2020 International Symposium on Reliable Distributed Systems (SRDS)

Intrusion-Tolerant and Confidentiality-Preserving Publish/Subscribe Messaging

Sisi Duan, Chao Liu, Xin Wang, Yusen Wu, Shuai Xu, Yelena Yesha, Haibin Zhang
University of Maryland, Baltimore County

***Abstract*—We present Chios, an intrusion-tolerant publish/subscribe system which protects against Byzantine failures. Chios is the first publish/subscribe system achieving decentralized confidentiality with fine-grained access control and strong publication order guarantees. This is in contrast to existing publish/subscribe systems achieving much weaker security and reliability properties.**

Chios is flexible and modular, consisting of four fully-fledged publish/subscribe configurations (each designed to meet different goals). We have deployed and evaluated our system on Amazon EC2. We compare Chios with various publish/subscribe systems. Chios is as efficient as an unreplicated, single-broker publish/subscribe implementation, only marginally slower than Kafka and Kafka with passive replication, and at least an order of magnitude faster than all Hyperledger Fabric modules and publish/subscribe systems using Fabric.

I. INTRODUCTION

Publish/Subscribe (pub/sub) is a popular messaging pattern allowing disseminating information from publishers to different subsets of interested subscribers via an overlay of brokers (servers). Publishers advertise information to the brokers and send publications as advertised. Subscribers express their interests for receiving a subset of publications by issuing subscriptions to brokers. Upon receiving publications from publishers matching the interests of subscribers, brokers send the corresponding publications to the interested subscribers.

One distinguishing feature of a pub/sub system is that it decouples publishers and subscribers in both time and space: publishers and subscribers do not need to know or synchronize with one another. This feature enables both system flexibility and scalability. Pub/Sub systems are widely used in practice, such as Amazon SNS [4], AMQP [56], Apache Kafka [7], FAYE [25], Google Cloud Pub/Sub [30], and MQTT [44]. Pub/Sub serves as the core middleware for numerous applications, e.g., data collection and analysis, Internet-of-Things (IoT), network management and monitoring, streaming services.

Despite their popularity, existing pub/sub systems (built in both industry and academia) suffer reliability and confidentiality problems. Let us illustrate the issues with a health record exchange pub/sub system [31], [24], where the actors include patients and providers (physicians, hospitals, pharmacists), both of which can be publishers and subscribers. Publications may be medical files (e.g., reports, X-ray images) sent from patients or providers to patients or providers. Publications may also be new drug information and updates about the availability of facilities sent from providers to patients. For instance, an emergency unit receives a patient in critical conditions and disseminates the patient medical files as a publisher to various hospital units, while the hospital units may submit subscriptions (e.g., specialties, qualifications, schedule for patient admission and treatment sessions). As another example, a new-born is identified by a hospital for a rare dermatology disease. The hospital represents the new-born and the parents to send the medical images to some local expert dermatologists for timely treatment.

Consider another example of a market report notification system, where publishers are private sectors publishing paid market reports, and subscribers are investors who receive reports according to their interests (e.g., reports for certain categories, reports for specific periods). The brokers match publications with the interests and send the publications to interested (and paid) investors.

With the examples in mind, we now discuss the challenges of building intrusion-tolerant pub/sub systems.

Confidentiality and fine-grained access control (Or: Two-way information control). Publications in both examples (health records, private market reports) need to be confidentiality-protected. In fact, confidentiality in pub/sub is strongly tied to *access control*, a process by which subscribers are granted access to certain publications based upon certain rules. The middleware community has long been expecting pub/sub systems where publishers can define by whom and how their data can be accessed, preferably not just role-based but also attribute-based.

For instance, the "ideal" situation for health record exchange is that publishers (patients and providers on behalf of patients) can decide by whom, when, and how their health records can be viewed or used. Patients should be able to decide which doctors can see their records, either exactly (by name), or those that meet certain criteria (e.g., "D.C. doctors", "more than 15-year practice in dermatology", "no malpractice history") [19]. For the market report example, publishers may enforce access control based on subscribers' qualifications, attributes, and if subscribers paid for the service (to the brokers).

Confidentiality-preserving pub/sub with fine-grained access control *enhances* the conventional pub/sub systems with *two-way* information control. In conventional pub/sub systems, subscribers can filter the information via subscriptions, but publishers cannot control who can receive the publications. The one-way information control is undesirable for applications such as cross-domain pub/sub systems where publications need to be protected (as shown in the health exchange

2575-8462/20/$31.00 ©2020 IEEE
DOI 10.1109/SRDS51746.2020.00039

and market report examples), most pub/sub systems in private corporate networks, and any IoT and big data applications where individual user data are sensitive.

Achieving the goal *securely*, however, is difficult. Existing pub/sub systems with confidentiality or access control either rely on non-cryptographic trusted domains (an overly strong assumption), centralized architectures, and/or violate the decoupling feature of pub/sub systems [31], [32], [33], [54], [8], [61], [40], [27], [58], [53]. Building a decentralized pub/sub system with fine-grained access control is deemed to be a major open problem [47]. First, the approach to encrypting publications using the keys of subscribers does not work, because, due to the decoupling feature of pub/sub systems, publishers do not know the identities or keys of subscribers. Second, publishers cannot encrypt the data using the keys of brokers either, as brokers would know the publications in plaintext. Even if a single broker is compromised, all historical publications will be leaked.

Reliability. Another challenge of building intrusion-tolerant pub/sub systems is reliability under Byzantine (arbitrary) failures. Existing reliable pub/sub systems [20], [36], [38], [39], [50] only achieve weak reliability notions. One particular reliability notion is *publication total order* which guarantees subscribers should receive relevant publications in the same order. For instance, in the stock market, seeing a high price followed by a low price means something very different from seeing a low price followed by a high price; it is vital to ensure that all subscribers receive the price information in the same order. Publication total order would be easy to achieve if brokers use Byzantine fault-tolerant (BFT) state machine replication to maintain a total order of publications and ask subscribers to deliver publications according to the total order.

Even so, due to the two-way control (publication filtering via subscriber interests and publisher access control), not *all* publications will be sent to all subscribers. Therefore, the approach that brokers maintain a total order fail to work. In particular, subscribers do not know if they should wait for or skip publications with certain sequence numbers, as subscribers do not know if the corresponding publications are on the way or will never arrive.

Discussion. With the rise of blockchains, two pub/sub systems using blockchains (Hyperpubsub [62] and Trinity [50]) were proposed to defend against (Byzantine) failures. Both systems make a black-box usage of existing pub/sub systems and blockchain systems. Hyperpubsub uses Apache Kafka and Hyperledger Fabric [6], while Trinity combines MQTT and one of the four blockchains (Fabric, Tendermint [55], and test networks for Ethereum [57] and IOTA [34]). The two systems, however, suffer from at least three problems. First, both systems are essentially auditing systems using blockchains. The overall systems are not Byzantine fault-tolerant, as neither Kafka (only partially crash fault-tolerant) or MQTT (not fault-tolerant) can defend against Byzantine failures. Both liveness and safety are violated if any brokers of the two systems are compromised. Second, both Hyperpub and Trinity leveraging fully-fledged blockchains have demonstrated poor performance, because blockchains are essentially storage systems not designed for pub/sub systems, and many features of blockchains are not needed for pub/sub systems. Third, both Hyperpubsub and Trinity directly combine existing pub/sub and blockchains systems and therefore require a much larger number of nodes and resources than a blockchain system or a conventional pub/sub system.

Neither Hyperpubsub nor Trinity achieves confidentiality or publication total order, two goals we aim to address in this paper.

Our contribution. We design, implement, and evaluate Chios, a Byzantine fault-tolerant (BFT) pub/sub system with fine-grained access control and strong reliability, without sacrificing the decoupling property of pub/sub. Chios's security assumption is standard to BFT and threshold cryptography, i.e., an adversary cannot corrupt more than 1/3 of the total brokers. We summarize our contribution in the following:

- We formally define the properties of a BFT and confidentiality-preserving pub/sub system, covering strong access control and message ordering guarantees, in the sense of cryptography and reliable distributed systems.
- We demonstrate Chios is provably secure under our definitions by devising and extending cryptographic and reliable distributed system protocols (e.g., vector-label-input threshold encryption, broadcast encryption with decentralized key distribution). Chios is the first pub/sub system achieving decentralized and fine-grained access control as well as publication total order. We compare Chios with existing pub/sub systems in Table I.
- Chios is versatile and modular, supporting three additional and fully-fledged pub/sub instances designed to meet different goals (e.g., different performance metrics, different application scenarios). This includes an instance that combines threshold encryption and broadcast encryption to enable more efficient and dynamic access control. For the instance, we also provide an optimized instantiation that is more efficient than a trivial instantiation. Both the general protocol and the instantiation use a novel approach to maintain the decoupling property of pub/sub.
- We implement and evaluate Chios, showing that all its variants are nearly as efficient as its unreliable (unreplicated) counterpart and existing pub/sub systems (Kafka and Kafka with passive replication) and orders of magnitude faster than blockchain-based systems (Fabric, Trinity, Hyperpubsub). None of existing pub/sub systems or blockchain-based systems achieve decentralized confidentiality or strong order guarantees.

II. RELATED WORK

Fault-tolerant pub/sub. Most of industry pub/sub systems (Apache Kafka [7], FAYE [25], Google Cloud Pub/Sub [30], and MQTT [44]) do not have strong fault tolerance guarantees. For instance, Kafka is crash fault-tolerant for its controller part. For its broker components, most Kafka implementations are not fault-tolerant, though Kafka can be configured to use passive replication for weak fault tolerance. Pub/Sub systems

systems	brief description	Byzantine publisher	Byzantine broker	confidentiality and access control	publication total order	publication liveness
Chios	BFT and confidentiality-preserving	●	●	decentralized; attribute-based	●	●
Kafka [7]	favor performance over reliability	○	○	○	○	○
AMQP [56]	"pub/sub for business"	○	○	use trusted virtual host; password for access control	○	○
Hyperpubsub [62]	auditing system for Kafka	○	○	○	○	○
Trinity [50]	auditing system for MQTT	○	○	○	○	○
P2S [20]	crash fault-tolerant	○	○	○	◐	●
PubliyPrime [39]	Byzantine failure detection	○	●	○	◐	●
JM [36]	deconstructing BFT using a large number of nodes	●	●	○	○	●
IRC [33]	access control using ABE	●	●	centralized authority needed; expensive pairing-based crypto	○	○
EventGuard [53]	use trusted components	○	●	trusted nodes for confidentiality	○	○

Table I

CHARACTERISTICS OF REPRESENTATIVE PUB/SUB PROTOCOLS. ◐DENOTES PARTIAL SUPPORT. P2S AND PUBIYPRIME ACHIEVE WEAKER ORDERING GUARANTEES THAN PUBLICATION TOTAL ORDER. (THE FORMAL DEFINITIONS OF PUBLICATION TOTAL ORDER AND PUBLICATION LIVENESS ARE IN SEC. III-A.)

with strong reliability have been mostly studied for the case of crash failures [38], [60], [20]. Only a handful of works consider a weaker subset of Byzantine failures [36], [39] and none of them achieve publication total order. Besides, PubliyPrime [39] does not handle Byzantine publishers or subscribers.

Pub/Sub with payload confidentiality. Confidentiality in pub/sub systems can be generally divided into two categories [47]: 1) confidentiality for publication headers and subscription constraints; 2) payload confidentiality (the ability to hide the payload of the publications, e.g., the patient health record). The confidentiality issue has become a major obstacle to wider adoption of pub/sub systems [47].

Chios addresses payload confidentiality but not confidentiality for publication headers or subscription constraints. Most prior pub/sub systems that handle payload confidentiality rely on overly strong "trusted domain" assumptions and do not maintain the decoupling feature of pub/sub systems that is essential to pub/sub system flexibility and scalability [8], [61], [40], [27], [58]. Srivatsa and Liu [53] devised EventGuard with many goals similar to ours. EventGuard, however, assumes a trusted service for confidentiality and authenticity.

Pub/Sub with access control (but no fault tolerance). While there are a number of pub/sub systems [31], [32], [33], [54] that use attribute-based encryption (ABE) [12] to achieve fine-grained access control, they all suffer from the following problems: 1) Efficient ABE schemes rely on relatively slow pairing-based cryptography. 2) All these systems use a trusted central authority which is a single point of failure. While the so-called *decentralized ABE* schemes exist [43], decentralization here actually means that *anyone* can serve as an ABE authority by creating a public key and issuing private keys to different users, but it does not mean that the keys are generated interactively among distributed nodes.

Reliable distributed systems with confidentiality. Several works achieve confidentiality in distributed file or storage systems that support *store* and *retrieve* operations [2], [28], [35], [42], [17], [10], [48]. In these systems, clients apply encryption, or secret sharing, to the data before the data is uploaded to the system. Notably, Depspace [10] explores how to use publicly verifiable secret sharing and hash function to encrypt and locate client data, but it does not achieve linearizability. AVID [17] suggests the use of threshold encryption to provide access control for Byzantine reliable broadcast and asynchronous verifiable information dispersal. AVID, however, considers a much simpler setting and does not have an implementation.

Yin et al. [59] built a BFT protocol which privately processes user data by separating agreement from execution and using threshold signatures. Assuming the same architecture, Duan and Zhang [23] provided a more efficient construction that uses only symmetric encryption. Both protocols require a lot more nodes than a conventional BFT protocol.

Many recent works [18], [14], [41] explore how to perform private computation on blockchains using trusted execution environments (TEEs), e.g., Intel SGX. These systems require trusting a single TEE vendor (e.g., Intel). Some cryptographic proposals use zkSNARKs [9] or multi-party computation [21] to achieve private computation. These approaches are limited in practice, as the cost to deal with *generic* operations is very high, and the throughput is low.

III. System and Threat Model

Background on pub/sub systems. Pub/Sub systems enable disseminating information from publishers (information sources) to subscribers (interested recipients) via an overlay of brokers (servers). Publishers advertise information to the brokers and send publications as advertised. Subscribers express their interests for receiving a subset of publications by issuing subscriptions. Brokers store subscriptions received from subscribers. Upon receiving matching publications from publishers, brokers send the corresponding publications to the interested subscribers. Besides storing subscriptions, brokers may maintain routing tables to deliver subscribers information.

The communication between publishers and subscribers is decoupled both in time and space. In particular, publishers and subscribers do not need to know or synchronize with one another. Indeed, direct communication among end-customers may not be possible. The decoupling feature enables flexible and scalable information exchange and also avoids maintenance and charging difficulties for end-customers. Moreover, this allows anonymity between publishers and subscribers (assuming brokers are correct).

This paper considers *topic-based* pub/sub, which is dominant in industry pub/sub systems (e.g., Kafka, FAYE, MQTT, Amazon SNS, Google Cloud Pub/Sub). In topic-based pub/sub, a publication includes a *header* and a *payload*. The header contains the *topics* and their values (e.g., ID = "Alice", county = "Orange", price = "105"), while the payload contains the complete bulk data. Correspondingly, a subscription includes a set of *constraints* on the topics (e.g., ID = "Alice", county = "Franklin" or "Orange", price = 100). The brokers need to match publications against stored subscriptions according to the constraints of the topics ("equation," "and," "or" for topic-based pub/sub).

BFT. We consider BFT state machine replication (SMR) protocols, where f out of n replicas may fail arbitrarily (Byzantine failures) and a computationally bounded adversary can coordinate faulty replicas. A replica *delivers operations*, each *submitted* by some client. The client should be able to compute a final response to its submitted operation from the responses it receives from replicas.

A. Formalizing BFT Pub/Sub

Syntax. In our setting, publishers and subscribers are clients. Publishers can be subscribers and vice versa. We use brokers, servers, and replicas interchangeably. We consider an overlay network, where brokers are connected in a complete graph.

A BFT pub/sub system consists of the following (possibly interactive) operations (reg, advertise, sub, pub, notify, read). An interactive registration algorithm reg is run by clients and brokers. Through the reg algorithm, new clients can be registered in the system and brokers can verify and store client (access) attributes (e.g., ages, certificates) enabling them to have access to publications in the future. For instance, a publisher may want only clients with certain attributes to see its publications. Clients should be able to register independently, and in particular, potential publishers and subscribers need not know one another. A client may not need to decide at this stage if the client would like to register as a publisher, a subscriber, or both, but rather may do this later via advertise and sub.

Publishers advertise to the replicas information that will be sent to all or a subset of clients. The advertise messages may be viewed as special publications. Subscribers send brokers subscriptions to express their interests via a sub operation. Brokers store subscriptions received from subscribers. Upon receiving matching publications from publishers via a pub operation, brokers send the corresponding publications to the interested subscribers via a notify operation. The read operation is similar to that of popular pub/sub systems (e.g., Kafka) and allows a client to read particular data of interests from brokers.

Operations (reg, advertise, sub, pub) change broker state and are collectively called write operations. Operations (notify, read) do not change broker state.

In our system, a publisher can send an encrypted publication together with access control rules ac to the system. We say a subscriber (a client) is *authorized* to see a publication m, if the publisher submitting m has listed the subscriber in its access control rules ac.

Goals. The goal of our secure BFT pub/sub system is to achieve CIA (confidentiality, integrity, availability) against malicious brokers, publishers, and subscribers. As in a BFT system, we assume a strong adversary that can passively corrupt f out of n replicas and adaptively corrupt an unbounded number of clients. We divide the goals into confidentiality and reliability goals.

Confidentiality and access control goals. We provide a unified definition of security covering all confidentiality aspects (access control as specified by data providers and confidentiality for non-subscribers and brokers). Specifically, given a BFT pub/sub system, we associate the following game to an adversary $\mathcal{A}$ in Fig. 1.

- $\mathcal{A}$ chooses to corrupt a fixed set of f brokers.
- $\mathcal{A}$ interacts with honest parties arbitrarily and chooses to corrupt clients adaptively.
- $\mathcal{A}$ selects two messages m_0 and m_1, an ac, and a unique tag tid that specifies an instance, and submits them to the encryption oracle for the system. $\mathcal{A}$ cannot corrupt any clients specified by ac (otherwise, $\mathcal{A}$ would have trivially won the game). The oracle randomly selects a bit b and computes an encryption c of m_b with ac and tid, and sends the ciphertext to $\mathcal{A}$.
- $\mathcal{A}$ interacts with honest parties arbitrarily subject only to the following two conditions that 1) $\mathcal{A}$ cannot ask the decryption oracle for the ciphertext c with ac and tid, and 2) $\mathcal{A}$ cannot corrupt any clients specified by ac.
- Finally, $\mathcal{A}$ outputs a bit b'.

Figure 1. We define the advantage of the adversary $\mathcal{A}$ to be the absolute difference between 1/2 and the probability that $b' = b$.

Note it is easy to have a unique tid for a client operation (e.g., using a concatenation of the client identity cid and the timestamp of the operation ts). We comment that we do not need to additionally define decryption consistency (as in threshold encryption), as this is captured by Agreement 2 of the reliability goals (introduced below).

Our definition is easily shown to imply input causality (causal order) [51], which prevents the faulty replicas from creating an operation derived from a correct client's but that is delivered (and so executed) before the operation from which it is derived. The problem of preserving input causality was introduced in BFT atomic broadcast protocols by Reiter and Birman [51], later refined by Cachin et al. [15], and recently generalized by Duan et al. [22]. Preserving causal order equally makes sense in BFT pub/sub systems.

We do not aim to achieve confidentiality on publication headers or subscription constraints, although they need to be protected for some applications.

Reliability goals. We have the following reliability goals:

- **Agreement 1**: If any correct replica delivers a write operation m, then every correct replica delivers m.
- **Agreement 2**: If any correct subscriber delivers a publication p matching its subscription T, then every correct subscriber who has the same subscription T and has access to p delivers p.
- **Total Order 1**: If a correct replica has delivered write operations $m_1, m_2, \cdots, m_s$ and another correct replica has delivered $m'_1, m'_2, \cdots, m'_{s'}$, then $m_i = m'_i$ for $1 \leq i \leq min(s, s')$.
- **Total Order 2 (Publication total order)**: If a correct subscriber has delivered $p_1, p_2, \cdots, p_s$ for a subscription T and another correct subscriber has delivered $p'_1, p'_2, \cdots, p'_{s'}$ for T, and if the two subscribers have the same access attributes, then $p_i = p'_i$ for $1 \leq i \leq min(s, s')$.
- **Liveness 1**: If a write operation m is submitted to $n - f$ correct replicas, then all correct replicas will eventually deliver m.
- **Liveness 2 (Publication liveness)**: If a publisher is correct and submits p matching a subscription T, then all correct subscribers that issued a subscription T and have access to p will eventually deliver p. If a subscriber issues a subscription T, then it will deliver all authorized publications matching T.
- **No Creation**: If a subscriber delivers a publication, then the publication was published by some publisher.
- **No Duplication**: A subscriber delivers no publications twice.

Agreement 1, Total Order 1, and Liveness 1 are properties for all write operations. The other properties are ones for pub/sub operations with respect to subscribers. We have considered access control when defining these properties. The properties can be easily simplified to work without considering access control.

Prior formalization on reliable pub/sub systems [60], [20], [50], [39] only consider a much smaller subset of properties we defined here. In particular, a weaker notion of publication total order was considered in several systems [60], [20], [50], where neither subscription restraints nor access control rules are considered, and total order is enforced among all publications across all subscribers. The weaker notion is immediately implied by the total order property of brokers (Total Order 1), as subscribers can directly deliver publications in the sequence number order determined by brokers. Moreover, in [60], Total Order 1 is not required, because they did not use a state machine replication approach.

No Creation and No Duplication have been previously formalized by Jehl and Meling [36] but with different names ("authentication" and "uniqueness").

IV. THE CHIOS SYSTEM

Chios addresses two important problems in pub/sub systems, achieving decentralized, privacy-preserving pub/sub with fine-grained access control, and ensuring publication total order even with the two-way information control.

We first review threshold encryption. Then, we describe a toy protocol achieving all security goals except publication total order. Finally, we show our core protocol (Chios) achieving publication total order.

A. Review of VIL Threshold Encryption

Conventional labeled threshold encryption takes a single string as the label. We extend the primitive to support a vector of strings $L = (L_1, \cdots, L_s) \in \{0,1\}^{**}$ as labels. By a vector we mean a sequence of zero or more strings, and we let $\{0,1\}^{**}$ denote the space of all vectors. Our scheme supports an *arbitrary* number of vectors, each of which can be of *arbitrary* length.

Syntactically, a robust (t, n) VIL (variable-input-length) threshold encryption consists of the following algorithms. A probabilistic key generation algorithm TGen takes as input a security parameter l, the number n of total servers, and threshold parameter t, and outputs (pk, vk, sk), where pk is the public key, vk is the verification key, and $sk = (sk_1, \cdots, sk_n)$ is a list of private keys. A probabilistic encryption algorithm TEnc takes as input a public key pk, a message m, and a vector label L, and outputs a ciphertext c. A probabilistic decryption share generation algorithm $\mathsf{ShareDec}$ takes as input a private key sk_i, a ciphertext c, and a label L, and outputs a decryption share τ. A deterministic share verification algorithm Vrf takes as input the verification key vk, a ciphertext c, a label L, and a decryption share τ, and outputs $b \in \{0,1\}$. A deterministic combination algorithm Comb takes as input the verification key vk, a ciphertext c, a label L, a set of t decryption shares, and outputs a message m, or $\perp$ (a distinguished symbol).

Our VIL threshold encryption scheme, TDH2-VIL, extends the TDH2 threshold encryption by Shoup and Gennaro [52].

B. A Toy Protocol: Chios without Publication Total Order

System setup. We assume that the number of brokers is n, and f out of n brokers can fail arbitrarily (Byzantine failures). We set up an $(f + 1, n)$ VIL threshold encryption (TGen, TEnc, $\mathsf{ShareDec}$, Vrf, Comb) so that a public key pk and verification keys vk are associated with the system, while a secret key is shared among all brokers, with a broker i having a key sk_i for $i \in [1..n]$.

Publisher and subscriber registration. In Chios, communication among publishers and subscribers is decoupled both in time and space. Publishers and subscribers do not need to know or synchronize with one another. A client (a publisher or a subscriber) registers with brokers using their attributes. During the registration, the brokers collectively verify and store client attributes. Chios runs BFT to ensure the registration information is consistent among brokers. More specifically:

- A client sends its attributes and the corresponding proof to brokers as a special registration operation.

- Upon receiving a registration operation, brokers verify the correctness of client attributes. Brokers discard the operation if the verification fails. (Note the verification of the client attributes can be done offline or online, as in PKI registration.) Brokers run the BFT protocol to assign a sequence number to the registration operation and store the operation in sequence number order. Brokers send replies signaling the success of registration.
- Upon receiving $f+1$ matching replies, the client completes the registration.

Advertisements and subscriptions. During the advertisement process, publishers advertise to the system their publication scopes, and the brokers broadcast the type of events to all potential subscribers (who show an intent to receive subscriptions during the registration process or later via subscriptions). The advertise operation can be viewed as a special pub operation. During the subscription process, subscribers submit their subscriptions which are stored at the brokers. Advertisements and subscriptions are treated as BFT write operations that need to be ordered.

Publishing (with confidentiality and fine-grained access control). Let ts, op, o, $hr = [hr_1..hr_s]$, $\mathsf{ac} = [\mathsf{ac}_1..\mathsf{ac}_t]$, and p be the timestamp, the operation type (pub), the executable operation o (which makes Chios stateful), the header, the access control policies, and the payload of a publication, respectively. The header hr consists of the topics of a publication and optionally additional associated-data that do not need to be privacy-protected. The approach provides fine-grained (per-publication) and attribute-based access control.

- A publisher cid takes as input ts, op, o, hr, p, and ac, and computes a threshold encryption ciphertext as follows. The vector of labels L for the client is of the form $(\mathsf{cid}, \mathsf{ts}, \mathsf{op}, hr, \mathsf{ac})$. The client cid takes as input the threshold encryption public key pk, L, and p, and outputs a labeled ciphertext $(L, c) \overset{\$}{\leftarrow} \mathsf{TEnc}(pk, p, L)$ using our vector-label-input threshold encryption. It sends brokers (L, c) as a BFT write operation.
- Upon receiving a client publication, brokers run the BFT protocol to order the publication (by assigning a sequence number to the publication), store the publication, and execute the associated operation o in sequence number order. The brokers send replies to the write request which may contain the executed result for the publisher.
- Upon receiving $f+1$ matching replies, the client completes the publish operation.

Notify. During the process, brokers enforce access control and send publications to authorized and interested subscribers. More specifically:

- Brokers decide authorized and interested subscribers for a publication (L, c) by matching publication topics with existing subscription constraints, checking access control policies associated with the publication, and checking global access control policies already installed in the brokers. For authorized and interested subscribers, each broker $i \in [1..n]$ sends them its decryption share $\tau_i \overset{\$}{\leftarrow} \mathsf{ShareDec}_{sk_i}(L, c)$ and the sequence number sn assigned to the labeled ciphertext (L, c).
- Upon receiving $f+1$ matching publications with *valid* decryption shares from the brokers with the same sequence number sn, a subscriber runs Comb to obtain the publication in plaintext and delivers it.

Read. As in Kafka, Chios can serve as a storage system and an authorized client can read stored data (publications) at brokers via engaging a protocol between the client and brokers.

- A client sends brokers a read request for a particular publication of the form (L, c).
- Upon receiving a read request, brokers decide if the client is authorized by checking access control policies associated with the publication. If the client is allowed to have access to the publication, each broker $i \in [1..n]$ sends the client its decryption share $\tau_i \overset{\$}{\leftarrow} \mathsf{ShareDec}_{sk_i}(L, c)$.
- Upon receiving $f+1$ matching replies with *valid* decryption shares from the brokers with the same sequence number sn, the client runs Comb to obtain the publication in plaintext and delivers it.

The above system achieves all properties in Sec. III-A except publication total order.

C. Chios with Publication Total Order

Intuitively, to achieve publication total order, each subscriber needs to maintain a log of valid publications received and deliver them according to the sequence number order assigned by brokers; however, due to access control and subscriber interests, *not all* publications will be sent to all subscribers. Therefore, subscribers do not know if they should wait for or skip publications with certain sequence numbers.

To tackle the issue, we first require servers to additionally maintain topic-based sequence numbers in addition to the global sequence numbers. Doing so, however, does not suffice, as even if two subscribers have the same subscriptions, they may not receive the same publications due to the access control rules. We thus also require that servers send empty messages with sequence numbers to subscribers who are not authorized to receive the corresponding publications. This way, subscribers can safely skip empty publications and go ahead to deliver publications with larger publication sequence numbers.

We now describe in more detail how Chios achieves publication total order. As illustrated in Fig. 2, we maintain two tables: a table for data blocks and a table for publication order indices. The data block table maintains all operations in the system, which are stored in the database. The publication order index table contains metadata of the data blocks and can be derived from the data block table. The index table is stored either in the database or in memory.

For each operation, we store the sequence number (sn), the client id (cid), the operation type (op), the message payload (p), timestamp (ts), access control rules (ac), and the publication topics (tp). Certain fields in the data blocks can be NULL.

sn	cid	ts	op	*p*	ac	tp
0	1000	4	pub	m_0	NULL	price="105"
1	1001	6	write	m_1	101	NULL
2	1000	10	pub	m_2	100,101	price="105", county="Orange"

Data Blocks

tp	S-PS
price="105"	0-0,2-1
county="Orange"	2-0

Publication Order Indices

Figure 2. Data blocks and the publication order indices.

The publication order index table helps achieve topic-based total order (i.e., total order for the publications according to the topics). Specifically, for each topic, we maintain a simple data structure S-PS, where the S field consists of the sequence numbers of operations (sn, the same sequence numbers as in the data blocks table), and the PS field consists of the per-topic sequence numbers (ps).

The PS field contains incremental sequence numbers for a specific topic, ensuring there is no gap in the sequence numbers for operations with the same topic. For instance, as shown in Fig. 2, in the data block table, operations with sequence number 0 and 2 are publications. There are two topics involved in the data block table: price = "105" and county = "Orange". Correspondingly, there are two topics in the publication index table. As both publications have the topic (price = "105"), the topic in the index table has two S-PS numbers: 0-0 and 2-1. The numbers 0 and 2 in the S field are the sequence numbers in the data block table, while the numbers 0 and 1 in the PS field are per-topic sequence numbers. Specifically, brokers distinguish three cases:

- For authorized and interested subscribers, each broker $i \in [1..n]$ sends them $(\mathsf{tp}, ps, \tau_i)$, where tp is the topic, ps is the topic sequence number, and $\tau_i \xleftarrow{\$} \mathsf{ShareDec}_{sk_i}(L, c)$ is the decryption share for broker i.
- For unauthorized and interested subscribers, each broker $i \in [1..n]$ sends them $(\mathsf{tp}, ps, \perp)$, where $\perp$ is a short distinguished symbol denoting an empty message payload (so that subscribers can safely skip the sequence numbers for a particular topic).
- For uninterested subscribers, brokers send nothing.

Each subscriber maintains a log of publications (either empty publications or publications in plaintext) for each topic tp. It delivers publications according to the ps order, and example of which is illustrated in Fig. 3. More specifically,

- Upon receiving $f + 1$ matching publications of the form $(\mathsf{tp}, ps, \tau_i)$ from different brokers, a subscriber runs Comb to obtain a publication in plaintext p and stores p in Δ in its ps's position.
- Upon receiving $f + 1$ matching publications of the form $(\mathsf{tp}, ps, \perp)$ from different brokers, the subscriber directly skips the empty publication in the array Δ in its ps's position.
- The subscriber delivers a publication $p \in \Delta$ with a sequence number ps, if all publications with sequence numbers smaller than ps are either delivered (for non-empty publications) or skipped (empty publications).

V. A MODULAR FRAMEWORK

Chios provides a modular framework allowing trade-offs between functionality, security, and efficiency. Chios currently supports four modules, including an encryption-free module (Module 1), the module using threshold encryption (Module 2, the one we described in Sec. IV), a module using hybrid encryption, and one with broadcast encryption. Due to space limit, we mainly focus on the first two modules. We describe the other two modules in greater detail in the full technical report.

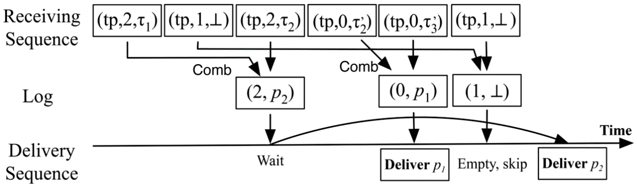


Figure 3. An example of how a subscriber delivers publications assuming $f = 1$ and $n = 4$. The subscriber receives a sequence of messages from BFT brokers and stores them in its buffer. It first receives $f + 1 = 2$ matching messages with $ps = 2$. It then runs Comb to obtain a publication p_2 in plaintext and stores in its log. The subscriber has to wait until publications with smaller sequence numbers (i.e., $ps = 0, 1$) have been dealt with. After the subscriber receives 2 matching messages with $ps = 0$, it runs Comb to obtain p_1 and delivers p_1. It then waits for messages with $ps = 1$. After the subscriber receives two empty messages for $ps = 1$, it directly skips the message and delivers message p_2 stored.

VI. IMPLEMENTATION

Chios consists of a Java library and a Python library with about 30,000 lines of new code. We utilize BFT-SMaRt [11] written in Java as the underlying consensus engine, as BFT-SMaRt is "the most advanced and most widely tested implementation of a BFT consensus protocol" [37]. We use LevelDB [29] as the database. We extend the BFT-SMaRt library and implement a key-value store service. The Java library serves as an ordering service, which assigns a sequence number to a client operation. Then, we wrap the library in Python and develop all the core functionalities.

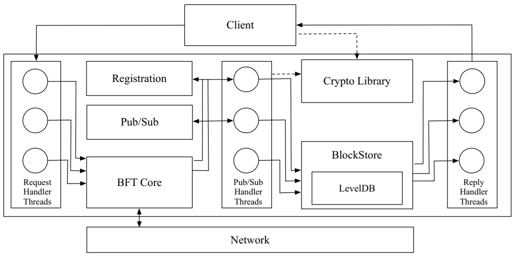


Figure 4. System architecture and message flow.

Fig. 4 illustrates the system architecture and the message flow. The client operations are first handled through a request handler thread pool and the operations are then relayed to the BFT core. The BFT core batches concurrent client operations and assigns a sequence number to each operation. The ordered client operations are then processed by a pub/sub handler thread pool. Each thread processes a client operation at a time and outputs a reply according to the operation type. Chios uses a *batch-process, block-store* approach, where operations are batched according to a tunable parameter BlockSize, ordered, processed, and the results are stored in the database in blocks.

We use ECDSA for authentication and use SHA-256 as our hash function. We implement TDH2-VIL and threshold PRF [16] using the Charm Python library [3]. We use the NIST P-256 curve to provide 128-bit security. We use AES and CBC with ciphertext stealing as our blockcipher and encryption scheme, respectively, to implement the NNL scheme [45].

VII. EVALUATION

Settings. We deployed Chios on Amazon EC2 using up to 31 nodes for brokers and 25 nodes for clients (running up to 1,200 clients in total). Each node, by default, is a compute-optimized *c5.2xlarge* type with 8 virtual CPUs (vCPUs) and 16GB memory. We also test the performance using a general-purpose *t2.medium* type with two vCPUs and 4GB memory to evaluate the performance on different hardware. We evaluate our protocols in both LAN and WAN settings, where the LAN nodes are selected from the same EC2 region, and the WAN nodes are uniformly selected from different regions.

We evaluate the protocols under different network sizes (number of replicas) and contention levels (number of concurrent clients). For each experiment, we use f to represent the network size, where $3f+1$ brokers are launched in total. We use P, C, and B to represent the encryption-free module (Module 1), the threshold encryption module (Module 2), and the broadcast encryption module, respectively. Let $\mathsf{Mod} \in \{\mathsf{P}, \mathsf{C}, \mathsf{B}\}$ and let $\mathsf{op}(\mathsf{Mod})$ represent the operation op in the operation using the Mod module. For instance, $\mathsf{pub}(\mathsf{C})$ denotes pub operations for Module 2.

We examine the average latency under no contention where only one client sends a single operation to the servers. We examine the throughput under high contention of client requests. We evaluate the number of operations processed every second for every 2,000 operations and use the average throughput of the entire experiment.

Overview. For the minimum one failure setting ($f = 1$), the Chios protocol with all desirable features (pub/sub, decentralized confidentiality, and fine-grained access control), achieves throughput of 45 kops/s for pub operations in LAN. To rigorously demonstrate Chios's performance, we first compare Chios Module P with five other pub/sub systems. Next, we evaluate the performance for different Chios Modules.

Comparison with five other pub/sub systems. We first compare Chios Module P with the following five systems, where Chios-Solo, Kafka, and Fabric-Solo are unreplicated systems, while Kafka-Rep and Fabric-Kafka are crash fault-tolerant systems:

- Chios-Solo. Unreplicated, single-node version of Chios.
- Kafka. As we summarized in Table I in Sec. I, Kafka favors performance over reliability and does not achieve any security or reliability goals which we surveyed even in the crash failure model.
- Kafka-Rep. Kafka also supports passive (primary-backup) replication for its brokers with no total order guarantees.
- Fabric-Kafka [6]. Fabric is a popular permissioned blockchain system. Fabric currently does not protect against Byzantine failures. Fabric-Kafka uses the Zookeeper [49] system in Kafka to achieve consensus and is thus only crash fault-tolerant. Hyperpubsub is pub/sub auditing system using Fabric (with Raft [46]) and it is thus slower than Fabric-Kafka.
- Fabric-Solo [6] uses a single node for consensus and is thus not fault-tolerant. One Trinity instance [50] uses Fabric-Solo as its pub/sub auditing system and is slower than Fabric-Solo.

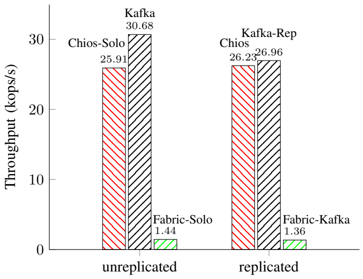


Figure 5. Throughput of Chios, Chios-Solo, Kafka, Kafka-Rep, Fabric-Kafka, and Fabric-Solo.

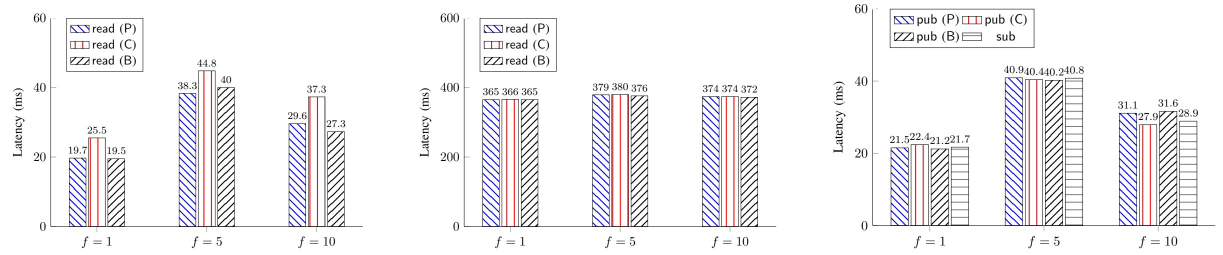
To evaluate the P module of Chios, we randomly assign topic number for publications during evaluation. We implement a read/write smart contract for Fabric and use the write operation for the write throughput. Our evaluation for throughput is standard: publishers send brokers operations, and we increase the number of publishers to obtain the peak throughput. We first find out the number of publishers when each system reaches peak throughput. To ensure a fair comparison, we evaluate the systems under the same *total* workload. Namely, the total number of operations sent publishers is the same for all systems. We let the size of all operations be 1kB and we utilize network sizes that tolerate one failure, i.e., four for Chios and three for Fabric and Kafka.

We report the throughput in LAN using 200 clients of the six systems in Fig. 5. Chios Module P is as efficient as Chios-Solo and is only marginally less efficient than Kafka and Kafka-Rep. Chios is significantly more efficient than Fabric-Kafka and Fabric-Solo and thus even much more efficient than Hyperpubsub and Trinity.

It is unfair to compare Chios with Kafka with more nodes, as Kafka uses independent server instances for horizontal scalability.

Latency of Chios modules. We assess the latency in both the LAN and WAN settings. We let the BlockSize be one to understand the latency caused by the protocol itself. In the LAN setting, the network latency is relatively small, so the overhead is more caused by the BFT agreement and execution of operations (e.g., verifying operation types, database interaction). In the WAN setting, the network latency causes more performance degradation than that in the LAN setting. For the threshold encryption and broadcast encryption modules, the latency evaluated includes the overhead of client-side encryption.

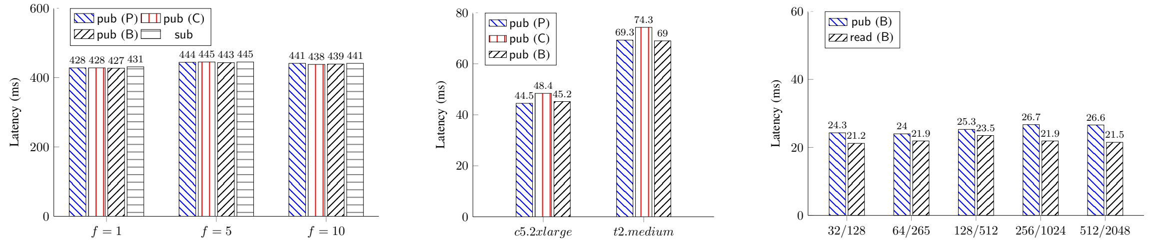
For read *operations.* We assess the latency for read operations in all three encryption modules as the network size increases. Fig. 6(a) reports the latency for the LAN setting. As $\mathsf{read}(\mathsf{P})$ involves no encryption, it has the lowest latency among all



(a) Latency for read operations in the LAN setting with $f = 1, 5$ and 10.

(b) Latency for read operations in the WAN setting with $f = 1, 5$ and 10.

(c) Latency for pub operations in the LAN setting with $f = 1, 5$ and 10.

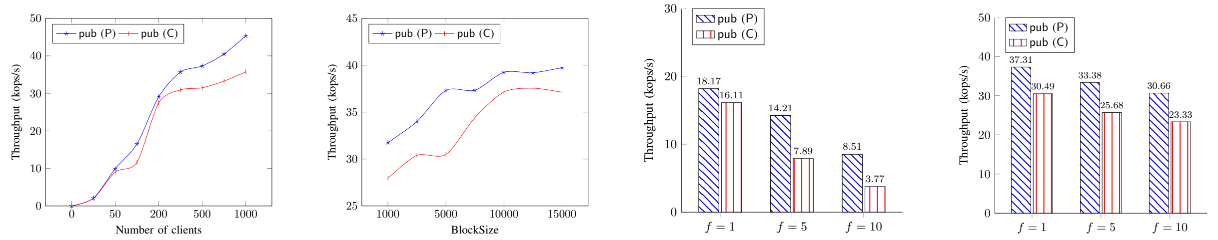


(d) Latency for pub operations in the WAN setting with $f = 1, 5$ and 10.

(e) Latency for pub operations in the LAN setting on different hardware with $f = 1$.

(f) Latency for read(B) and pub(B) operations in LAN with $f = 1$, where a/b represents that a out of b subscribers are revoked.

Figure 6. Latency of different Chios modules.



(a) Throughput in the LAN as the number of client threads increases.

(b) Throughput in the LAN as the size of block increases.

(c) Peak throughput in the WAN using 500 clients.

(d) Throughput in the LAN using 500 clients.

Figure 7. Throughput of different Chios modules.

three modules. For read(C), replicas verify the ac rules, decrypt the ciphertext, and send decryption shares to the clients. Additional overhead is thus incurred. For read(B), we test the latency for the content distribution phase, as the key distribution phase needs to be done only once. The performance of read(B) is consistently better than that of read(C), as it uses symmetric cryptography only. In the WAN setting, the latency difference among the three modules is smaller, as shown in Fig. 6(b), mainly because network latency dominates the overhead.

For pub *and* sub *operations.* We report their latency in Fig. 6(c) and Fig. 6(d). We also report the latency of pub using different hardware in Fig. 6(e). We find that the latency for pub operations is higher than that of read operations. We also find that the latency difference for pub operations between the LAN and WAN settings is much higher than that for read operations. The findings are expected, as Chios implements the BFT read optimization which reduces much communication overhead.

Other operations. In Fig. 6(f), we evaluate the performance of read(B) and write(B) operations as the number of subscribers increases. In all these experiments, we randomly revoke 1/4 of the total subscribers. We find for both operations, the latency is steady, regardless of the number of subscribers. The reason is that the broadcast encryption module uses symmetric cryptography only for the content distribution phase.

Throughput of Chios modules. We evaluate the throughput of Chios with varying BlockSize in the LAN setting when $f = 1$. Fig. 7(a) demonstrates the throughput when the BlockSize is the 5,000 and as the number of concurrent clients increases from 25 to 1,000. The system reaches peak throughput when the number of concurrent clients is larger than 800. The peak throughput that we observe for pub (P) is around 40 kops/s in LAN and 18 kops/s in WAN. We report the throughput when the total number of clients is 375 and as the BlockSize increases in Fig. 7(b). We observe that the throughput becomes larger when the BlockSize increases; however, after BlockSize is larger than 10k, the throughput ceases to increase. In all experiments, the throuhgput for pub (C) is lower than that of pub (P) due to the cryptographic overhead.

We report the throughput for pub (P) and pub (C) using up to 31 servers and 500 concurrent clients for the LAN setting and the WAN setting, in Fig. 7(d) and Fig. 7(c), respectively. For both the LAN and WAN settings, we find that the throughput for both modules degrade when the number of servers increases (**resembling that of BFT-SMaRt, the consensus engine for Chios**), and the throughput for pub (C) degrades more significantly due to the cryptographic overhead.

VIII. CONCLUSION

We design and implement Chios, a highly efficient and intrusion-tolerant pub/sub system. Chios addresses two major challenges in pub/sub in terms of confidentiality and reliability: Chios achieves decentralized confidentiality with fine-grained and attribute-based access control and publication total order with two-way information control. Chios provides modular instances designed to meet different goals. Through extensive evaluation, we demonstrate Chios is efficient.

REFERENCES

[1] B. Adida, O. de Marneffe, O. Pereira, and J.-J. Quisquater. Electing a University President using open-audit voting: Analysis of real-world use of Helios. *EVT/WOTE 2009.*

[2] A. Adya et al. FARSITE: Federated available and reliable storage for incompletely trusted environments. *OSDI '02.*

[3] J. A. Akinyele et al. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013.

[4] Amazon Simple Notification Service (SNS). https://aws.amazon.com/sns/

[5] E. Androulaki, C. Cachin, A. D. Caro, and E. Kokoris-Kogias. Channels: Horizontal scaling and confidentiality on permissioned blockchains. *ESORICS 2018.*

[6] E. Androulaki et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. *EuroSys 2018.*

[7] Apache Kafka. https://kafka.apache.org/

[8] J. Bacon, D. M. Eyers, J. Singh, and P. R. Pietzuch. Access control in publish/subscribe systems. *DEBS 2008.*

[9] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. *CRYPTO 2013.*

[10] A. Bessani, E. Alchieri, M. Correia, and J. Fraga. DepSpace: A Byzantine fault-tolerant coordination service. *EuroSys 2008.*

[11] A. Bessani, J. Sousa, and E. Alchieri. State Machine Replication for the Masses with BFT-SMART. *DSN 2014.*

[12] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. *IEEE S&P 2007.*

[13] D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. *CRYPTO 2005.*

[14] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for Hyperledger Fabric. *SRDS 2019.*

[15] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). *CRYPTO 2001.*

[16] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18(3), 219–246.

[17] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. *SRDS 2005.*

[18] R. Cheng et al. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *EuroS&P, 2019*

[19] CMS advances interoperability & patient access to health data through new proposals. https://edit.cms.gov/newsroom/fact-sheets/cms-advances-interoperability-patient-access-health-data-through-new-proposals

[20] T. Chang, S. Duan, H. Meling, S. Peisert, and H. Zhang. P2S: A fault-tolerant publish/subscribe infrastructure. *DEBS 2014.*

[21] R. Cramer, I. Damgard, and J. Nielsen. Secure multiparty computation and secret sharing. Cambridge University Press.

[22] S. Duan, M. K. Reiter, and H. Zhang. Secure causal atomic broadcast, revisited. *DSN 2017.*

[23] S. Duan and H. Zhang. Practical state machine replication with confidentiality. *SRDS*, 2016.

[24] Benjamin Eze et al. Policy-based data integration for e-health monitoring processes in a B2B environment: experiences from Canada. *JTAER* 5, 1 (April 2010), 56–70.

[25] FAYE, Simple pub/sub messaging for the web. https://faye.jcoglan.com

[26] A. Fiat and M. Naor. Broadcast encryption. *CRYPRO 1993.*

[27] L. Fiege et al. Security aspects in publish/subscribe systems. *DEBS'04.*

[28] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *TCS*, 243 (1-2): 363–389, 2000.

[29] S. Ghemawat and J. Dean. LevelDB, A fast and lightweight key/value database library by Google. 2014.

[30] Google Cloud Pub/Sub. https://cloud.google.com/pubsub/

[31] M. Ion, G. Russello, and B. Crispo. An implementation of event and filter confidentiality in pub/sub systems and its application to e-health. *CCS 2010.*

[32] M. Ion, G. Russello, and B. Crispo. Supporting publication and subscription confidentiality in pub/sub networks. *SecureComm 2010.*

[33] M. Ion, G. Russello, and B. Crispo. Design and implementation of a confidentiality and access control solution for publish/subscribe systems. *Computer Networks 56*, 7 (2012), 2014–2037.

[34] IOTA. https://www.iota.org/

[35] A. Iyengar, R. Cahn, C. Jutla, and J. Garay. Design and implementation of a secure distributed data repository. *IFIP ISC*, 1998.

[36] L. Jehl and H. Meling. Towards Byzantine fault tolerant publish/subscribe: a state machine approach. *HotDep 2013.*

[37] A. Kate, Y. Huang, and I. Goldberg. Distributed key generation in the wild. IACR Cryptology ePrint Archive 2012: 377 (2012).

[38] R. Kazemzadeh and H. Jacobsen. Reliable and highly available distributed publish/subscribe service. *SRDS 2009.*

[39] R. Kazemzadeh and H. Jacobsen. Publiyprime: Exploiting overlay neighborhoods to defeat byzantine publish/subscribe brokers. *University of Toronto, 2013.*

[40] H. Khurana. Scalable security and accounting services for content-based publish/subscribe systems. *SAC 2005.*

[41] A. E. Kosba et al. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. *IEEE Security & Privacy 2016.*

[42] J. Kubiatowicz et al. OceanStore: An architecture for global-scale persistent storage. *ASPLOS*, 2000.

[43] A. B. Lewko and B. Waters. Decentralizing Attribute-Based Encryption. *EUROCRYPT 2011.*

[44] MQTT. www.mqtt.org/

[45] D. Naor, and M. Naor, and J. B. Lotspiech. Revocation and tracing schemes for stateless receivers. *CRYPTO 2001.*

[46] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. *ATC*, 2014.

[47] E. Onica, P. Felber, H. Mercier, and E. Rivière. Confidentiality-preserving publish/subscribe: A survey. *ACM Comput. Surv.*, 2016.

[48] R. Padilha and F. Pedone. Belisarius: BFT Storage with confidentiality. *NCA 2011.*

[49] P. Hunt, M. Konar, F. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *ATC*, 2010.

[50] G. S. Ramachandran et al. Trinity: A Byzantine fault-tolerant distributed publish-subscribe system with immutable blockchain-based persistence. *ICBC.*

[51] M. K. Reiter and K. Birman. How to securely replicate services. *ACM TOPLAS*, vol. 16 issue 3, pp. 986–1009, ACM, 1994.

[52] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *EUROCRYPT '98.*

[53] M. Srivatsa and L. Liu. Securing publish-subscribe overlay services with EventGuard. *CCS 2005.*

[54] M. A. Tariq, B. Koldehofe, and K. Rothermel. Securing Broker-Less Publish/Subscribe Systems Using Identity-Based Encryption. *IEEE Trans. on Parallel and Distributed Systems* 25, 2 (2014), 518–528.

[55] Tendermint core. https://github.com/tendermint/tendermint

[56] S. Vinoski. 2006. Advanced message queuing protocol. *IEEE Internet Computing* 10, 6 (2006).

[57] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Byzantium Version.

[58] A. Wun and H.-A. Jacobsen. A Policy Management Framework for Content-Based Publish/Subscribe Middleware. *Middleware 2007.*

[59] J. Yin et al. Separating agreement from execution for Byzantine fault tolerant services. *SOSP 2003.*

[60] K. Zhang, V. Muthusamy, H-A. Jacobsen. Total order in content-based publish/subscribe systems. *ICDCS 2012.*

[61] Y. Zhao and D.C. Sturman. Dynamic Access Control in a Content-based Publish/Subscribe System with Delivery Guarantees. *ICDCS 2006.*

[62] N. Zupan, K. Zhang, and H-A. Jacobsen. Hyperpubsub: a decentralized, permissioned, publish/subscribe service using blockchains: demo. *Middleware Posters and Demos 2017.*