Secure Publish-Process-Subscribe System for Dispersed Computing

Weizhao Jin
USC & Information Sciences Institute
weizhaoj@usc.edu

Bhaskar Krishnamachari
University of Southern California
bkrishna@usc.edu

Muhammad Naveed
University of Southern California
mnaveed@usc.edu

Srivatsan Ravi
USC & Information Sciences Institute
sravi@isi.edu

Eduard Sanou
University of Southern California
eduardsanou@gmail.com

Kwame-Lante Wright
Carnegie Mellon University
kwame.wright@gmail.com

Abstract

Publish-subscribe protocols enable real-time multi-point-to-multi-point communications for many dispersed computing systems like Internet of Things (IoT) applications. Recent interest has focused on adding processing to such publish-subscribe protocols to enable computation over real-time streams such that the protocols can provide functionalities such as sensor fusion, compression, and other statistical analysis on raw sensor data. However, unlike pure publish-subscribe protocols, which can be easily deployed with end-to-end transport layer encryption, it is challenging to ensure security in such publish-process-subscribe protocols when the processing is carried out on an untrusted third party. In this work, we present $\mathcal{XYZ}$, a secure publish-process-subscribe system that can preserve the confidentiality of computations and support multi-publisher-multi-subscriber settings. Within $\mathcal{XYZ}$, we design two distinct schemes: the first using Yao's garbled circuits (the GC-Based Scheme) and the second using homomorphic encryption with proxy re-encryption (the Proxy-HE Scheme). We build implementations of the two schemes as an integrated system atop the Message Queue Telemetry Transport (MQTT) pub-sub protocol. We evaluate our system on several functions and also demonstrate real-world applications based on it. The evaluation shows that the GC-Based Scheme can finish most tasks two orders of magnitude times faster than the Proxy-HE Scheme while Proxy-HE can still securely complete tasks within an acceptable time for most functions but with a different security assumption and a simpler system structure.

1 Introduction

Modern interconnected networked systems like Internet of Things are dispersed often requiring a strategic, opportunistic movement of computation to data, and data to computation, in a fashion that best suits user application needs. Recent developments in IoT enable applications to use multi-point-to-multi-point communication by use of the publish-subscribe (pub-sub) paradigm [21]. The publish-subscribe messaging allows multiple data consumers to connect to streams of real-time data from multiple sensors. Commonly used examples of pub-sub protocols are Message Queue Telemetry Transport (MQTT) [31], Advanced Message Queuing Protocol (AMQP) [30] and commercial pub-sub platform as a service (PaaS) providers such as PubNub [42] with their own proprietary protocols and APIs. The key idea behind pub-sub protocols is the use of a broker as a relay, which is typically centralized and implemented on a cloud server, such as Mosquitto of MQTT [35]. Sensors (also called publishers) publish messages to specified "topics" that are sent to the broker; data consumers (also called subscribers) send to the broker a subscribe request to specified topics and receive data from the broker. The broker in a traditional pub-sub system plays primarily a message-forwarding role with optional extension to client authentication, but this basic functionality does not serve the emerging need for data processing in such systems [33]. Enabling data processing on the broker before forwarding instead of merely relaying raw data helps provide more meaningful data derived from raw sensor data and detect potential anomalies. In some cases, it can also reduce the overall throughput and improve clients' energy efficiency. This is important in IoT environments considering the fact that most IoT devices are of a low-budget setting. One of the examples of such intermediate processing is that PubNub recently introduced processing capability into their PaaS system in the form of real-time compute functions as "BLOCKS" [43].

However, as brokers are typically hosted on third-party servers, adding computational processing to pub-sub middleman introduces concerns about security. An application that wishes to make use of a third-party broker for traditional pub-sub messaging could always use end-to-end encryption to provide security [10, 41], but with computational functionality being moved to the server, such encryption is no longer enough. Additionally, approaches like moving computation to clients simply do not work when sensitive individual data has to be aggregated but protected, for example, building privacy-preserving machine learning models from sensitive data of medical sensors [1] and federally aggregating model parameters from multiple IoT users while preventing information leakage especially for users with small datasets [44]. To our knowledge, with the exception of proposals to utilize trusted execution environments, such as Intel SGX [5, 29], there is no prior practically implemented protocol that provides secure computation on a pub-sub broker for IoT. Given the security vulnerabilities identified with SGX in recent years [16, 28, 45], a system based on secure computation would be more desirable. The development of such a system has the potential to dramatically lower the barrier for use of third-party edge/cloud-based computation, especially for privacy-sensitive data streams such as data from smart homes or wearable devices collecting physiological information [3]. This is the focus of our work. To reduce the impact from performing secure computation on the overhead of IoT devices as much as possible, clients in our system are only required to encrypt or decrypt the data, which is the relatively cheap part in secure computation and has already existed in previous secure pub-sub messaging protocols.

Our Contributions. Our core contributions can be summarized as follows:

1. It is challenging to integrate multi-party computation into IoT messaging protocols constricted by its pub-sub structure despite the growing need of intelligent movement between secure computation and data in such a dispersed system. We build a system, $\mathcal{XYZ}$, to bridge this gap.
2. $\mathcal{XYZ}$ is a secure publish-process-subscribe system based on MQTT for IoT applications that can perform secure multi-party computation on the broker side. We propose and implement two distinct multi-party computation schemes with different system constructions and security assumptions. Both schemes support multi-publisher-multi-subscriber settings.
3. The first scheme is based on Yao's garbled circuits [49]. We introduce techniques like communication reduction and seed synchronization to circumvent the constraints of traditional pub-sub. We also provide forward-secure seed extension for extra security.
4. The second scheme is based on homomorphic encryption [6] and proxy re-encryption [37]. We introduce techniques like key exchange reduction to mitigate conflicts between proxy re-encryption and traditional pub-sub structure, and subscriber representative mechanism to support multi subscribers.
5. We evaluated $\mathcal{XYZ}$ on different functions, i.e., mean, variance, weighted mean, private set intersection and secure federated learning. The function library in our system can be easily extended to support more complex functions. We also provide concrete real-world IoT applications based on our system.

The rest of the paper is structured as follows: in §2 we provide an overview of our secure publish-process-subscribe system and its related work; in §3 and §4 we introduce two different schemes (the GC-Based Scheme and the Proxy-HE Scheme) in our system respectively; in §5, we describe the detailed implementation of our system; lastly in §6, we evaluate our system and demonstrate concrete real-world applications.

2 Overview and Related Work

2.1 Overview

Our secure publish-process-subscribe protocol should handle secure computation on the broker's side using encrypted data from publishers and distribute encrypted processed data to subscribers. Our protocol involves a publishers, b publishers and third-party server(s). We assume a semi-honest adversary $\mathcal{A}$ who can corrupt a certain set of clients and the server(s). A semi-honest adversary does not deviate from the protocol but tries to learn as much information as possible. Additionally, certain collusion is restricted. Our security definition requires that $\mathcal{A}$ only learns the data from corrupted publishers and final outputs from corrupted subscribers, but nothing about honest parties' inputs.

Definition 1 (UC-Security)**.** *A protocol π securely realizes $\mathcal{F}$ in the presence of $\mathcal{A}$ in the real world, if there exists a simulator $\mathcal{S}$ in the ideal world such that for all inputs, probability distributions of the ideal world and the real world are indistinguishable.*

Ideal Functionality. We describe the ideal functionality as a generic definition of a secure publish-process-subscribe protocol. Our ideal functionality $\mathcal{F}$ interacts with participating parties as shown in Figure 1.

Real-World Schemes. Our system adopts two different multi-party computation schemes using garbled circuits and homomorphic encryption with proxy re-encryption respectively with different security assumptions and system constructions. This availability of different designs offers users more choices to better fit specific needs when constructing secure publish-subscribe-process systems in practice. The summary of scheme comparison is listed in Table 1. The two schemes have different security assumptions. The detailed definitions of $\mathcal{A}$ in our two different schemes will be described in §3 and §4. In terms of the system complexity, our GC-Based Scheme needs both the garbler and the broker on third-party servers. Even with our reduced communication

Initialization

- Each new publisher sends a policy to the broker specifying allowed computation on its data.

Publish

- Each publisher publishes its data to $\mathcal{F}$. If the data from a publisher is not received in the given time period, $\mathcal{F}$ marks it as null.

Subscribe

- To subscribe to the computation C, each subscriber sends a subscription message to the broker containing requested C.
- The broker sends C and its subscribers to $\mathcal{F}$.

Process

- $\mathcal{F}$ determines a subset $P' \subset P$ of publishers whose data can be used to compute C, then sends P' to the broker.
- The broker sends back $P_C \subset P'$ whose policies allow C.
- If data of all available publishers in P_C is enough to compute C, $\mathcal{F}$ evaluates it and sends the result to subscribers, otherwise $\mathcal{F}$ sends an empty message to subscribers.

Figure 1. Ideal World Functionality

extension, which circumvents the direct communication among the garbler and clients in pub-sub setup, this scheme still requires the garbler as an independent intermediate party. Additionally, extra procedures like key ratcheting and seed synchronization are needed for indirect communication between clients and the garbler as well as seed sharing for multi-party support. On the contrary, the broker can act as a proxy in the Proxy-HE Scheme, which can be easily constructed on top of the standard pub-sub protocol. As a trade-off, the Proxy-HE Scheme is two orders of magnitude times slower than the GC-based Scheme in general.

Scheme	GC-Based	Proxy-HE
Adversary	can't control both Garbler and Broker	can't control both Broker and Subscribers
System Complexity	needs both Garbler/Broker	only needs Broker
Issues	seed sharing; data reusability	computation overhead
Typical Cost	~10 ms	~1000 ms

*Typical cost is from computing variance with 100 publishers.

Table 1. Comparison of Two Schemes in Our System

2.2 Related Work

Secure Pub/Sub System. A few previous studies on secure pub/sub messaging systems have been conducted and secure pub/sub schemes have been proposed [38, 47], but these work all focused on the simple pub/sub system without a computation functionality from the broker.

Cryptographic Primitives & Systems. Our secure publish-process-subscribe system is related to the work on garbled circuits [14, 15, 27, 32] and homomorphic encryption [12, 26]. However, existing schemes do not fit our real-time publish-process-subscribe system. Kamara et al. developed two protocols, a covertly (in the covert adversary model, an adversary is caught probabilistically) secure protocol that outsources the garbled circuit generation and a maliciously secure protocol that outsources evaluation [32]. Carter et al. also proposed a maliciously secure protocol that outsources garbled circuit evaluation but uses a new oblivious transfer mechanism to reduce bandwidth and computation [15]. In another paper, Carter et al. proposed a maliciously secure protocol that outsources garbled circuit generation [14]. Bachrach et al. developed a protocol that allows a set of parties with data stored in the cloud to compute on encrypted data using a third-party evaluator [27]. All the work above attempted to provide a solution to more general cloud server models using garbled circuits but didn't address the issues from the publish-subscribe protocol's unique messaging mechanism. Different from how garbled circuits work, homomorphic encryption [6] allows arbitrary computation on encrypted data. Gentry proposed the first fully homomorphic encryption scheme [12, 26] followed by several improved schemes, e.g., the BGV scheme [11]. Dijk et al. showed that privacy-preserving outsourced computation on data from multiple parties and supplying output to multiple parties requires, in addition to homomorphic encryption, access-controlled ciphertexts and re-encryption [46]. They reduce a scheme that computes data from two parties and supplies outputs to two parties to black-box program obfuscation, which is hard to accomplish in general [7]. Additionally, restrictions on parties in the paper make its potential application less realistic. Nikolaenko et al. proposed a scalable privacy-preserving system for ridge-regression combining additive homomorphic encryption and Yao's garbled circuits [40]. In their setting, a single evaluator is interested in learning ridge regression over data of a large number of data owners without learning the individual data of data owners. Our system works in a different way: (*a.*) we don't want to reveal output to the evaluator, (*b.*) we have multiple subscribers who want the output of computation from multiple publishers, and (*c.*) our data owners, publishers, are oblivious of subscribers and subscribers are oblivious of publishers. Nikolaenko, et al.,[39] proposed a similar system but for privacy-preserving matrix factoring instead of ridge regression.

IoT with Proxy Re-Encryption. We also make use of proxy re-encryption in our Proxy-HE Scheme. This scheme, first proposed as a method to delegate decryption rights [37], solves the asynchronous encryption issue in the publish-subscribe protocol. Polyakov et al. proposed a proxy re-encryption scheme based on homomorphic encryption to tackle this problem [10, 41]. However, their work focused on the simple publish-subscribe setup and did not further address the issue that publishers and subscribers need to communicate back and forth via the broker to generate re-encryption keys every time a communication is established. In our work, we use the proxy re-encryption library built by them but apply it into the Proxy-HE Scheme in our secure publish-process-subscribe system with additional communication optimization.

3 The GC-Based Scheme

In this section, we describe our scheme designed with Yao's garbled circuits. The overall structure of the scheme is shown in Figure 2. In the real world, $\mathcal{F}$ is replaced by our scheme described in §3.2.

3.1 Components

Yao's Garbled Circuits. Yao's garbled circuits [49] is a secure computation scheme that allows the participating parties to evaluate their private inputs in a function even if they do not trust each other. The components of Yao's garbled circuits GC, with algorithms $(G, Encode, Eval, D)$, can be defined as follows [48]:

1. On input circuit c, the garbling algorithm G outputs a garbled circuit C, encoding e and decoding d.
2. On inputs (e, x), the encoding algorithm $Encode$ outputs a garbled output X, where x is the original input. Then the evaluation algorithm $Eval$ takes in (C, X) and outputs a garbled result Y.
3. On inputs (d, Y), the decoding algorithm D outputs the plaintext.

Reduced Communication Extension. The basic protocol assumes direct communication with the garbler. However, the publishers and subscribers in our system communicate with the garbler only through the broker. To address this issue, we describe an extension that allows clients and the garbler to generate wire labels/masks independently. This ensures our scheme's compatibility with a standard publish-subscribe system where all communication is only through the broker.

Publishers and the garbler share a truly random seed s and use a pseudorandom number generator to independently generate two wire labels for each input bit, circumventing wire label exchange between publishers and the garbler. Similarly, subscribers for the computation C and the garbler share a truly random seed s' and use a pseudorandom number generator to independently generate output masks, avoiding direct output mask exchange between subscribers and the garbler.

Seed Synchronization. The above method requires synchronization between clients and the garbler. We adapt the key ratcheting protocol of Signal, a popular secure messaging protocol, to generate seeds securely. Ratchet keys work by advancing a secret key at every round using the preimage-resistance property of a cryptographic hash function [19] [25]. At any round, a seed can be derived from a ratchet key to be used to generate pseudorandom strings. To maintain synchronization of the ratchet keys between the clients and the garbler, when sending values, publishers add the round of the ratchet key to derive the seeds used to generate the labels in the message. When the broker requests the garbling of the circuit to the garbler, it also specifies the rounds of the values it will use, such that the garbler can advance the ratchet key accordingly to derive the same seed and generate matching labels. Similarly, the garbler tells the broker the function ratchet key round for generating the mask, such that the broker can forward this information to subscribers which in turn advance their stored ratchet keys to derive a matching mask.

Forward-Secure Seeds. While the extension reduces publishers' and subscribers' communication with the garbler significantly, an adversary stealing a seed s from a publisher and colluding with the broker compromises the confidentiality of all of the publisher's inputs, including past, current, and future inputs. Similarly, an adversary stealing the seed s' for the computation C from a subscriber and colluding with the broker compromises the confidentiality of outputs of all executions.

We design an extra procedure that ensures that seeds are forward-secure, i.e., an adversary stealing a seed wouldn't be able to compromise the confidentiality of any past inputs and outputs. The key ratcheting used in our scheme can make all seeds s and s' forward-secure. An adversary stealing publishers' seed s or subscribers' seed s' would still learn all current and future inputs of the publisher or outputs for computation C. But once the adversary compromises target clients, it will learn this information anyway with or without stealing the seeds. The detailed protocol can be found in Figure 3.

3.2 Protocol Design

As shown in Figure 2, the design of our GC-Based Scheme includes four major parties: publisher(s), the broker, the garbler, subscriber(s). We first describe the threat model for this scheme and then explain the detailed design in two different settings: single-publisher-single-subscriber and multi-publisher-multi-subscriber.

Adversary Model. In the GC-Based Scheme, we have four parties in $GC(a, b)$: a publishers, the broker, the garbler and b subscribers as defined.

Definition 2 (The GC-Based Adversary). *A semi-honest adversary $\mathcal{A}_{\mathcal{GC}}$ can corrupt any subset of b subscribers and at most $a - 2$ publishers. $\mathcal{A}_{\mathcal{GC}}$ can corrupt the broker or the garbler, but not at the same time. In other words, the broker and the garbler can not collude with each other.*

Single-Publisher-Single-Subscriber. In order to publish a value, the publisher generates two wire labels w_0 and w_1 for every bit b of the value, sends both labels w_0 and w_1 to the garbler, and only w_b to the broker; the broker receives the computation request from the subscriber and requests the garbler to garble the circuit for $XOR \circ C$; the garbler sends the masked result back

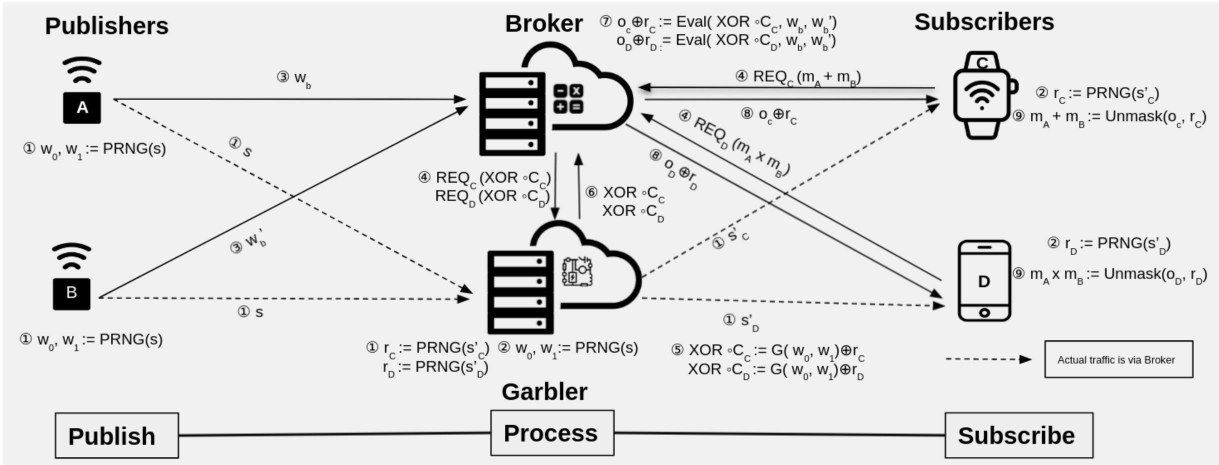

Figure 2. Structure of the GC-Based Scheme: two cloud servers but the actual communication between clients and the garbler is via the broker

to the broker for it to evaluate; the subscriber unmasks the result from the broker.

Multi-Publisher-Multi-Subscriber. The multi-sub functionality can be realized by the garbler sharing the same mask seed with subscribers under the same computation request. However, the functionality of multiple publishers requires sharing of seeds for wire labels. This is achievable but needs extra sharing constructions(for example, sharing same seeds among publishers for the computation involving them but this has to be done for each computation separately) and additional trust among publishers (an adversary controls both the broker and other publishers sharing seeds with honest publishers can reveal honest publishers' data).

Figure 3 depicts how our GC-Based Scheme works.

3.3 Security Analysis

We describe a simulator $\mathcal{S}$ that simulates the view of the adversary $\mathcal{A}_{GC}$ in the ideal world to prove that our scheme guarantees both correctness and security.

$\mathcal{S}$ receives from $\mathcal{F}$ the number of publishers $|P_C|$ whose policy allows computing C on their data. $\mathcal{S}$ creates $2l|P_C|$ number of random wire labels $(r_0^0, r_0^1), \ldots, (r_{2l|P_C|-1}^0, r_{2l|P_C|-1}^1)$, where l being the bit-length of a publisher's input. We use a blackbox garbled circuit simulator from the projective prv.sim secure garbling scheme with circuit $M \circ C$ being the side information as described in [9].

$\mathcal{S}$ receives $\mathcal{F}(M \circ C, \vec{x}_C)$ from $\mathcal{F}$, where M is an XOR masking function. $\mathcal{S}$ sends $\mathcal{F}(M \circ C, \vec{x}_C)$ to the garbled circuit simulator and obtains a fake garbled GC_{fake}. $\mathcal{S}$ generates a random string o_r of the same length as output. $\mathcal{S}$ sends $(GC_{fake}, r_0^0, \ldots, r_{2l|P_C|-1}^0, o_r)$ to the adversary. As garbled circuits distribution is independent of the input wire labels, GC_{fake} is computationally indistinguishable from the GC in the real execution. The random output o_r in ideal execution is indistinguishable from $o + r$ in the real execution.

In the ideal world, $\mathcal{S}$ creates a fake garbled circuit. Note that this fake garbled circuit doesn't use wire labels $(r_0^0, r_0^1), \ldots, (r_{2l|P_C|-1}^0, r_{2l|P_C|-1}^1)$ for garbling. Otherwise, the adversary could use $r_0^0, \ldots, r_{2l|P_C|-1}^0$ labels to evaluate the circuit on $0^{l|P_C|}$, which would allow the adversary to distinguish between real and ideal executions.

Our security assumption allows at most $a - 2$ corrupted publishers. $\mathcal{A}_{\mathcal{GC}}$ learns the final output with $a - 2$ corrupted publishers' inputs while $\mathcal{S}$ has the same information. Neither of them could learn the input of the two honest publishers' inputs.

The view of $\mathcal{S}$ in the ideal world is indistinguishable from the view that $\mathcal{A}_{\mathcal{GC}}$ has in the real world execution.

4 The Proxy-HE Scheme

In the previous section, we construct a secure publish-process-subscriber scheme using Yao's garbled circuits, which requires both the broker and the garbler. Additionally, our GC-Based Scheme restricts the adversary to compromising only one third-party server (the broker or the garbler) at one time while also requiring communication between the garbler and clients goes through the broker. Besides the limitations listed above, standard garbled circuits also suffer from reusability, namely for each type of computation involving different publishers, the wire labels have to be different. This introduces the issues that a publisher needs to publish different versions of the same data for each computation and that a publishers' current inputs can not be extended to computation involving a distinct set of publishers in the future. In a real system, these issues can potentially increase the overhead by a large margin.

In this section, we design a Proxy Homomorphic Encryption (Proxy-HE) Scheme with a simpler structure and a different security assumption while addressing the issues discussed above. Besides homomorphic encryption, our scheme also uses proxy re-encryption to solve

Initialization

- Each new publisher sends the broker a policy specifying allowed computations on its data.
- Each new publisher generates and sends to the garbler a truly random seed s. This seed will be used to create wire labels without interaction.

Subscribe

- To subscribe computation C, each subscriber sends a subscription request containing C to the broker. If the broker does not allow the subscriber to learn C's output, it sends an error message back to the subscriber.
- Each new subscriber shares a truly random seed s' with the garbler for masking/unmasking the result.

Publish

- To publish kth value, the publisher generates two pseudorandom wire labels, w_0 and w_1, using a seed s from a pseudorandom number generator (PRNG), for each bit of the value. w_0 is ith and w_1 is $(i+1)$th numbers in pseudorandom sequence generated using s; $2kL \le i < 2(k+1)L$, L being the bit-length of a value.
- For each input bit b, the publisher sends only wire label w_b to the broker.

Process

- After receiving w_b, the broker sends the garbler identifiers of publishers along with the set of subscribers allowed to the computation, then requests the garbler to garble circuit for $XOR \circ C$. XOR is used to mask the output of the circuit.
- The garbler independently generates input wire labels using the seed s from each publisher contributing input and an output mask r using s' for the output.
- The garbler generates a garbled circuit GC for the circuit $XOR \circ C$ using both wire labels for each input bit, w_0 and w_1 for a bit b. The garbler uses the mask r it to mask the output o of C, such that evaluating GC would result in a masked output $o \oplus r$.
- The broker evaluates the garbled circuit using wire labels sent by publishers in set P_C, obtains masked output $o \oplus r$, and sends $o \oplus r$ to all subscribers of computation C.
- Subscribers in the set S_C use r to unmask the output o.

Forward-Secure Seeds

- Generate a truly random key K_0.
- Generate, using pseudorandom function (PRF) with key K_0, a pseudorandom seed s_0 and a pseudorandom key for the ratchet round 1. Seed s_0 is used to generate pseudorandom strings during ratchet round 0.
- At round i, using PRF with key K_i, generate a pseudorandom seed s_i and key for ratchet round $i+1$. Seed s_i is used to generate pseudorandom strings during ratchet round i.

Figure 3. The GC-Based Scheme

encryption issues in secure publish-subscribe systems. The overall structure is shown in Figure 4. In the real world, $\mathcal{F}$ is replaced by our scheme described in §4.2.

4.1 Components

Homomorphic Encryption. Homomorphic encryption (HE) [6] is a scheme that allows computation to be performed on encrypted data without revealing the original data to the computing parties. The components of HE in general, with algorithms $(G, Enc, Eval, D)$, can be defined as follows:

1. The key generation algorithm G outputs a key pair (Pk, Sk). The encryption algorithm Enc takes in messages $m_1, \cdots, m_n$ and Pk, then outputs $C_1, \cdots, C_n$.
2. On inputs $(C_1, \cdots, C_n)$ and the computation f, the evaluation algorithm $Eval$ outputs the result C_{result}.
3. The decryption algorithm takes inputs (Sk, C_{result}) and outputs the plaintext result $f(m_1, \cdots, m_n)$.

Proxy Re-Encryption. Proxy re-encryption (PRE) was first proposed to delegate decryption rights [37] and can be applied in many cryptographic scenarios nowadays. PRE enables ciphertexts to be decrypted by a secret key that is not paired with the original public key encrypting the plaintexts. PRE, with algorithms (KG, E, RG, RE, D), can be defined as follows [4]:

1. The standard key generation algorithm KG outputs a key pair (Pk_A, Sk_A) for Party A and another key pair (Pk_B, Sk_B) for Party B. Party A uses Pk_A to encrypt the message m with the encryption algorithm E, which outputs ciphertext C_A.
2. The re-encryption key generation algorithm RG takes the inputs (Pk_A, Sk_A, Pk_B, Sk_B) and outputs a key $Rk_{A\to B}$ for re-encryption. On inputs $(Rk_{A\to B}, C_A)$, the proxy then applies the re-encryption algorithm RE and outputs $C_{A\to B}$.
3. Party B applies the decryption algorithm D on inputs $(C_{A\to B}, Sk_B)$ and get the output m.

In our scheme shown in Figure 5, publishers can encrypt the data using their own public key; after proxy re-encryption, subscribers are able to decrypt the ciphertext with subscribers' secret key. This solves the asynchronization issue under the traditional publish-subscribe encryption, allowing publishers to publish messages without the need to wait for subscribers' public keys to encrypt their data. In our implementation, we use the PRE built-in PALISADE library [41].

Key Exchange Reduction. As shown in Figure 5, proxy re-encryption would require that, once a subscriber requests computation, publishers (also the key authority of themselves) involved in the computation have to receive the public key from the subscriber and then regenerate a re-encryption key for re-encrypting the original encrypted data [41]. This introduces the asynchronous communication issue again under the IoT context and also an additional communication cost. To solve this problem, we design a solution called key exchange reduction. At the initialization state of the system, the broker asks all subscribers to upload their public keys. Then each publisher regenerates a re-encryption key for each subscriber once they receive subscribers' public keys from the broker. The broker maintains a map between a re-encryption key and its subscriber-publisher pair. Every time when a new client joins the system,

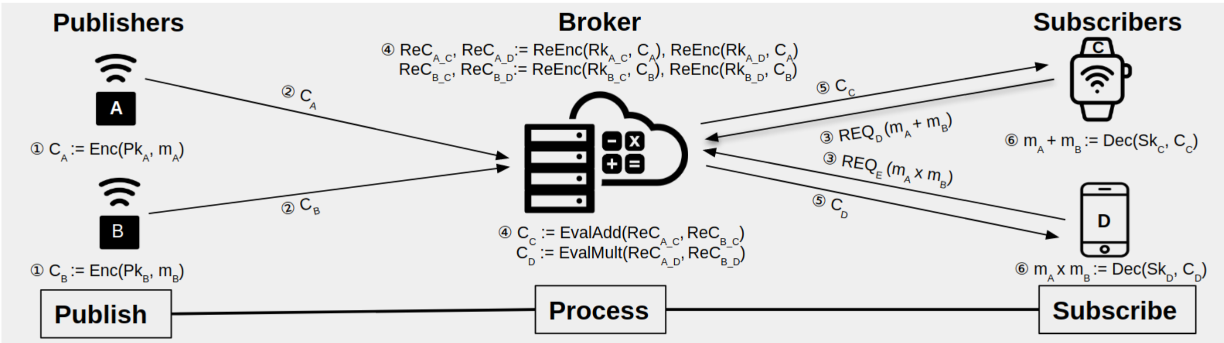

Figure 4. Structure of the Proxy-HE Scheme: only one cloud server which handles both communication and computation

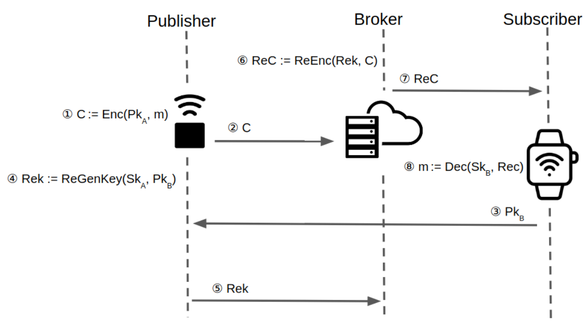

Figure 5. Single-Hop PRE Scheme without Key Exchange Reduction

the broker updates the map. Under this design, whenever a subscriber requests computation, the broker only needs to find the re-encryption keys from the map to re-encrypt the data without going back to the publishers, which reduces the key exchange communication.

4.2 Scheme Design

Figure 4 depicts the structure of the Proxy-HE Scheme, which only has three major parties without the need for a garbler. Similarly, we describe the threat model and the two different settings of this scheme.

Adversary Model. In the Proxy-HE Scheme $PHE(a, b)$, we have three parties: a publishers, the broker and b subscribers as defined.

Definition 3 (The Proxy-HE Adversary)**.** *A semi-honest adversary $\mathcal{A}_{\mathcal{PHE}}$ can corrupt both the broker and at most $a-1$ publishers, or, $\mathcal{A}_{\mathcal{PHE}}$ can corrupt any subset of the b subscribers and at most $a-2$ publishers. $\mathcal{A}_{\mathcal{PHE}}$ can not corrupt the broker and subscribers at the same time.*

We believe the assumption that the broker can not collude with subscribers is acceptable because it is inevitable that the adversary can obtain a publisher's raw data when it controls both the server and subscribers.

Single-Publisher-Single-Subscriber. In this setting, the publisher publishes encrypted data using its public key; the broker re-encrypts the data using the re-encryption key of the publisher and the subscriber, and performs homomorphic computation; the subscriber requests desired computation then decrypts results sent from the broker using its private key. Proxy re-encryption is used here to solve the asynchronization issue between the publisher's publishing encrypted data and the subscriber's requesting secure computation.

Multi-Publisher. When a computation involves several publishers, it is important to have all the data from different publishers encrypted under the same key setting. Otherwise, the broker would not be able to perform homomorphic computation on the inputs from different publishers encrypted by different keys. Luckily, it is viable to handle this issue using proxy re-encryption. Once the encrypted data is re-encrypted, the data from different publishers can be used for homomorphic computation (i.e. these data are now considered as encrypted under the same key) and yield correct results, even though the re-encryption keys are different.

Multi-Subscriber. When a group of subscribers request the same computation, it is wasteful to recompute the result for each subscriber. It is quite straightforward that we can reduce the cost by only computing the result once and then distributing it to all subscribers having the same request. However, it would be challenging to do so under the scheme of homomorphic encryption since each subscriber has its own key pair. We noticed that this problem is similar to the asynchronization problem between publishers and subscribers but now among subscribers. Hence, we apply PRE (2 hops) and a key map between subscribers to circumvent repetitive computation. The multi-subscriber functionality works as in Figure 6. The scheme selects the subscriber with the smallest ID as the subscriber representative, and the broker performs the encrypted computation based on this representative's key pair. Before distributing the computation result, the broker re-encrypts the result for each subscriber using the re-encryption key associated with the representative and the subscriber. Each subscriber then decrypts it using its own private key to get the result.

Our final scheme can be found in Figure 7.

Initialization

- Each subscriber is assigned with a ID number. Each subscriber works with the broker to generate re-encryption keys for others subscribers whose ID numbers are larger than its.

Process

- Each time when a group of subscribers request the same computation, the broker selects the one with the smallest ID as the subscriber representative. The broker uses the representative X for computation as in the single-subscriber setting.

Re-Encryption

- Once the computation is finished, the broker re-encrypts the message using Pk_{rs} for each pair of X and one of other subscribers.

Decryption

- Each subscriber decrypts the message with its own private key.

Figure 6. Multi-Subscriber Functionality of Proxy-HE

Initialization

- Each client generates its own pair of public and private keys.
- Each publisher generates a re-encryption key Pk_{rp} associated with each subscriber and the broker updates the key map.
- Each subscriber is assigned with a ID number. Each subscriber works with the broker to generate re-encryption keys for others subscribers whose ID numbers are larger than its.

Publish

- Each publisher encrypts its data M using its own public key and sends its encrypted data along with a policy specifying allowed computations on its encrypted data to the broker.

Subscribe

- To subscribe computation C, each subscriber sends a subscription requesting C to the broker and its public key Pk_s. If the broker does not allow the subscriber to learn C's output, it sends an error message back to the subscriber.

Process

- **First Re-Encryption** Once the broker approves subscribers' request, it selects the representative subscriber X with the smallest ID in a group of subscribers requesting the same C. Then the broker reencrypts M using Pk_{rp} associated with X.
- The broker performs requested C on re-encrypted data.
- **Second Re-Encryption** The broker re-encrypts the message using Pk_{rs} for each pair of X and one of other subscribers.
- The broker sends the result of HE operations to subscribers.

Decryption

- Each subscriber decrypts the message with its own private key.

Figure 7. The Proxy-HE Scheme

4.3 Security Analysis

Our scheme PHE can guarantee both correctness and security under the adversary assumptions.

Correctness. The correctness of our scheme is built on the correctness of basic homomorphic encryption and multi-hop proxy re-encryption. If both hold true [10, 26], we can prove that our scheme is correct. Our scheme $PHE = (KG, RG, Enc, RE, Eval, Dec)$ can be proven correct: for all $(Pk, Sk) \leftarrow KG(1^\lambda)$ with the security parameter 1^λ, $Rk \leftarrow RG(Pk_B, Sk_A)$, all functions f and messages m in the message space M,

$$Pr[Dec(Sk_s, RE(Rek_s, Eval(f, RE(Rek_1, Enc(Pk_1, m_1)), \cdots, RE(Rek_n, Enc(Pk_n, m_n)))))) = f(m_1, \cdots, m_n)] = 1.$$

Privacy. As mentioned before, we do not consider the collusion adversary scenario where the compromised broker is able to collude with subscribers. Specifically, we assume the compromised broker has no access to any subscriber's secret key. We describe a simulator $\mathcal{S}$ that simulates the view of the adversary $\mathcal{A}_{\mathcal{PHE}}$.

In the real world, when $\mathcal{A}_{\mathcal{PHE}}$ compromises both the broker and a subset of publishers and subscribers, the bound of the number of publishers allowed to be compromised is $a-1$. This means that if there is only one honest publisher and the rest are all controlled by $\mathcal{A}_{\mathcal{PHE}}$, that honest publisher's input is unknown to $\mathcal{A}_{\mathcal{PHE}}$ due to the fact $\mathcal{A}_{\mathcal{PHE}}$ does not have access to any private key that can decrypt the data. In the case where $\mathcal{A}_{\mathcal{PHE}}$ compromises both any subset of the subscribers and a subset of publishers, the bound of the number is $a - 2$. $\mathcal{A}_{\mathcal{PHE}}$ has the final plaintext output of the computation and the rest of $a - 2$ publishers' inputs but is unable to infer the values of the 2 honest publishers' inputs. In the ideal world, $\mathcal{S}$ submits compromised publishers' inputs but learns nothing about the honest publishers' inputs. $\mathcal{S}$'s view in the ideal world is indistinguishable from $\mathcal{A}_{\mathcal{PHE}}$'s view in the real world.

5 System

$\mathcal{XYZ}$ is designed to work on top of one of the standard pub-sub protocols, MQTT, which allows communication in a pub-sub model arbitrated by a broker. Note that our work can easily be extended to other pub-sub protocols. In this section, we first discuss the general system design around MQTT and later explain the different system setups for the GC-Based Scheme and the Proxy-HE Scheme respectively.

5.1 General Design

System Implementation Around MQTT. MQTT allows subscribers and publishers to indirectly communicate with each other via the broker by publishers publishing data to topics and subscribers receiving it from topics after subscribing to them. In order to integrate our schemes into the MQTT protocol, we require each client (either publisher or subscriber) to have a device-specific topic that allows a two-way authenticated communication between each client and the broker. The broker (and the garbler in the GC-Based Scheme) will handle the computation and distribute data. We implement the broker using Eclipse Mosquitto 1.4.15 [23] and clients using Eclipse Paho 1.3 in Python [24]. Both of them support versions 5.0, 3.1.1, and 3.1 of MQTT. We implement the cryptographic components, namely garbled circuits, homomorphic encryption and proxy re-encryption in C/C++. In our system, the Mosquitto broker has to be configured using access control list file such that certain topics where publishers publish unprocessed data are inaccessible to other clients to protect private inputs from publishers and only authorized subscribers can have access to certain computation topics.

Supported Functions. We implement five functions in our system: mean, variance, weighted mean, private set intersection (PSI) and secure federated learning (FL). Mean and variance are the two standard statistical functions. Weighted mean is a function that calculates average where each data point has its weight and contributes to the final mean unevenly. Weighted mean is considered a simple version that can be extended to more complicated functionalities like secure machine learning. PSI is a function that can compute the intersection of different sets of private items from different parties without revealing any information besides the intersection. Secure FL helps securely aggregate model parameters for federated learning. We will explain the details on how we construct these functions in our system later in this section.

5.2 The GC-Based Scheme

Ratchet Keys. In the GC-Based Scheme, in order to improve synchronization and security of seeds, we adapt the key ratcheting protocol which can advance a secret key at every round and then deriving a seed from a ratchet key to generate pseudorandom strings. To set up the ratchet keys, the broker will forward the messages to the garbler such that clients can establish a ratchet key with the garbler. This design choice of relaying messages to the garbler through the broker is important to maintain the MQTT semantics on the clients. However, we need to add authentication in the MQTT messages using digital signatures and a key exchange protocol. This way, the secrets can be shared between the clients and the garbler via the broker. For publishers, this authenticated key exchange is used to derive the publisher's ratchet key. For subscribers, for every computation subscription, key exchange is performed to derive a key to encrypt the function ratchet key from the garbler.

Extending Libgarble. Libgarble [36] is a garbling library written in C based on JustGarble [8]. It extends JustGarble and adds new optimizations, such as, half-gates [50] to combine the free-XOR optimization with AND gates that only require two ciphertexts. As Libgarble is currently in development, it lacks some functionality which we have to add in order to build and garble circuits. On the garbling side, we implement the NOT gate (expressed as the XOR of the input with 1 to take advantage of the free-XOR optimization) and the OR gate. We add arithmetic blocks to be used when building circuits in order to allow signed fixed-point multiplication and signed fixed-point division. The motivation to operate with fixed-point numbers is to apply arbitrary functions based on arithmetic operations without the constraint of having just integer values in our GC-Based Scheme. Based on these implementations, we can build mean, variance and weighted mean for the scheme.

5.3 The Proxy-HE Scheme

PALISADE Setup. We use PALISADE library v1.10.5 [34] in our implementation for the Proxy-HE Scheme. PALISADE currently provides three different homomorphic schemes, namely BGV [11], BFV [22] and CKKS [18]. BGV is commonly believed to have better performance on integers than BFV does [13]. Thus, in our system, we choose BGV for integer operation and CKKS for real number operation. PALISADE also has built-in proxy re-encryption functionality for both schemes. Note that the PALISADE library supports multi-hop PRE [41]. In our evaluation, we slightly modified the default scheme parameters from PALISADE for benchmark purposes. Of course, these parameters can be tuned to have better system performance in the future. The detailed parameter configuration can be found in Table 2.

Scheme	BGV	CKKS
ring dimension	8192	8192
security level	HEStd_128_classic	HEStd_128_classic
multi depth	4	3
sigma	3.2	\
plaintext modulus	65537	\
scale factor bits	\	50
batch size	\	8

Table 2. Scheme Parameter Configuration: to fairly compare BGV with CKKS, we try to keep ring dimensions and CRT moduli the same; we choose the base 128-bit security in our evaluation; we also select the minimum viable values of multi depth for maximum possible multiplicative depth in our evaluation.

Extending PALISADE. The current PALISADE library provides basic functions like addition and multiplication. To better fit our needs, we extend PALISADE to have functions desired in our system as discussed. More functions can be included in our system in the future. We describe our implementation of the functions under the Proxy-HE scheme as follows:

- **Mean** In our implementation, we choose between BGV and CKKS for mean. It is straightforward to implement mean on CKKS. However, BGV can only perform secure integer operations such that we cannot calculate mean by multiplying the sum and the inverse of n (a real number). This means the computation of mean on BGV requires the plaintext of n. Thus, a mean implementation on BGV will leak the number of publishers involved in the computation. If this number is not sensitive, we can choose BGV as well.
- **Variance and weighted mean** Implementing variance and weighted mean on BGV shares a similar

averaging structure as mean. Thus, the scheme selection for these two functions is the same as above.
- **Private set intersection** We use the algorithm proposed by Chen et al [17] to construct our PSI. Since CKKS is the scheme of approximation, it will be hard to implement this PSI algorithm on CKKS (the step where we compare the result with zero can introduce errors with approximation). Due to this limitation of CKKS, currently we only consider BGV for PSI.
- **Secure federated learning** We use a similar secure FedAvg structure in [44] for our federated learning function. The current system uses CKKS.

Data Transfer. We use serialization [20] to save cryptocontexts, keys and encrypted data into binary files for data transferring. During the initialization, we create and store cryptocontexts and corresponding keys on each machine. At the publish-process-subscribe stage, MQTT transfers encrypted data in binary. The sizes of binary files are usually 100 kb to 200 kb.

6 Evaluation

6.1 Setup

We have evaluated $\mathcal{XYZ}$ in an environment where we run publishers, subscribers and the broker/garbler on three computers (Ubuntu 18.04.5 LTS with 2 Intel Xeon E5-2690 v4 cores and 4GB of RAM).

We have selected five functions of varying complexity (mean, variance, weighted mean, PSI and SFL) to evaluate the cost of the different schemes discussed above. For the Proxy-HE Scheme, both BGV and CKKS are evaluated. We evaluate these functions upon receiving the values (real number for garbled circuits and CKKS; integer for BGV) from a variable number of publishers and sending results to subscribers.

We put the cost into two categories: time cost for the actual computation and communication cost for the data transferred between clients and the broker as in the size of data. Our measured time includes the time of publishers encrypting data, the time of the broker evaluating data (also the time used for garbling in the GC-Based Scheme and the time used for re-encrypting data in Proxy-HE) and the time of the subscriber decrypting results. We use the size of the data exchanged to show communication costs on MQTT.

6.2 Results

In this part, we mainly focus on evaluating our system regarding its multi-publisher-multi-subscriber functionalities as well as its performance using different schemes.

In Figure 8 we show the cost results for the most relevant steps of the secure computation of the three numerical operations involving a varying number of publishers. In this part, we assume that one subscriber requests the computation. From our results, it is clear that for these three functions, the GC-Based scheme has a huge advantage in both time cost and communication cost as the number of publishers increases while the difference is not largely noticeable for a small set of publishers. This is because the most expensive steps of our GC-Based Scheme are the garbling and evaluation, which do not change much for multiple publishers. Among these functions, mean is the lowest cost operation; variance and weighted mean share similar results for both schemes. For two different implementations of Proxy-HE, CKKS has nearly double the time cost of BGV using the current version of PALISADE, but the communication costs are close.

To microbenchmark our multi-subscriber functionality, we test our system on mean operation for all three implementations. We here only compare the computation cost since the communication cost between the broker and the subscribers has a nearly linear relationship with the number of subscribers. Under our GC design, the number of subscribers should not affect the time cost in the view of the system. For Proxy-HE, the major cost of this functionality is the re-encryption part on the broker's side. As shown in Figure 11, the re-encryption is more expensive on CKKS than BGV in our system. The cost is noteworthy when having a large number of subscribers. For example, the multi-subscriber functionality costs around 102 seconds in total on CKKS for a system of 1000 subscribers. However, an alternative solution would be each subscriber parallelly re-encrypting the message instead of the broker doing all the re-encryption (around 20 ms for each subscriber on BGV and around 90 ms on CKKS). Of course, this would require each subscriber to maintain a re-encryption key map and gets the ID of the representative along with the message, which adds workload on subscribers.

For the distribution of costs, we show the results in Figure 9 and Figure 10. Since the cost distributions of variance and weighted mean are quite similar, here we only show the figures of mean and variance. Note that for the Proxy-HE Scheme here we only show the results of BGV for demonstration. We evaluate this part under the setting where multi publishers but one subscriber are involved to better illustrate the cost distribution. For the GC-Based Scheme, the most expensive part is the garbling, whose cost is usually double the cost of the evaluation part. In general, for Proxy-HE, re-encryption takes up a great portion of the overall cost and it holds a dominant position in mean. However, with more complicated functions, the homomorphic evaluation part increases the overall cost by a large margin (in variance and weighted mean) and costs PSI the most among all parts. In short, large-size input data has a large re-encryption overhead whereas complex computation has a large evaluation overhead.

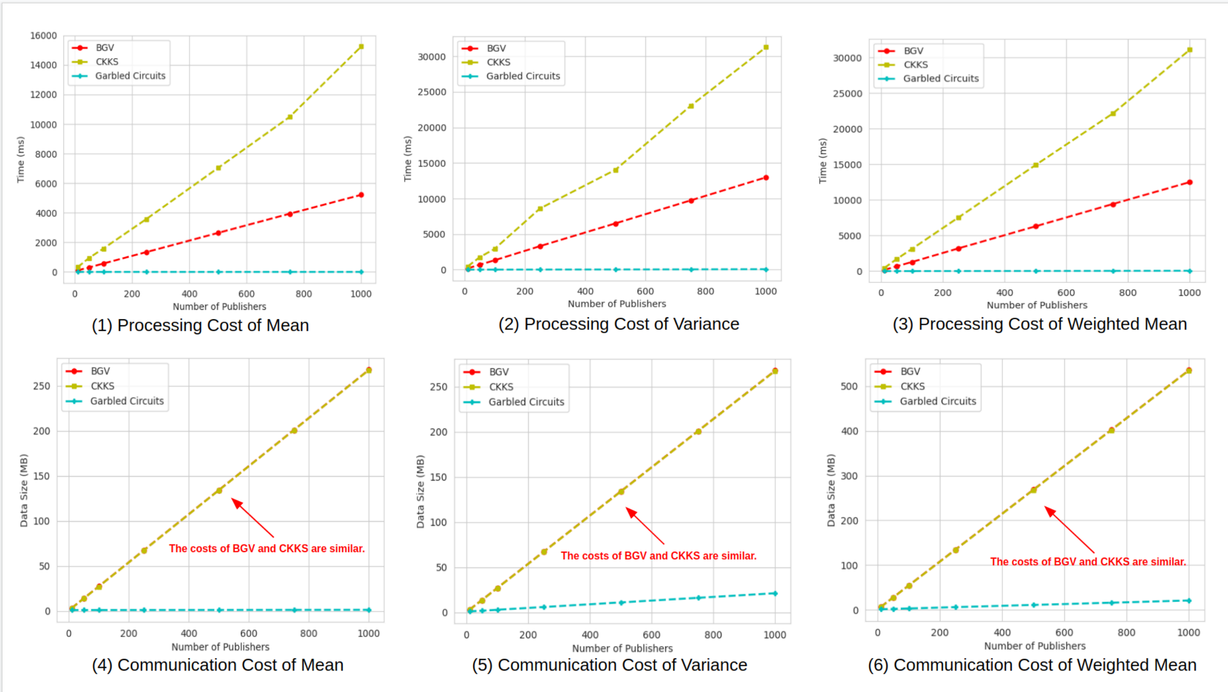

Figure 8. Microbenchmark Results: here we have the cost on the server of three basic statistical operations (mean, variance and weighted mean).

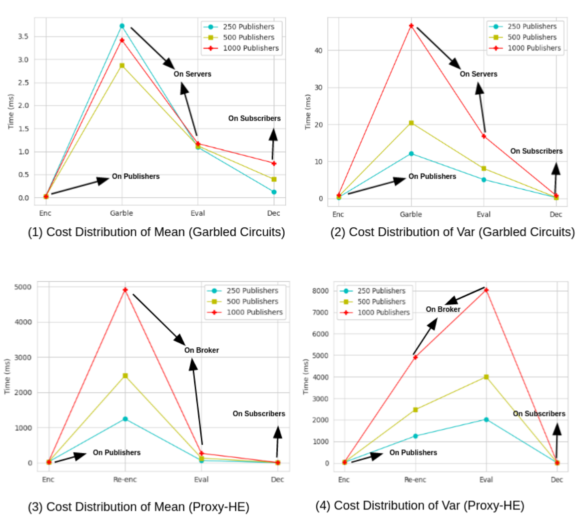

Figure 9. Distribution of Costs for Each Step

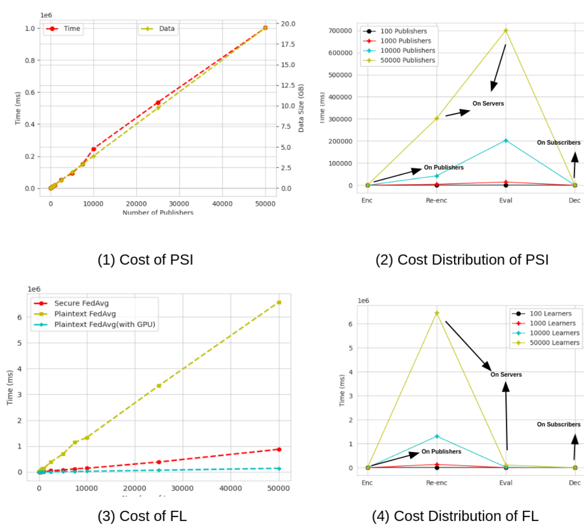

Figure 10. Microbenchmarks of PSI (BGV) and FL (CKKS)

At the moment we implement PSI on BGV implementation. Each publisher has an array with 10 elements. During the evaluation of PSI, the first publisher's data will be computed with the rest of the publishers' data iteratively. The result is shown in Figure 10. The cost of PSI in our BGV implementation is nearly linear against the number of publishers for both types of costs.

We test our FL function on CKKS on a medium-size convolutional neural network with 10 million parameters. We compare the cost of FL function in our system to the cost of plaintext FedAvg function (GPU comparison runs on Google Compute Engine backend) shown in Figure 10. The major performance drawback of our FL function is the re-encryption step because of input data size. Additionally, our Fl function can be seen as an encrypted form of the regular FedAvg with acceptable approximation loss, thus the performance of our model is similar to plaintext-trained models.

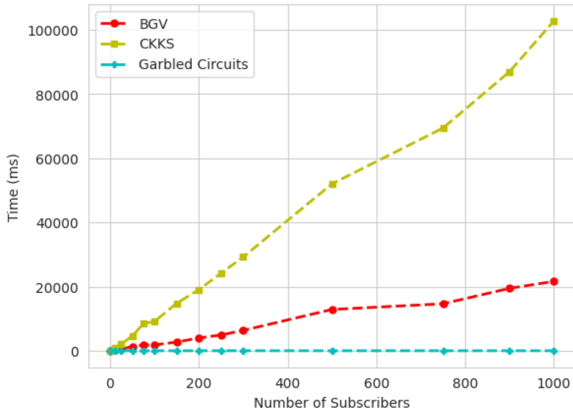

Figure 11. Microbenchmarks of Multi-subscriber Functionality

Encrypt	Evaluate	Decrypt	Data
26.537 ms	24.361 ms	106.101 ms	1.196 MB

Table 3. Cost of Contact Tracing in Our System.

6.3 Applications

We prepare concrete applications for IoT scenarios to show potential practical use of our system. For demonstration purposes, we use Proxy-HE Scheme here.

Contact Tracing. During the global pandemic, contact tracing becomes a promising tool to help identify the potential patients who might have contact with confirmed Covid-19 patients and slow down the spread of the virus. However, privacy is a major concern since the computation of regular contact tracing can reveal sensitive personal location data. We demonstrate a simple application using our system that implements PSI for contact tracing without the need for plaintext location information. Due to the sensitivity of the subject and the lack of available public datasets, we wrote a python program to generate random personal location datasets for testing purposes. Each person in the dataset has 10 visited locations in the same hour (in real cases the time granularity can be set to be more accurate).

In this application, we have two publishers (one is the confirmed patient and the other is the potential contact) with IoT devices recording their location data. These IoT devices, for example, can be smartwatches and smartphones. The subscriber can be public health authorities who are interested in contact tracing patients. The broker receives the encrypted location data from publishers, performs PSI on the data and returns whether two publishers have been in contact with each other.

In Table 3, we can see the cost of PSI in our system using BGV. The cost is low as we have a relatively small number of publishers but can be exceptionally high as the number of publishers goes up.

Daily Statistics of Parking Lots. The dataset for this application is the live status of the parking lots of a major airport [2]. In particular, the airport provides updates of the number of occupied and free parking spaces for each one of the 9 parking lots every 5 minutes. This makes a total of 288 published values per day per parking lot.

Statistics	BGV	CKKS	BGV	CKKS
Mean	47.0 ms	173.6 ms	115.1 MB	77.2 MB
Variance	2543.7 ms	4485.8 ms	115.2 MB	77.2 MB

Table 4. Cost Required to Evaluate Different Statistical Measures of the Parking Lot Dataset.

In this application, we are interested in obtaining daily statistics of the parking lots without revealing private data at fine time granularity. For this scenario, we have one publisher for one lot, which will be sending the current number of free and occupied spots every 5 minutes. We simulate the scenario by running it 288 times. The broker will accumulate the data from each day and compute the daily statistics.

Using the occupied spots data and the free spots data, we compute the mean and variance of the number of cars during a day. From these statistics, we can have an understanding of the parking lot's operation state from a daily perspective without invading detailed data. We observe that for the given amount of data, it can be computed in a short period of time shown in Table 4.

7 Conclusion

We present $\mathcal{XYZ}$, a secure publish-process-subscribe system with two multi-party computation schemes, i.e., the GC-Based Scheme and the Proxy-HE Scheme with different security assumptions and system constructions. To properly fit constraints from the traditional publish-subscribe structure, we also propose optimizations such as reduced communication extension and seed synchronization in the GC-Based Scheme and key exchange reduction along with multi-subscriber support in the Proxy-HE Scheme. Without the need for two third-party servers, our Proxy-HE Scheme has less system complexity than the GC-Based Scheme does, but yields larger overhead due to the time-consuming homomorphic encryption. Additionally, our system supports multiple publishers and multiple subscribers as well as provides an extensible library of several functions.

Our secure publish-process-subscribe system starts the conversation on integrating secure computation into IoT systems, but future work needs to be further considered on expanding supported functions as well as adding support for a distributed set of brokers.

References

[1] Mert Al, Thee Chanyaswad, and Sun-Yuan Kung. 2018. Multi-kernel, deep neural network and hybrid models for privacy preserving machine learning. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2891–2895.

[2] Los Angeles. 2017. *LAX parking lot live data*. https://data.lacity.org/dataset/Los-Angeles-International-Airport-LAX-Parking-Lots/dik5-hwp6

[3] Orlando Arias, Jacob Wurm, Khoa Hoang, and Yier Jin. 2015. Privacy and security in internet of things and wearable devices. *IEEE Transactions on Multi-Scale Computing Systems* 1, 2 (2015), 99–109.

[4] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. 2006. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Transactions on Information and System Security (TISSEC)* 9, 1 (2006), 1–30.

[5] G. Ayoade, V. Karande, L. Khan, and K. Hamlen. 2018. Decentralized IoT Data Management Using BlockChain and Trusted Execution Environment. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. 15–22. https://doi.org/10.1109/IRI.2018.00011

[6] Kannan Balasubramanian and Jayanthi Mathanan. 2018. Homomorphic Encryption Schemes: A Survey. In *Algorithmic Strategies for Solving Complex Problems in Cryptography*. IGI Global, 97–110.

[7] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference*. Springer, 1–18.

[8] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient Garbling from a Fixed-Key Blockcipher. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 478–492.

[9] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 784–796.

[10] Cristian Borcea, Yuriy Polyakov, Kurt Rohloff, Gerard Ryan, et al. 2017. PICADOR: End-to-end encrypted Publish–Subscribe information distribution with proxy re-encryption. *Future Generation Computer Systems* 71 (2017), 177–191.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 13.

[12] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Annual cryptology conference*. Springer, 505–524.

[13] Alycia Carey. 2020. On the Explanation and Implementation of Three Open-Source Fully Homomorphic Encryption Libraries. (2020).

[14] Henry Carter, Charles Lever, and Patrick Traynor. 2014. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 266–275.

[15] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. 2013. Secure outsourced garbled circuit evaluation for mobile devices. In *Proceedings of the 22nd USENIX conference on Security*. USENIX Association, 289–304.

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[17] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1243–1255.

[18] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[19] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2016. A Formal Security Analysis of the Signal Messaging Protocol. *IACR Cryptology ePrint Archive* 2016 (2016), 1013.

[20] AzothAmmo et al. 2020. *cereal - A C++11 library for serialization*. https://uscilab.github.io/cereal/

[21] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.

[22] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[23] Eclipse Foundation. 2021. *Eclipse Mosquitto 1.4.15*. https://projects.eclipse.org/projects/iot.mosquitto/releases/1.4.15

[24] Eclipse Foundation. 2021. *Eclipse Paho*. https://www.eclipse.org/paho/index.php?page=clients/python/index.php

[25] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Joerg Schwenk, and Thorsten Holz. 2014. How Secure is TextSecure? Cryptology ePrint Archive, Report 2014/904.

[26] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices.. In *STOC*, Vol. 9. 169–178.

[27] Ran Gilad-Bachrach, Kim Laine, Kristin E Lauter, Peter Rindal, and Mike Rosulek. 2016. Secure Data Exchange: A Marketplace in the Cloud. *IACR Cryptology ePrint Archive* 2016 (2016), 620.

[28] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2.

[29] Pascal Gremaud, Arnaud Durand, and Jacques Pasquier. 2017. A Secure, Privacy-preserving IoT Middleware Using Intel SGX. In *Proceedings of the Seventh International Conference on the Internet of Things* (Linz, Austria) *(IoT '17)*. ACM, New York, NY, USA, Article 22, 2 pages. https://doi.org/10.1145/3131542.3140258

[30] ISO/IEC 19464:2014 2014. *Information technology – Advanced Message Queuing Protocol (AMQP) v1.0 specification*. Standard. International Organization for Standardization.

[31] ISO/IEC 20922:2016 2016. *Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1*. Standard. International Organization for Standardization.

[32] Seny Kamara, Payman Mohassel, and Ben Riva. 2012. Salus: a system for server-aided secure function evaluation. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 797–808.

[33] Bhaskar Krishnamachari and Kwame Wright. 2017. The Publish-Process-Subscribe Pradigm for the Internet of Things. *USC ANRG Technical Report, ANRG-2017-04* (2017).

[34] Yuriy Polyakov Kurt Rohloff, Dave Cousins. 2020. *PALISADE HOMOMORPHIC ENCRYPTION SOFTWARE LIBRARY*. https://palisade-crypto.org/

[35] R Light. 2013. Mosquitto-an open source mqtt v3. 1 broker. *URL: http://mosquitto. org* (2013).

[36] Alex J. Malozemoff. 2017. *libgarble, Garbling library based on JustGarble*. https://github.com/amaloz/libgarble

[37] Masahiro Mambo and Eiji Okamoto. 1997. Proxy cryptosystems: Delegation of the power to decrypt ciphertexts. *IEICE transactions on fundamentals of electronics, Communications and computer sciences* 80, 1 (1997), 54–63.

[38] Javier Munster. 2018. *Securing Publish/Subscribe*. Ph.D. Dissertation.

[39] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. 2013. Privacy-preserving matrix factorization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 801–812.

[40] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 334–348.

[41] Yuriy Polyakov, Kurt Rohloff, Gyana Sahu, and Vinod Vaikuntanathan. 2017. Fast proxy re-encryption for publish/subscribe systems. *ACM Transactions on Privacy and Security (TOPS)* 20, 4 (2017), 1–31.

[42] PubNub 2017 (last retrieved). PubNub Real Time Data Stream Networks. http://www.pubnub.com

[43] PubNub Blocks 2017 (last retrieved). PubNub Blocks Real Time Computation. https://www.pubnub.com/products/blocks/

[44] Dimitris Stripelis, Hamza Saleem, Tanmay Ghai, Nikhil Dhinagar, Umang Gupta, Chrysovalantis Anastasiou, Greg Ver Steeg, Srivatsan Ravi, Muhammad Naveed, Paul M Thompson, et al. 2021. Secure Neuroimaging Analysis using Federated Learning with Homomorphic Encryption. *arXiv preprint arXiv:2108.03437* (2021).

[45] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.

[46] Marten Van Dijk and Ari Juels. 2010. On the impossibility of cryptography alone for privacy-preserving cloud computing. *HotSec* 10 (2010), 1–8.

[47] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L Wolf. 2002. Security issues and requirements for internet-scale publish-subscribe systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. IEEE, 3940–3947.

[48] Sophia Yakoubov. 2019. A gentle introduction to Yao's Garbled circuits.

[49] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 162–167.

[50] Samee Zahur, Mike Rosulek, and David Evans. [n.d.]. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 220–250.