and two untrusted modules. The scheduler and reliable multicast module running in an SGX enclave (orange color) are trusted. The *scheduler module* prioritizes client transactions and adapts transactions' dissemination speeds according to their SLAs. The *reliable multicast module* disseminates transactions and conducts a gap-filling task when it finds a gap from its received per-client SLA transaction sequence; for non-SLA transactions, gap-filling is unnecessary. These two trusted modules are 1.7k LOC (§6), and the enclave memory stores only uncommitted transactions. If this node is a consensus node,

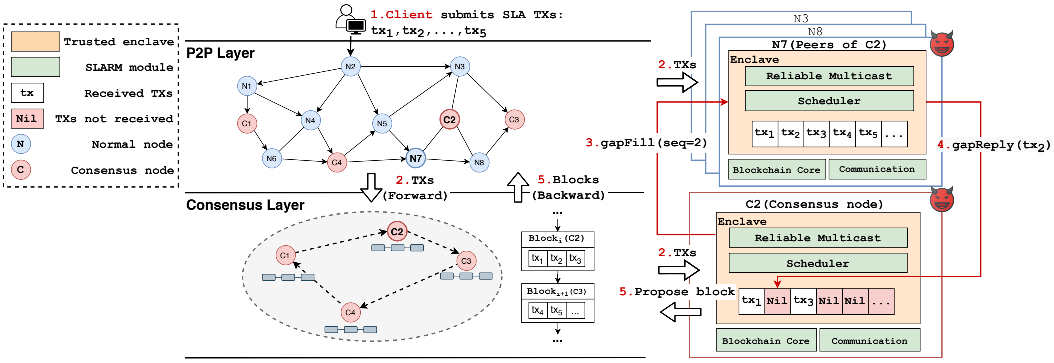


Fig. 3. SLARM's architecture. All SLARM components are in green.

transactions are forwarded to the blockchain core module.

On each SLARM node, the blockchain core module (including the consensus protocol) and network communication module (including TCP/UDP) are outside SLARM's enclave and are not trusted. The blockchain core module performs consensus on committing which block, maintains a local copy of the blockchain, and executes committed transactions extracted from committed blocks.

Transaction lifecycle. Figure 3 also shows the life cycle of an SLA transaction in SLARM: (1) A client submits a series of SLA transactions to a nearby node N2. (2) After N2's scheduler module receives these transactions, N2 decides the SLAs of these transactions (§3.2) and piggybacks each transaction with an SLA metadata indicating the remaining time to the transaction's SLA deadline. N2 initializes the SLA metadata as the transaction's SLA deadline then disseminates the transactions.

During the dissemination, each node updates the remaining deadline of these transactions and prioritizes the propagation of them according to their remaining deadlines (§4.1.1). To prevent faulty nodes maliciously altering the remaining deadline of SLA transactions, the updating is only involved in each node's SGX enclave.

If a node C2 (3) detects a gap (tx_2) in its received transactions, C2 sends a `gapFill` message to ask for tx_2 from a random subset of peers (§4.2). (4) If a peer in the subset has the transaction, the peer replies C2 with a `gapReply` message. In SLARM, non-SLA transactions do not involve gap-filling. (5) The consensus nodes pack transactions into blocks and agree on which block to commit next. P2P nodes can adjust the transactions' dissemination speeds by updating the Gossip parameters according to the transactions' remaining deadlines in the committed blocks (§4.1.2).

Security challenges. An open security challenge is that a P2P reliable multicast protocol for blockchains must tackle targeted attacks on protocol messages. Recent work [94] shows that attackers can selectively defer certain types of P2P protocol messages during a client transaction's dissemination, which can trigger the default peer adjustment mechanism in a P2P network and maliciously make certain victim nodes adjust most of their peers to faulty nodes. Such an Eclipse attack can further arbitrarily delay the dissemination of transactions of these victim nodes and violate the transactions' SLA in SLARM.

Even if such Eclipse attacks do not exist, because the `gapFill` correction messages and normal `disseminate` messages are explicitly distinguishable in existing P2P reliable multicast protocols, attackers can easily selectively drop protocol messages and stop the entire commit progress of many clients' SLA transactions. Existing reliable multicast protocols [34, 53, 59, 80], including Erlay [80], the latest notable P2P reliable multicast protocol tailored for blockchain security, are especially vulnerable to such attacks. §5 will present the complete set of attacks that SLARM aims to tackle.

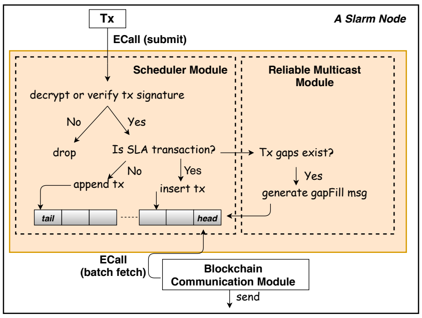
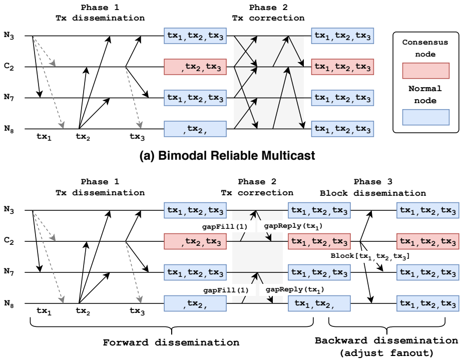


Fig. 4. Slarm's enclave code invocations (via ECall) and outbound messages (via ECall). Enclave is in orange.



(a) Bimodal Reliable Multicast

(b) SLARM Reliable Multicast

Fig. 5. Bimodal [34] and Slarm multicast.

4 The Slarm Basic Protocol

This section describes Slarm's bi-directional reliable multicast protocol. Slarm disseminates transactions to the entire network by Gossip. To enforce the SLA on commit latency, Slarm's conservative scheduler module prioritizes transactions (§4.1.1) and adjusts transactions' dissemination speeds according to their SLA deadlines (§4.1.2). For enforcing the SLA on sequence execution of smart contract transactions, the reliable multicast module efficiently fixes lost transactions, which ensures nodes receive gap-free transactions (§4.2).

On a Slarm node N, when disseminating an SLA transaction, Slarm's scheduler module subtracts the deadline of this transaction with two trustworthy (conservative) time variables: $N.RTT$, the recent worst-case Round Trip Time cost node N's peers sending a transaction to N; and tx_N^{wait}, the time cost an SLA transaction tx spent on node N's scheduler queue. §5.2 will present how Slarm makes these two variables *conservative*: Slarm can avoid faulty nodes' malicious behaviors of making the subtracted elapsed time from a transaction's remaining SLA deadline smaller than the actual elapsed time of the transaction's dissemination path. To ease discussion, this section assumes these two variables are conservative first, and there exists a trusted timer to measure millisecond-level time durations. We will discuss the conservative time measurement mechanism and the trusted timer design in §5.

4.1 Enforcing SLA on Commit Latency

The end-to-end commit latency of a transaction contains two parts: the transaction dissemination latency τ_d (from a client to the entire network) and the consensus latency τ_c (including the time cost of agreeing which block to commit and that of the committed block reaching a connected peer of the client, see §2.1). According to our evaluation (§7.1), τ_d takes up about 52.3% - 86.1% of the end-to-end commit latency and varies significantly with the transaction workload. The consensus latency (τ_c) is often stable (§2.1), and the block commit and dissemination events are infrequent (e.g., happens once in the network every 5s in PoA [4]). Therefore, Slarm assumes τ_c as a constant and subtracts τ_c from a transaction's SLA deadline once a transaction is sent from its client to a connected P2P node. Slarm focuses on making τ_d meet the remaining SLA deadline.

4.1.1 Prioritizing SLA Transactions Each node runs a scheduler module in Slarm's SGX enclave (Figure 4) to prioritize SLA transactions over non-SLA transactions. This module is essential because typical applications often have a large portion of non-SLA transactions (Table 2). Non-SLA transactions can easily block SLA transactions and violate SLAs (§7). Moreover, transactions with the same SLA deadline requirement can have different remaining SLA deadlines during dissemination.

We now present Slarm's transaction prioritizing mechanism. As shown in Figure 4, when a Slarm P2P node N receives a transaction tx sent from a client, N submits the transaction to its local Slarm enclave by an `ECall`. The scheduler module running in N's Slarm enclave (§3.3) verifies tx's signature and retrieves tx's SLA requirements from the SLA specification stored on chain (signed by the administrator). The scheduler then generates an SLA metadata m, which indicates the transaction's initial remaining SLA deadline, appends m to tx, and inserts tx into Slarm's local priority queue according to the deadline m (a smaller deadline means higher priority).

If transaction tx is sent from N's peer, the same `ECall` is invoked to push the transaction to N's Slarm enclave. N's peer has encrypted tx, so N's enclave decrypts tx, subtracts the remaining deadline with the actual time spent in transferring tx from N's peer (suppose it is N_{prev}) to N.

However, measuring the actual transfer time of tx is quite challenging. One native approach is to record the sending timestamp of tx in N_{prev}'s Slarm enclave and to subtract N's local receiving timestamp with this sending timestamp in N's Slarm enclave. This approach is problematic for two reasons. First, there is no globally synchronized clock among nodes. Different nodes' clocks may have time drifts of tens of seconds or even minutes [17, 19, 72]. Second, if N or N_{prev} is faulty, their local times can be manipulated by the faulty host, making the time subtracted from τ_d different with the actual time spent during transmission.

To tackle this challenge, we present a trustworthy mechanism to record N's peers' round-trip time cost ($N.RTT$) conservatively as tx's elapsed time during transmission from N's peers to N. The details are in §5.2. The scheduler module subtracts each received SLA transaction's remaining deadline (i.e., SLA metadata m) with $N.RTT$ and inserts the transaction into the local priority queue according to m. Slarm's own `gapFill` messages are also inserted to the queue's head (§4.2). Non-SLA transactions are appended to the queue's tail.

The communication module outside the enclave retrieves messages in the priority queue and sends the messages to the node's peers. To achieve obliviousity against attacks outside the enclave, in Slarm, the communication module does a batch fetch of messages from the priority queue's head using an `ECall`, subtracts each SLA transaction's τ_d with its time cost tx_N^{wait} spend on this node, encrypts each message, exits the `ECall` with the batch, and sends the batch. If the node is a consensus node, the node proposes the batch in one block.

Overall, updating (subtraction) of the SLA metadata indicating a transaction's remaining deadline is only involved in each node's Slarm enclave after receiving a transaction and before sending the transaction to peers. This prevents malicious nodes from altering transactions' SLA metadata.

4.1.2 Fanout Adaptive Gossip The scheduler module adjusts SLA transactions' dissemination speeds to their SLA deadlines by adapting the Gossip fanouts. Fanout refers to the number of nodes selected in each round of Gossip, and is the critical parameter that determines the overall dissemination latency of Gossip. Using a higher fanout reduces Gossip's dissemination rounds and leads to lower dissemination latency, although too high a fanout inhibits the performance due to the increased message overhead.

All existing P2P multicast protocols are conceptually a uni-directional protocol: they do not include any form of feedback mechanism on the multicast strategy. Specifically, they use the same set of multicast parameters (i.e., fanout) for all transactions with different SLA deadlines, without providing the P2P nodes any feedback regarding the SLA on the commit latency. For instance, in Gossip protocols described in [34] and [91], the fanout is 1, while in [51] and [55] the fanout is another constant.

In these Gossip protocols with fixed fanouts, the non-stringent transactions may be disseminated to the entire network much earlier than its SLA deadline, while the SLA stringent transactions are delayed. Therefore, it is essential to disseminate different SLA transactions with different speeds throughout the network.

To achieve this goal, our idea is that we can leverage the block dissemination of a blockchain's consensus protocol as an SLA feedback to design a new bi-directional Gossip protocol for Slarm. As shown in Figure 5, nodes first disseminate transactions throughout the network (forward dissemination), then adapt the Gossip fanouts of transactions with different SLAs according to transactions' remaining deadlines in the committed blocks (backward dissemination).

On the forward, the P2P nodes first disseminate all transactions with a default fanout f (e.g., $f = 10$). On the backward, the consensus nodes disseminate committed blocks containing the transactions throughout the network; P2P nodes receive the committed blocks and monitor the committed transactions' remaining deadlines. Some transactions of SLA$_1$ may be committed to the blockchain with a negative remaining deadline. Nodes compute the moving mean fraction f_{MA} of SLA$_1$ transactions committed to the blockchain with negative remaining deadlines in recent blocks. If f_{MA} is larger than a certain threshold (e.g., 50%), these transactions must be disseminated faster by the P2P nodes, and nodes therefore set the Gossip fanout f to $f + 1$ for SLA$_1$ transactions. If the transactions' remaining deadlines in the committed blocks are always positive and are larger than a specific threshold (e.g., 5 secs), nodes set f to $f - 1$.

4.2 Enforcing SLA on Sequential Execution

Each Slarm node runs a reliable multicast module to enforce the SLA requirement on sequential execution (i.e., gap-free) of SLA transactions. The default P2P multicast protocols (typically Gossip and flooding) in existing blockchain systems are not designed to achieve the gap-filling task. Specifically, the flooding protocol [26] forwards all newly received transactions on each node to all the node's neighbors, an extremely bandwidth consuming process even for a low throughput blockchain [78]. Worse, flooding has no guarantee of filling gaps for smart contract transactions. In Gossip, when a node receives a transaction for the first time, rather than sending the transaction to all neighbors (flooding), it randomly selects a subset of neighbors and forwards the transaction to them [65]. However, Gossip provides no guarantee for gap-filling.

To achieve the gap-filling task in a P2P network, many reliable multicast protocols are presented [34, 59, 80]. They typically follow a two-phase disseminate-correct protocol, as shown in Figure 5. The first phase is an unreliable Gossip that makes the best-effort attempts to disseminate transactions efficiently. The second phase is an anti-entropy protocol, where each node contacts its peers to exchange both the lost and the latest transactions. Erlay [80] is a reliable multicast protocol and is integrated with Bitcoin [78]. Compared with Bitcoin's flooding multicast, Erlay reduces network consumption by 41%. In our evaluation, reliable multicast protocols have better SLA rates than flooding and Gossip but still show low SLA rates under traffic spikes (Figure 1a).

The low SLA rate of existing reliable multicast protocols under stress is due to their strong reliability guarantee. Existing reliable multicast protocols are mainly designed for MPI, where all nodes are guaranteed to receive all transactions eventually. Therefore, in the correction phase, nodes handle both the *gap-filling* of the lost transactions and the *refreshing* of the latest transactions (Figure 6). However, achieving both of these two tasks is extremely inefficient; it causes high CPU and network resource consumptions, which reduces the SLA rate. For example, Erlay [80] achieves these two tasks by performing the set-reconciliation in the correction phase. Nodes compute set sketches of the received transactions, regularly exchange the sketches with their peers, and transfer the different transactions. Computing the sketch is computationally intensive and is quadratic in the number of different transactions.

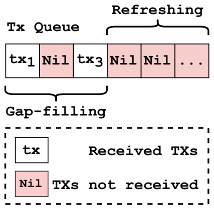


Fig. 6. Gap-filling and Refreshing tasks.

Our main observation to solve this performance degradation under stress is that the strong reliability guarantee of existing reliable multicast protocols is overly restricted for blockchain systems. In a blockchain system, there are many consensus nodes (e.g., all nodes can propose blocks in PoET [42]). If one of these consensus nodes has received a transaction, the transaction can be eventually proposed to the blockchain.

Based on this observation, Slarm performs only gap-filling in the correction phase, as shown in Figure 5. Specifically, each node can easily detect gaps according to the sequence numbers of received transactions, without any communications among nodes. If gaps exist, a node requests for the lost transactions from a random subset of its peers.

This protocol is simple, efficient, yet reliable in enforcing the SLA on sequential execution. It ensures nodes receive gap-free transactions without computing and exchanging the set sketches of received transactions. Although some latest transactions are delayed, however, in our evaluation (500 nodes,fanout=3), Slarm can fix 92.5% of lost transactions, with 42.5% lower CPU usage and 30.7% lower message complexity compared with Erlay [80], which leads to high SLA rates in both normal and high workloads (Figure 1a).

4.3 SLA Enforcement Probability Analysis

Now we analyze Slarm's probability in enforcing deadlines of SLA transactions. Recall Slarm's SLA variables (§3.2): $\tau_d + \tau_c$ and n, where $\tau_d + \tau_c$ is the median commit latency (life cycle) of an SLA transaction in an uncongested P2P network; n is the total number of SLA transactions generated from all clients per second. Suppose n is smaller than the consensus protocol's throughput. We investigate the probability p: the percentage of these SLA transactions that can meet their SLAs ($c \times T$), where $c = 2$ by default.

In SLARM, the SLA transactions are disseminated with a two-phase Gossip-correction (gap-filling) protocol. Slarm first disseminates transactions in Gossip [66], where transactions are propagated to the entire network in rounds. During the propagation rounds of a transaction, each independent node randomly selects a subset of its peers and forwards the transaction to the peers. This random transmission process on each node is a Poisson process [76, 84]. Based on the fact that the superposition of independent Poisson processes is also a Poisson process [2], an SLA transaction's Gossip latency to the entire network satisfies the Poisson distribution.

SLARM adopts a light-weight gap-filling protocol to fix the missed transactions at each hop during Gossip. Because each node only requests for lost transactions from one-hop peers, this gap-filling latency at each hop is stable. Moreover, the total number of hops for an SLA transaction to be disseminated to all P2P nodes is concentrated around its mean value $log(N)$ (in a network of size N). Therefore, the latency caused by gap-filling can be regarded as constant.

An SLA transaction's total dissemination latency is the sum of the Gossip latency and the gap-filling latency. As we have proved, the latency of Gossip transaction dissemination follows the Poisson distribution, and the gap-filling latency is constant, thus the total dissemination latency τ_d of SLARM's SLA transactions also follows the Poisson distribution.

The end-to-end commit latency of a transaction is $\tau_d + \tau_c$. Since the consensus latency τ_c is small (29.0%) and relatively stable (constant), confirmed in our evaluation. Therefore, $\tau_d + \tau_c$ also satisfies Poisson distribution. Suppose the median commit latency of a transaction is T (§3.2), according to Poisson Distribution's probability function $p(x, T) = \frac{e^{-T}T^x}{x!}$, if we set the SLA transaction's deadline as $2 \times T$, 96.0% of these transactions can meet their SLA. This high theoretical rate matches Slarm's actual SLA satisfaction rate in our evaluation.

5 Slarm's Security Design and Analysis

5.1 Design and Security Analysis of Slarm's Trusted Timer

Design of the Fine-grained Trusted Timer. To maintain each SLA transaction's remaining deadline, Slarm nodes rely on a trusted timer to measure the transaction's elapsed time (related time) on each node and during node-to-node transmissions (§4.1.1).

Intel SGX already provides a hardware-protected trusted timer (for short, SGX timer) for enclave programs [39]. Programs running in SGX enclaves can request for the local trusted timestamp by API call `sgx_get_trusted_time()`. The time source of the SGX timer is hardware-protected and transfers the time packets to enclave programs over a secure channel, so the faulty node cannot modify the trusted time packets [30].

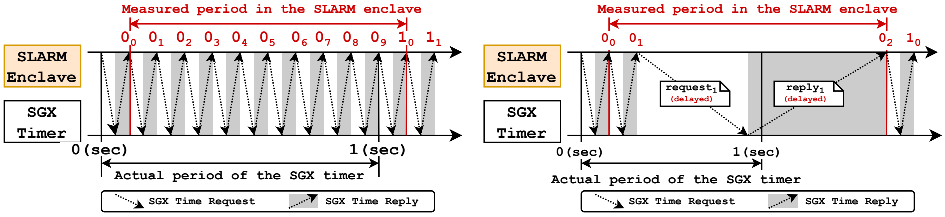
However, due to two limitations, the SGX timer cannot meet the requirements of Slarm. First, the SGX timer is coarse-grained, which provides only sec-level resolution, while the elapsed time of an SLA transaction on a node can be as low as tens of milliseconds. Second, the faulty node can damage the SGX timer's accuracy. For instance, although the node cannot tamper the SGX time source (hardware protected) and the time packet contents (transferred via the secure channel), the node can still delay the time packets, distorting the time for enclave programs (Figure 7). We propose a fine-grained trusted time architecture for Slarm that overcomes these two limitations.

Our fine-grained trusted timer is built upon the hardware-protected SGX timer. We spawn two threads in each node's Slarm enclave: a *timer thread* continuously pulls the hardware-protected SGX timer ticks (sec-level), and a *counting thread* incrementing a variable that counts the CPU cycles within each SGX timer tick (similar to TimeSeal [30]). Nodes calculate the fine-grained trusted timestamp by, i.e., $t = SGXticks + \frac{current\_cpu\_cycles}{total\_cpu\_cycles\_per\_sec}$, where $SGXticks$ is the hardware-protected SGX timer ticks.

We now provide a detailed security analysis about how to handle attacks from the faulty node. The faulty node can conduct only (1) *delay attacks* on the time packets transferred between the SGX timer and the Slarm enclave, (2) *scheduling attacks* on the two timer threads, and (3) *CPU frequency scaling attacks* to reduce the resolution of the counting thread [30, 44, 81].

Detecting delay attack. The node can maliciously delay the time request and reply packets transferred between Slarm's enclave and the SGX timer. As shown in Figure 7a, the *timer thread* continuously pulls the SGX timer; because the SGX timer only provide sec-level resolution, the timer thread obtains ten repeated timestamps of 0 sec ($0_0 \sim 0_9$) before obtaining the first 1 sec timestamp (1_0). If the faulty node delays $request_1$ and $reply_1$ (Figure 7b), the measured period of 0 sec ($0_0 \sim 0_2$) in Slarm's enclave will be much longer than the actual period of SGX timer. As a result, Slarm calculates wrong elapsed time for SLA transactions.

Detection of the delay attacks relies on the intuition that the number of repeated timestamps per SGX timer tick (sec-level) shows large variations under delay attacks. Request latency of the API call `sgx_get_trusted_time()` is stable without attacks [69], so the number of repeated timestamps is also concentrates around its mean value. In Figure 7b, the timer thread only obtains three repeated timestamps ($0_0 \sim 0_2$) of 0 sec under the delay attack, much fewer than the expected value (i.e., 10 repeated timestamps of 0 sec in Figure 7a). If the number of repeated timestamps between two adjacent SGX time ticks is fewer than the expected value, a node can determine itself is under the delay attack and *kick-outs* itself from Slarm's network. When a node tick-out itself from Slarm's network, the node stops processing all incoming messages and multicasts all transactions waiting in the transaction queue. Thus, the node's peers can no longer receive any RTT response from this node, and drop this node from their peer lists.



(a) No delay attacks on the SGX time packets. (b) Delay attacks on the SGX time packets.

Fig. 7. SLARM requests for the hardware-protected SGX time. The time packets can be delayed by the faulty node.

Detecting scheduling attack. The malicious OS can schedule out the *timer thread* and the *counting thread* running in the SGX enclave at any instant. We let each node in SLARM detect this scheduling attack and kick-out itself from SLARM's network when the attacks occur.

If the faulty node schedules out the counting thread, the number of counted CPU cycles per SGX timer tick is reduced, making the counting thread downgrade the trusted timer's resolution to seconds. If the number of counted CPU cycles per second falls below a specified threshold (e.g., 1000 CPU cycles per second to provide millisecond-level resolution), the node kick-outs itself from SLARM's network. If the faulty node schedules out the timer thread (or both of the two threads), the repeated timestamps per second will reduce, which is the same with the delay attack.

Detecting CPU frequency attack. A privileged attacker can manipulate the power management feature of modern Intel CPUs and reduce the CPU frequency. As the SGX timer is backed up with its own battery, the SGX timer's frequency is not affected by the node host's power management [30]. Reducing the CPU frequency leads to fewer CPU cycles per second counted by the counting thread, achieving the same effect as the scheduling attack on the counting thread.

5.2 Design and Security Analysis of SLARM's P2P Protocol

This section presents our security guarantee: SLARM can maintain a high SLA satisfaction rate in §4.3 in the face of attacks (e.g., selective targeted attacks mentioned in §3.3). As SLARM's SLA prioritization mechanism runs in each node's SGX, an attacker can conduct attacks only on messages exchanged between different nodes' enclaves. Specifically, an attacker can corrupt, reorder, drop, and delay messages. Since SLARM's protocol does not rely on message orders to enforce SLA, we do not need to maintain message orders. Since message drop and delay cannot be distinguished in the Internet, we call them deferring attacks and handle them together. The drop and delay attacks can be either selective or random. This section shows how SLARM defends against message corruption attacks, presents a message oblivious mechanism to handle selective deferring attacks and a trustworthy (conservative) RTT mechanism to handle random deferring attacks.

Avoiding message corruption attacks. The metadata in SLA transactions is sensitive. If faulty nodes maliciously access and modify the metadata, they can easily break SLARM's SLA guarantee. For example, faulty nodes can change the *deadline* of an SLA transaction from *5s* to *50s*; then the SLA transaction will not be prioritized even if it is stringent to disseminate, leading to SLA violation. To prevent such attacks, SLARM uses SGX and protects SLA transactions' metadata updates on all SLARM nodes (§4.1). Specifically, we put all the uncommitted transactions in enclaves and pack the scheduler logic as `ECalls` to read incoming transactions, and decide the transactions to be disseminated. In this manner, even if a faulty node runs our scheduler module and tries to poison the metadata, she cannot access the metadata and control the disseminated messages because of the shield of SGX.

Handling selective deferring attacks. In SLARM, faulty nodes can selectively defer P2P messages and easily violate the SLA of many clients' SLA transactions (e.g., the selective targeted attacks mentioned in §3.3). If SLARM's protocol messages in the network have different sizes (e.g., if `gapFill` is not larger than 10 bytes while a transaction dissemination message size can reach 250 bytes), this exposes a large attack interface to attackers (§3.3). Thus, we design to make all P2P messages in Slarm oblivious by filling in extra dummy payload and encrypting messages with the same symmetric key. In this way, all messages in the Slarm network are oblivious (250 bytes). A faulty node cannot distinguish whether a message is a dissemination message (either an SLA transaction or non-SLA transaction), a conservative RTT ping-pong message or a `gapFill` message.

Capturing random deferring attacks. However, even if Slarm's P2P messages are oblivious, it is still challenging to meet the SLA requirements of transactions in an asynchronous network. Even if messages are oblivious, faulty nodes might randomly and arbitrarily drop/delay P2P messages, which can greatly defer the transaction's one-hop transmission.

Slarm captures the random deferring attacks by calculating the trustworthy per-node RTT value with a Trustworthy RTT maintaining protocol and uses the RTT value as the transaction's conservative one-hop elapsed time during the transmission between nodes. Since the Internet latency among two peers is asymmetric due to IP routing, Slarm uses a complete RTT value conservatively instead of its half. Within a random time interval (e.g., 5-10s, similar with Ethereum's ping interval), the Slarm enclave on each node N (including a faulty node) encapsulates a `ping` message with the same format as normal transaction dissemination (around 250 bytes) and sends that `ping` message with its latest RTT value to all its peers. When receiving a `ping` request, a peer node encapsulates a reply message within Slarm's local enclave and sends the pong message back to the inquiry node. The node N collects all the pong messages, all `ping` requests' RTT values from all its peers, and selects the *highest* calculation result as the latest RTT value.

For each node, this RTT mechanism is invoked frequently enough to capture random packet deferring attacks and network congestions, as each node has 50 peers in Ethereum, and all its peers also invoke this trustworthy RTT mechanism to exchange their worst-case RTT values at random moments. This mechanism does not increase Ethereum's message complexity either; it is just modified from Ethereum's default ping-pong mechanism.

This protocol provides trustworthy (conservative) RTT values against the random deferring attacks. A faulty node cannot manipulate an RTT message because all messages are decrypted within SGX. Because we select the *highest* RTT value, so an inquiry node will get a statistically worst case of the actual network one-hop delay, including the random delay conducted by faulty nodes, which makes Slarm's SLA update mechanism conservative.

Because faulty nodes cannot distinguish RTT messages and transaction dissemination messages, so the probability of their random deferring attacks deferring only node N's transaction dissemination messages without deferring any RTT around N is almost zero when the number of nodes is large. Therefore, Slarm's trustworthy RTT mechanism can capture such random attacks with a high probability and make SLA transactions more stringent (some transactions' SLA deadlines can be made negative, but still stringent and conservative), as the deadline for each SLA-transaction subtracts a statistically worst-case value of RTT.

Slarm nodes also need to conservatively estimate an SLA transaction tx's trusted local elapsed time on node N, denoted as tx_N^{wait}. As discussed above, although a faulty node can randomly defer transactions outside the enclave, the extra latency caused by this deferring attack is already captured by $N.RTT$. Therefore, node N uses tx's elapsed time in N's Slarm enclave as tx_N^{wait}, which can be measured by the trusted timer described in §5.1.

Overall, two critical variables used by the scheduler (§4), $N.RTT$ and local elapsed time tx_N^{wait}, are both *conservative*: if Slarm nodes infer that an SLA transaction meets the SLA deadline, the transaction actually meets it with high probability, even though the transaction's dissemination has been deferred by some faulty nodes on some hops.

6 Implementation

We implemented Slarm based on the latest Golang version of Ethereum [96] - a fully tested and actively maintained blockchain system. We leveraged Ethereum's P2P library to build Slarm's reliable multicast component and rewrote Ethereum's transaction logic for admitting, verifying, and scheduling SLA transactions. We carefully selected sensitive functions and put these functions into SGX enclaves. Since SGX only provides C/C++ SDKs, we rewrote all sensitive functions in C and used cgo to invoke `ECalls`. We modified 2037 lines of Golang code and implemented the scheduler and reliable multicast component for 1705 lines of C code. For encryption/decryption, we used AES-256, a symmetric key library provided by the SGX SDK. Slarm uses Ethereum's bootstrap nodes for doing SGX attestation [74] for all member nodes' Slarm enclaves. The bootstrap nodes store a list of attested nodes and provide it to each attested node for peer discovery.

7 Evaluation

Config	Cluster	AWS Cloud
# Nodes per-machine/Total	20/500	100/5k or 100/10k
Default consensus protocol	Clique-PoA	Clique-PoA
Node peer number	10	50
Block commit interval	5s	5s
Avg RTT	20ms	200ms
SLA	{16s, Yes}	{32s, Yes}
Workload	Online trading	Online trading
Bandwidth limitation	20Mbps	20Mbps

Table 1. Default evaluation settings (unless specified).

We evaluated Slarm's performance on both our cluster and the AWS cloud [92], with evaluation parameters shown in Table 1. In our cluster, each machine is equipped with 2.60GHZ Intel E3-1280 V6 CPU with SGX, 40Gbps NIC, 64GB memory, and 1TB SSD. On the AWS cloud, we launched 100 m5d.24xlarge instances with 96 vCPUs, 3.6TB SSD, and up to 25Gbps network bandwidth. To simulate commodity network links, we cap the bandwidth for each node to 20Mbps, same as existing study [54]. We ran 100 Slarm nodes on each VM instance, with each Slarm node running in a docker container. All AWS instances are run in the same zone (Ohio). We ran SGX in cluster with actual SGX hardware and in AWS with Intel's SGX simulator, because AWS does not provide SGX hardware. Table 5 shows that Slarm's performance in simulation mode is roughly the same as hardware mode because Slarm's performance is bounded to the network latency.

We evaluated Slarm with PoA [4], PoS [62], and PoET [43] consensus protocols, and Clique PoA [4] is Slarm's default consensus protocol. Because Slarm only decides which transactions are packaged into each block according to transaction SLAs and does not affect the consensus logic, Slarm is compatible with most blockchain consensus protocols (e.g., Aura [1] and IBFT [77]). We choose Clique PoA as the default consensus protocol for two reasons. First, Clique is decoupled from the cryptocurrency and can achieve high performance, which is especially suitable for applications with SLA latency requirements. Second, Clique is widely used and has become the default consensus protocol of Ethereum private network [8] as well as other notable permissioned blockchains [20, 24].

We evaluated Slarm with five P2P multicast protocols, including the Gossip [66], the flooding [26], and three reliable multicast protocols (Erlay [80], Corrected Gossip [59], and Deadline Gossip [53]). Among these reliable multicast protocols, Erlay is the only protocol developed for a blockchain system (i.e., Bitcoin [78]). Corrected Gossip is the most recent reliable multicast protocol. Deadline Gossip is a deadline-aware reliable multicast protocol. We evaluated the flooding protocol by running the original Ethereum [96] (version eth/64). Because all the other baseline multicast protocols are not open-sourced, we implemented them on the codebase of Ethereum.

Applications	SLA Txs	Non-SLA Txs	Traffic Spike	Tx Submission Rate
Mobile carriers [32, 40]	Pre-paid users (30%)	Post-paid users (70%)	×	200 tx/s
Voting [31, 56, 58]	Election management (30%)	Cast votes (70%)	×	1000 tx/s
CDN accounting [27, 57, 93]	Settlements (30%)	Usage data (70%)	200 tx/s SLA txs lasts 5s	200 tx/s
Disease control [5, 6, 99]	High-risk events (50%)	Low-risk events (50%)	×	200 tx/s
Online trading [67, 86]	Realtime trading (70%)	Non-realtime trading (30%)	200 tx/s SLA txs lasts 5s	200 tx/s

Table 2. Slarm's evaluated blockchain applications. Parameters are from the cited blockchain papers on this table. All SLA transactions are written in smart contracts.

We ran five applications with different portions of application-specific SLA transactions (Table 2). We also generated transaction spikes to simulate the real-world workloads.

in the network. Our default benchmark workload is *Online trading* because it has interactive transactions and is prevalent on the Internet. We set the transaction size as 250 bytes. The transaction sizes for baseline systems are either equal to or smaller than that of Slarm.

We define SLA satisfaction rate as the percentage of SLA transactions with positive remaining deadlines (§4.1) when their committed blocks reach their clients. We report throughput as tx/s for both SLA and non-SLA transactions. We define commit latency as the client perceived elapsed time for all committed transactions. Our evaluation focuses on the following questions:

§7.1 How does Slarm meet the transaction's SLA and what is the system's throughput?
§7.2 How resilient is Slarm's SLA and throughput on spikes of SLA transactions?
§7.3 How resilient is the SLA and throughput to node failures in Slarm?
§7.4 How is the SLA and throughput for five applications in Slarm?
§7.5 What are the limitations and potential future works of Slarm?

7.1 SLA and Performance

End-to-end SLA and performance. Figure 1a shows all systems' SLA satisfaction rate for SLA transactions (70% of all transactions in Slarm; 100% in other systems, because they cannot distinguish SLA transactions) on AWS. Figure 1b shows the throughput of all systems for all transactions on AWS. From 0 to 8s, the network launched 200 clients to generate 200 tx/s totally (to make the PoA consensus protocol reach its peak throughput), and 70% of them were SLA transactions. On 8s, we generated a traffic spike of 200 tx/s additional SLA transactions for five seconds. Overall, Slarm achieves the highest SLA satisfaction rate for SLA transactions and dropped from 98.1% to 82.1% at about 15s then quickly recovered. The other systems' SLA satisfaction rate all dropped significantly and recover much slower than Slarm. Gossip achieved the lowest SLA satisfaction rate because it selects only a subset of peers to disseminate transactions without any gap-filling.

Figure 1b shows that Gossip's throughput is the best, and Slarm incurred roughly 6.9% drop on all transactions' throughput compared to Gossip. This is because Slarm's reliable multicast performs a light-weight gap-filling upon Gossip (Figure 5). Other reliable multicast protocols and flooding incur much higher communication and computation overhead compared to Gossip.

Breakdown and micro-events. To investigate the reasons for Slarm's high SLA satisfaction rate and its throughput overhead, we collected all systems' mirco-events at the 7s in Table 3 and the same events at the 16s (lowest SLA rate for Slarm) in Table 4. In Table 3, Slarm's mean commit latency ($\tau_d + \tau_c$) is much less than $2 \times T$ = 32s, where T is 16s according to Slarm's commit latency in Table 3. This table explains Slarm's high satisfaction rate of 98.1% before the spike comes at 8s. In Table 4, Slarm incurs a higher burden on disseminating SLA transactions, so its per transaction τ_d increases from 9.9s in Table 3 to 26.8s in Table 4, leading to a decreased SLA satisfaction rate.

However, Slarm's SLA satisfaction rate was still much better than the other three reliable multicast protocols under traffic spikes because (1) though only performing the light-weight gap-filling task (§4.2), Slarm achieved a surprisingly high fix rate (> 90%) for lost SLA transactions;

System	Consensus	Avg P2P	Gap-filling		SLA Tx
	Latency(s) (τ_c)	Latency(s) (τ_d)	num	cost (s)	Miss Rate
Slarm	6.1	9.9	18	1.1	3.1%
Flooding	6.2	6.8	N/A	N/A	0.05%
Gossip	5.9	8.7	N/A	N/A	15%
Corrected	6.1	12.1	20	2.6	1.3%
Erlay	6.3	13.3	21	4.2	1.6%
Deadline	6.2	9.8	19	4.6	3.4%

Table 3. Slarm micro-events before the spike launched at the 8s in Figure 1a on AWS. Miss Rate means the ratio of missed SLA transactions in gaps on consensus nodes.

System	Consensus	Avg P2P	Gap-filling		SLA Tx
	Latency(s) (τ_c)	Latency(s) (τ_d)	num	cost (s)	Miss Rate
Slarm	7.5	24.8	187	6.6	7.4%
Flooding	7.2	33.7	N/A	N/A	11.1%
Gossip	7.8	53.1	N/A	N/A	64.1%
Corrected	8.6	75.3	197	31.4	2.9%
Erlay	7.8	88.3	198	39.8	5.9%
Deadline	8.0	63.7	200	41.6	36.2%

Table 4. Slarm micro-events at the 16s (lowest SLA satisfaction rate) in Figure 1a on AWS. Miss Rate means the ratio of missed SLA transactions in gaps on consensus nodes.

(2) τ_d of reliable multicast protocols increased by at least one order of magnitude, much larger than the SLA deadline $2 \times T = 32s$. The reason that Slarm's τ_d is much lower than all the other systems in Table 4 is two folds: (1) Slarm prioritizes the dissemination of SLA transactions over non-SLA transactions; (2) other systems either incurred high transaction miss rate on consensus nodes (64.1% for Gossip) or incurred high communication overhead in fixing all lost transactions which causes severe P2P network congestion (Erlay's τ_d is 88.3s).

To understand Slarm's SLA prioritization mechanism (§4.1.1), we broke down Slarm's portion of SLA and non-SLA transactions in each committed block in Figure 8, which shows that Slarm's mechanism gave high priority for SLA transactions in the first half of the committed blocks, and the non-SLA transactions took the majority in the second half. This implies that Slarm always schedules SLA transactions first and avoids non-SLA transactions to block SLA transactions' dissemination.

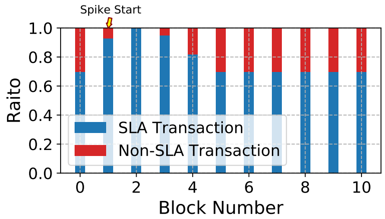
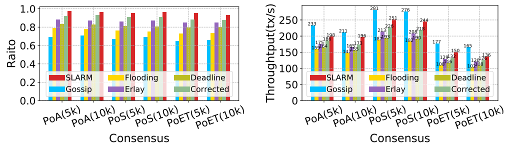


Fig. 8. Distribution of Slarm's SLA and non-SLA transactions in the committed blocks of Figure 1a. Non-SLA transactions are deferred by SLA transactions in Slarm.

Regarding all systems' client perceived latency of SLA transactions, Table 4 can give a good indication. By summing up the τ_d and τ_c columns, Slarm achieved the lowest latency for SLA transactions among all systems since the spike occurred in the 8s. Note that this spike is fair for all systems, because the added 200 tx/s for 5s in Figure 1a were all SLA transactions. For non-SLA transactions, Slarm does sacrifice their commit latency, indicated in Figure 8. Note that the first block in Figure 8 is committed at the 8s in Figure 1a, and Figure 8 shows that, in the first 20 committed blocks (each has a commit latency of 5s in PoA), non-SLA transactions took less than 10%. Since the online trading application has 30% non-SLA transactions, Figure 8 indicates that Slarm greatly sacrifices the commit latency of non-SLA transactions, matching Slarm's design goal of favoring SLA transactions.

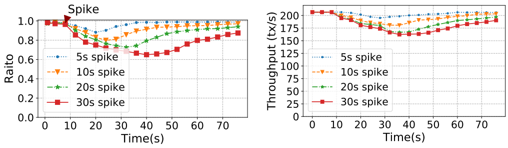
SGX overhead. No current cloud provider can run the SGX hardware on clouds, so we ran Slarm's enclave code using Intel's SGX simulator on AWS. Table 5 shows the micro-events of running Slarm's enclave code on both the SGX simulator on AWS and the SGX hardware of our cluster, which shows similar performance cost. Slarm's time spent in SGX is not the bottleneck of Slarm's performance, and we consider Slarm's performance results reported on AWS would be close to running physical SGX hardware on future clouds (if they can provide SGX hardware).

Scalability. In addition to PoA, we also evaluated Slarm on different P2P network scales (5k and 10k nodes) with two other high-throughput consensus protocols on Ethereum, shown in Figure 9. Overall, Slarm achieves a reasonable SLA satisfaction rate and overhead on throughput (Figure 9b) compare to Ethereum running with default Gossip multicast. Since typical applications (Table 2) are often deployed with no more than 5000 P2P nodes [27], we consider Slarm scalable to the network scale. Figure 1b also suggests that Slarm's throughput is mainly determined by the consensus throughput, because Slarm's gap-filling mechanism incurs low overhead (§4.2).



(a) SLA guarantees. (b) Throughputs.

Fig. 9. SLA guarantees and throughput of PoA [4], PoS [62], and PoET [43] with 5k and 10k P2P nodes.



(a) SLARM's SLA guarantee. (b) SLARM's throughput.

Fig. 10. SLARM's SLA and throughput under different lasting time lengths of transaction spikes.

7.2 Resilience on SLA Transaction Spikes

We also studied all systems' resilience to different degrees of SLA transaction spikes in Figure 10. We started with the same setting as the setting of 0s of Figure 1a, and we varied the lasting time lengths of the 200 tx/s additional SLA transactions from 5s to 30s.

For the 30s spike curve, SLARM's $N.RTT$ among all nodes at the 40s (lowest SLA rate) is 0.6s to 1.3s. The $N.RTT$ value on each SLARM node was larger than the SLA transactions' one-hop dissemination time cost observed on the node. This indicates that SLARM's trustworthy RTT mechanism (§5.2) can capture the random packet delay attacks or network congestions with high probability and make SLARM's SLA deadline update mechanism conservative (§4). Figure 10 shows that, on the 30s of lasting SLA spikes, SLARM' SLA satisfaction rate dropped to as low as 67% and then recovered. This is much better than other P2P multicast protocols in Figure 1a even if the spike of SLA transactions lasts for only 5s.

7.3 Robustness to Node Failures

Hardware failures or DoS attacks can trigger node failures in a P2P network, and such failures are more severe than traffic spikes. We measured SLARM's SLA satisfaction rate and throughput with node failures, as shown in Figure 11. On 8s, we randomly killed 10% of nodes in SLARM's network.

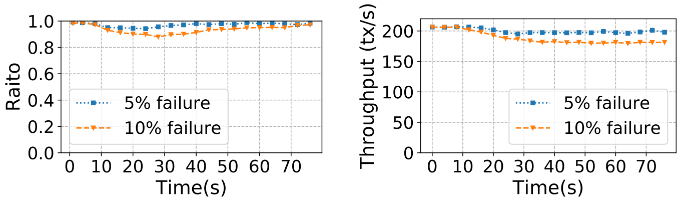
SLARM's SLA satisfaction rate dropped from 98.1% to about 89.2%, and its throughput for all transactions dropped from about 201 tx/s to 170 tx/s. In fact, given a more sparse P2P network, SLARM's SLA prioritization mechanism has to reconnect peers and recompute the conservative RTTs (§5.2) for nodes, keeping the SLA deadline conservative during the dissemination.

7.4 SLA and Performance on Applications

The above evaluation focuses on evaluating the online trading application in different scenarios. We deployed each of the five applications (Table 2) in SLARM individually and measured their SLA rates and throughputs according to their own application settings (e.g., portions of SLA transactions and spikes). Figure 12 shows the results for these applications. SLARM can achieve high SLA rates for different percentages of SLA transactions and can recover fast under traffic spikes (CDN accounting and online trading). The sla rate decline in the voting application is because the SLA transactions submitted by clients per second is always much higher than the system's throughput. Overall, SLARM's SLA mechanisms (§4 and §5) are generic for diverse applications.

SGX	Enc/Dec	No spike	Spike
Cluster	1.8ms	3.4s	20.8s
AWS	2.3ms	4.8s	21.3s

Table 5. SLARM's SGX micro-events in an SLA transaction's entire life cycle. **Enc/Dec** means the corresponding total encryption and decryption time in SGX; **No spike** means the corresponding total wait time of this transaction on all route paths before the spike in Figure 1a. **Spike** means an SLA transaction's corresponding total wait time on all route paths during the spike.



(a) Slarm's SLA guarantee.

(b) Slarm's throughput.

Fig. 11. Slarm SLA and throughput with node failures.

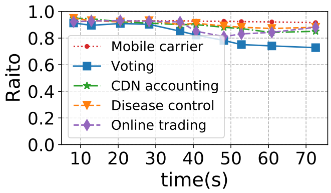


Fig. 12. SLA guarantees for five applications (Table 2).

7.5 Discussion

Limitations. Slarm has two limitations. First, Slarm's transaction scheduling mechanism requires nodes to be equipped with Intel SGX. Nowadays, SGX is prevalent in modern CPUs. More and more blockchain systems are developed with SGX (§8).

Second, Slarm's performance is designed to favor SLA transactions over non-SLA transactions, which may defer the non-SLA transactions. More and more blockchain transactions are submitted in interactively in web browsers or mobile apps. These transactions generally desire a more stringent SLA guarantee, while the non-SLA transactions can tolerate some delay, so Slarm's trade-off is worthwhile.

Implications. Existing research that simulates and analyzes blockchain systems generally focuses on the performance in "stable" workload, where the number of transactions submitted by clients per second does not exceed the system's peek throughput. However, this assumption is insufficient to handle complicated transaction arrival patterns in a production environment, and cannot reflect the actual system QoS in terms of the client-perceived commit latency.

This study adopts a simple but effective metric, i.e., *SLA satisfaction rate* to evaluate the blockchain systems with diverse workloads (Table 2). Our evaluations show that the Gossip [66], which is pervasively used in notable blockchain systems [78, 96], may cause severe performance issues (long commit latency) for blockchain applications implemented with smart contracts.

The main reason is that Gossip only provides the "best-effort" dissemination guarantee and does not perform any gap-filling efforts for lost transactions. Though efficient, Gossip introduces many gaps in each node's per-client received transactions. Because transactions invoking the stateful smart contract must be executed sequentially, the consensus nodes can only commit the transactions after a gap until the gap is fixed by client retry. This leads to extra latency and low SLA satisfaction rates even in the normal workload, confirmed in our evaluation (Figure 1a and Table 4).

To achieve the gap-filling task, reliable multicast protocols [34, 53, 59, 80] are proposed and deployed in blockchain systems [80]. These protocols provide a "strong" dissemination guarantee, which ensures all nodes eventually receive all transactions without gaps.

Our finds suggest that the strong dissemination guarantee is overkilled for blockchain systems for two reasons. First, due to the built-in redundancy in the blockchain's consensus layer, a blockchain system does not require all nodes to receive all transactions. For robustness and fairness, a typical blockchain consensus protocol has many consensus nodes. For instance, in PoET [42] and PoS [9], all P2P nodes are consensus nodes. A transaction can be committed to the blockchain if any consensus node has received the transaction. Second, to obtain the strong dissemination guarantee, existing reliable multicast protocols perform both the gap-filling and the refreshing tasks, which incurs significant computation and communication overhead (§4.2) and causes these protocols to exhibit degraded performance and SLA rates under high workloads (Figure 1a).

Slarm is a promising tradeoff between the efficiency of Gossip and the strong guarantee of reliable multicast protocols. Slarm only performs the light-weight gap-filling task, ensuring nodes receive gap-free transactions without incurring high overhead. Moreover, compared with the reliable multicast protocols that perform both the gap-filling and the refreshing tasks (fix 100% lost transactions), Slarm can fix about 95% lost transactions (§4.2).

Slarm, the first SLA-aware reliable transaction multicast protocol for permissioned blockchains, enables the research community to develop even more exciting blockchain systems and applications with diverse SLA requirements. For instance, Slarm can facilitate the development of new heterogeneous blockchain applications consisting of fine-grained SLA requirements (e.g., a blockchain application consisting of online trading, auction, and clearing transactions). In addition, because Slarm's SLA scheduling mechanism is conservative (§5.2), Slarm can make SLA guided verifications feasible (e.g., trustworthy SLA incentive and auditing mechanisms) and can match the demands of blockchain-driven CDN networks [27] and auditing applications [87]. Last but not least, Slarm may also be generalized to permissionless blockchains [78], because permissionless blockchains prioritize transactions based on transaction fees without any SLA guarantee. We leave these exciting innovations for future works.

8 Related Work

SGX-powered Blockchains. SGX improves diverse aspects of blockchain systems. Intel's PoET [83] and its variant [75] replaces the PoW puzzles with a trusted timer in SGX. REM [100] uses SGX to replace the "useless" PoW puzzles with "useful" computation (e.g., big data). Microsoft CCF [85] (originally named COCO) is a permissioned blockchain platform using SGX to achieve transaction privacy. Hawk [63] and zkLedger [79] focus on enhancing the confidentiality of smart contracts [36]. Ekiden [46] and ShadowEth [98] offload smart contracts' execution to a small group of SGX powered computing nodes to avoid the redundant smart contract executions on all consensus nodes. TeeChain [71] is a payment network and leverages SGX to prevent parties from misbehaving. Slarm is complementary to these SGX blockchain systems and can be integrated into them.

P2P reliable multicast. Existing reliable multicast protocols [34, 53, 59, 80] typically follow the dissemination-correction scheme. Erlay [80] integrates the reliable multicast with Bitcoin [78] to improve Bitcoin's bandwidth efficiency and security. Slarm's protocol differs from these protocols in that Slarm emphasizes achieving an SLA guarantee efficiently by co-designing the P2P and the consensus level (bi-directional).

P2P and SLA. Several SLA provisioning protocols have been proposed for P2P networks [52, 68, 97]. They all focus on one-to-one message routing, while blockchain's transaction delivery requires one-to-all multicast. Chryssis et al. show how to ensure on-time message delivery in Gossip multicast protocol [53]. Unlike Slarm, these protocols do not consider DoS or targeted deferring attacks toward protocol messages, crucial in blockchain deployments.

9 Conclusion

We present Slarm, the first blockchain transaction dissemination protocol that can meet SLAs of diverse applications. Slarm's integration with Ethereum has the potential to attract broad deployments of Internet-wide applications with SLA requirements. Slarm not only greatly improves the user-perceived SLAs of these applications but also improves the system's performance and efficiency. Slarm's source code and evaluation results are released on github.com/sigmetrics21/slarm.

References

[1] [n.d.]. Aura. https://openethereum.github.io/wiki/Aura.

[2] [n.d.]. Thinning and Superposition the Poisson Process. https://www.randomservices.org/random/poisson/Splitting.html.

[3] 2015. Why Scaling Bitcoin With Sharding Is Very Hard. https://petertodd.org/2015/why-scaling-bitcoin-with-sharding-is-very-hard.

[4] 2017. Clique PoA protocol & Rinkeby PoA testnet. https://github.com/ethereum/EIPs/issues/225.

[5] 2018. CDC is Testing Blockchain to Monitor the Country's Health in Real Time. https://www.nextgov.com/emerging-tech/2018/11/cdc-testing-blockchain-monitor-countrys-health-real-time/152622/.

[6] 2018. How IBM and the CDC are testing blockchain to track health issues like the opioid crisis. https://www.fastcompany.com/90231255/how-ibm-and-the-cdc-are-testing-blockchain-to-track-health-issues-like-the-opioid-crisis.

[7] 2018. Introduction to Smart Contracts. https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html.

[8] 2019. Ethereum Private Network. https://geth.ethereum.org/docs/interface/private-network.

[9] 2019. Ethereum's Proof of Stake FAQ. https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ.

[10] 2019. Flooding. https://en.wikipedia.org/wiki/Flooding_(computer_networking).

[11] 2020. Amazon EC2. https://aws.amazon.com/ec2/.

[12] 2020. Bitcoin Mempool Size. https://www.blockchain.com/charts/mempool-size.

[13] 2020. Enterprise Blockchain for Modern Business Networks. https://www.kaleido.io/.

[14] 2020. Enterprise Ethereum Alliance. https://entethalliance.org/.

[15] 2020. Ethereum Mainnet. https://ethereum.org.

[16] 2020. Ethereum Pending Transactions. https://etherscan.io/txsPending.

[17] 2020. Ethereum time drift check. https://github.com/ethereum/go-ethereum/blob/master/p2p/discover/ntp.go.

[18] 2020. Ethereum Time Units. https://solidity.readthedocs.io/en/v0.6.7/units-and-global-variables.html.

[19] 2020. Ethereum Whitepaper. https://ethereum.org/en/whitepaper/.

[20] 2020. Hyperledger Besu Consensus Protocols. https://besu.hyperledger.org/en/stable/Concepts/Consensus-Protocols/Overview-Consensus/.

[21] 2020. Hyperledger Burrow. https://www.hyperledger.org/use/hyperledger-burrow.

[22] 2020. Medicalchain. https://medicalchain.com.

[23] 2020. Mempool Transaction Count. https://www.blockchain.com/charts/mempool-count.

[24] 2020. Quorum. https://consensys.net/quorum/.

[25] 2020. ubiduwin. https://www.ubiduwin.com.

[26] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. 2003. P-Grid: a self-organizing structured P2P system. *ACM SiGMOD Record* 32, 3 (2003), 29–33.

[27] Elif Ak and Berk Canberk. 2019. BCDN: A proof of concept model for blockchain-aided CDN orchestration and routing. *Computer Networks* 161 (2019), 162–171.

[28] Michael P Andersen, John Kolb, Kaifei Chen, Gabriel Fierro, David E Culler, and Raluca Ada Popa. 2017. Wave: A decentralized authorization system for iot via blockchain smart contracts. *University of California at Berkeley, Tech. Rep* (2017).

[29] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[30] Fatima M Anwar, Luis Garcia, Xi Han, and Mani Srivastava. 2019. Securing Time in Untrusted Operating Systems with TimeSeal. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 80–92.

[31] Ahmed Ben Ayed. 2017. A conceptual secure blockchain-based electronic voting system. *International Journal of Network Security & Its Applications* 9, 3 (2017), 01–09.

[32] Jere Backman, Seppo Yrjölä, Kristiina Valtanen, and Olli Mämmelä. 2017. Blockchain network slice broker in 5G: Slice leasing in factory of the future use case. In *2017 Internet of Things Business Models, Users, and Networks*. IEEE, 1–8.

[33] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. 2018. Performance evaluation of the quorum blockchain platform. *arXiv preprint arXiv:1809.03421* (2018).

[34] Kenneth P Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. 1999. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)* 17, 2 (1999), 41–88.

[35] Mic Bowman and Andrea Miele. 2020. Sgx based flow control for distributed ledgers. US Patent App. 16/290,780.

[36] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper. https://github.com/ethereum/wiki/wiki/White-Paper Accessed: 2016-08-22.

[37] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).

[38] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).

[39] Shanwei Cen and Bo Zhang. 2017. Trusted time and monotonic counters with intel software guard extensions platform services. *Online at: https://software. intel. com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services. pdf* (2017).

[40] Abdulla Chaer, Khaled Salah, Claudio Lima, Pratha Pratim Ray, and Tarek Sheltami. 2019. Blockchain for 5G: opportunities and challenges. In *2019 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 1–6.

[41] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: why the system call API is a bad untrusted RPC interface. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 253–264.

[42] Lin Chen, Lei Xu, Nolan Shah, Zhimin Gao, Yang Lu, and Weidong Shi. 2017. On security analysis of proof-of-elapsed-time (poet). In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 282–297.

[43] Lin Chen, Lei Xu, Nolan Shah, Zhimin Gao, Yang Lu, and Weidong Shi. 2017. On security analysis of proof-of-elapsed-time (poet). In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 282–297.

[44] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 7–18.

[45] Xusheng Chen, Shixiong Zhao, Cheng Wang, Haoze Song, Jianyu Jiang, Ji Qi, Tsz On Li, T. H. Hubert Chan, Sen Wang, Gong Zhang, and Heming Cui. 2018. Efficient, DoS-resistant Consensus for Permissioned Blockchains. arXiv:arXiv:1808.02252

[46] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *arXiv preprint arXiv:1804.05141* (2018).

[47] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 185–200.

[48] Christopher D Clack, Vikram A Bakshi, and Lee Braine. 2016. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771* (2016).

[49] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.

[50] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1–12.

[51] P Th Eugster, Rachid Guerraoui, Sidath B Handurukande, Petr Kouznetsov, and A-M Kermarrec. 2003. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)* 21, 4 (2003), 341–374.

[52] Emad Felemban, Chang-Gun Lee, and Eylem Ekici. 2006. MMSPEED: multipath Multi-SPEED protocol for QoS guarantee of reliability and. Timeliness in wireless sensor networks. *IEEE transactions on mobile computing* 5, 6 (2006), 738–754.

[53] Chryssis Georgiou, Seth Gilbert, and Dariusz R Kowalski. 2011. Meeting the deadline: on the complexity of fault-tolerant continuous gossip. *Distributed Computing* 24, 5 (2011), 223–244.

[54] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 51–68.

[55] Katherine Guo, Mark Hayden, Robbert Van Renesse, Werner Vogels, and Kenneth P Birman. 1997. *GSGC: An Efficient Gossip-Style Garbage Collection Scheme for ScalableReliable Multicast.* Technical Report. Cornell University.

[56] Rifa Hanifatunnisa and Budi Rahardjo. 2017. Blockchain based e-voting recording system design. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*. IEEE, 1–6.

[57] Nicolas Herbaut and Nicolas Negru. 2017. A model for collaborative blockchain-based video delivery relying on advanced network services chains. *IEEE Communications Magazine* 55, 9 (2017), 70–76.

[58] Friðrik Þ Hjálmarsson, Gunnlaugur K Hreiðarsson, Mohammad Hamdaqa, and Gísli Hjálmtỳsson. 2018. Blockchain-based e-voting system. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 983–986.

[59] Torsten Hoefler, Amnon Barak, Amnon Shiloh, and Zvi Drezner. 2017. Corrected gossip algorithms for fast reliable broadcast on unreliable systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 357–366.

[60] Yutao Jiao, Ping Wang, Dusit Niyato, and Zehui Xiong. 2018. Social welfare maximization auction in edge computing resource allocation for mobile blockchain. In *2018 IEEE international conference on communications (ICC)*. IEEE, 1–6.

[61] Li-jie Jin, Vijay Machiraju, and Akhil Sahai. 2002. Analysis on service level agreement of web services. *HP June* (2002), 19.

[62] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388.

[63] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Symposium on Security & Privacy*. IEEE. http://eprint.iacr.org/2015/675.pdf

[64] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 711–725.

[65] Joao Leitao, José Pereira, and Luis Rodrigues. 2007. HyParView: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 419–429.

[66] Joao Leitao, José Pereira, and LuÍs Rodrigues. 2010. Gossip-based broadcast. In *Handbook of Peer-to-Peer Networking*. Springer, 831–860.

[67] Zhetao Li, Jiawen Kang, Rong Yu, Dongdong Ye, Qingyong Deng, and Yan Zhang. 2017. Consortium blockchain for secure energy trading in industrial internet of things. *IEEE transactions on industrial informatics* 14, 8 (2017), 3690–3700.

[68] Zhi Li and Prasant Mohapatra. 2004. QRON: QoS-aware routing in overlay networks. *IEEE Journal on Selected Areas in Communications* 22, 1 (2004), 29–40.

[69] Hongliang Liang, Mingyu Li, Qiong Zhang, Yue Yu, Lin Jiang, and Yixiu Chen. 2018. Aurora: Providing trusted system services for enclaves on an untrusted system. *arXiv preprint arXiv:1802.03530* (2018).

[70] Joshua Lind, Ittay Eyal, Florian Kelbert, Oded Naor, Peter Pietzuch, and Emin Gün Sirer. 2017. Teechain: Scalable blockchain payments using trusted execution environments. *arXiv preprint arXiv:1707.05454* (2017).

[71] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gun Sirer, and Peter Pietzuch. 2019. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, New York, NY, USA, 63–79. https://doi.org/10.1145/3341301.3359627

[72] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. 2018. Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network. *IACR Cryptol. ePrint Arch.* 2018 (2018), 236.

[73] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. {BITE}: Bitcoin Lightweight Client Privacy using Trusted Execution. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 783–800.

[74] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *HASP @ ISCA* 10 (2013).

[75] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. 2016. Proof of Luck: An Efficient Blockchain Consensus Protocol. In *SysTEX '16 Proceedings of the 1st Workshop on System Software for Trusted Execution* (Trento, Italy). ACM, 2:1–2:6. http://eprint.iacr.org/2017/249.pdf

[76] Yves Mocquard, Bruno Sericola, and Emmanuelle Anceaume. 2019. Probabilistic Analysis of Rumor-Spreading Time. *INFORMS Journal on Computing* (2019).

[77] Henrique Moniz. 2020. The Istanbul BFT Consensus Algorithm. *arXiv preprint arXiv:2002.03613* (2020).

[78] Satoshi Nakamoto. 2019. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report. Manubot.

[79] Neha Narula, Willy Vasquez, and Madars Virza. [n.d.]. zkLedger: Privacy-Preserving Auditing for Distributed Ledgers. *auditing* 17, 34 ([n. d.]), 42.

[80] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. 2019. Erlay: Efficient Transaction Relay for Bitcoin. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 817–831.

[81] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting {SGX} enclaves from practical side-channel attacks. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 227–240.

[82] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. 2009. Service level agreement in cloud computing. (2009).

[83] Giulio Prisco. 2016. Intel develops âĂŸSawtooth LakeâĂŹdistributed ledger technology for the Hyperledger project. *Bitcoin Magazine* (2016).

[84] S Sundhar Ram, A Nedić, and Venugopal V Veeravalli. 2009. Asynchronous gossip algorithms for stochastic optimization. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. IEEE, 3581–3586.

[85] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, CÃČÂľdric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. 2019. *CCF: A Framework for Building Confidential Verifiable Replicated Services.* Technical Report MSR-TR-2019-16. Microsoft. https://www.microsoft.com/en-us/research/publication/ccf-a-framework-for-building-confidential-verifiable-replicated-services/

[86] Moein Sabounchi and Jin Wei. 2017. Towards resilient networked microgrids: Blockchain-enabled peer-to-peer electricity trading mechanism. In *2017 IEEE Conference on Energy Internet and Energy System Integration (EI2)*. IEEE, 1–5.

[87] Bruce Schneier and John M Kelsey. 1999. Event auditing system. US Patent 5,978,475.

[88] Alberto Sonnino, Michał Król, Argyrios G Tasiopoulos, and Ioannis Psaras. 2019. AStERISK: Auction-based Shared Economy ResolutIon System for blocKchain. *arXiv preprint arXiv:1901.07824* (2019).

[89] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 309–324.

[90] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 423–440.

[91] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. 1998. A gossip-style failure detection service. In *Middle-wareâĂŹ98*. Springer, 55–70.

[92] Jinesh Varia. 2010. Migrating your existing applications to the aws cloud. *A Phase-driven Approach to Cloud Migration* (2010).

[93] Thang X Vu, Symeon Chatzinotas, and Björn Ottersten. 2019. Blockchain-based Content Delivery Networks: Content Transparency Meets User Privacy. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1–6.

[94] Matthew Walck, Ke Wang, and Hyong S Kim. 2019. TendrilStaller: Block Delay Attack in Bitcoin. In *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 1–9.

[95] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[96] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[97] Hao Yang, Minkyong Kim, Kyriakos Karenos, Fan Ye, and Hui Lei. 2009. Message-oriented middleware with QoS awareness. In *Service-Oriented Computing*. Springer, 331–345.

[98] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. 2018. ShadowEth: Private Smart Contract on Public Blockchain. *Journal of Computer Science and Technology* 33, 3 (2018), 542–556.

[99] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. 2016. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. *Journal of medical systems* 40, 10 (2016), 218.

[100] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert van Renesse. 2017. REM: Resource-Efficient Mining for Blockchains. http://eprint.iacr.org/2017/179. http://eprint.iacr.org/2017/179.pdf Accessed: 2017-03-24.