Secure Publish-Process-Subscribe System for the Internet of Things

Anonymous Author(s)

ABSTRACT

Publish-Subscribe protocols, such as Message Queue Telemetry Transport (MQTT), play a key role in enabling real-time multi-point to multi-point communications for many emerging internet of things (IoT) applications. Recently, there has been an interest in adding in-network processing capabilities to such publish-subscribe systems to enable computation over real-time streams to enable sensor fusion, compression, aggregation and anonymization, correlation and other statistical analyses on sensor data. However, unlike pure publish-subscribe systems which can employ transport layer encryption end-to-end, it is challenging to ensure confidentiality with traditional in-network processing, particularly if the processing is done on an untrusted third-party cloud server. We present the first secure publish-process-subscribe system, which utilizes Yao's garbled circuits in conjunction with novel cryptographic and system techniques, to preserve the confidentiality of computations. We built a full-fledged secure publish-process-subscribe system using our cryptographic protocol as a foundation and incoproating novel system techniques. Our evaluation of the proposed system using several example IoT problems shows that our solution is computationally efficient.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; • **Computer systems organization → Embedded and cyber-physical systems**; **Sensors and actuators**;

KEYWORDS

publish-process-subscribe, secure computation, IoT

1 INTRODUCTION

As the Internet of Things (IoT) is emerging, rich new sets of applications are being envisioned that will require large-scale and complex deployments. These are motivating the rise of new network and middleware protocols to address key challenges including scale, inter-operability, security, and robustness [1, 37, 38].

For example, smart cities concepts are being proposed that incorporate many IoT-based deployments and applications. These include smart transportation applications that make use of real time sensor feeds from road traffic sensors (both infrastructure based and participatory mobile sensing) [26, 30], smart parking meters [20], and bike sharing systems that utilize sensor feeds on location of bikes [29]; applications aimed at monitoring and improving the environment using static and mobile sensors to measure air and water quality [12, 25]; automated water and power meter reading applications [21]; smart sanitation applications that use streams of sensor data from smart trash [17] cans indicating their utilization levels for timely removal of urban trash, etc.

The first generation of IoT systems involving large numbers of densely-deployed real-time streaming sensors such as traffic road sensor deployments [33], and air quality measurement [4] have largely focused on monolithic, vertically integrated applications with essentially a single collection point (which serves as an independent application-specific gateway for other applications to pull data from, in some cases through the provision of suitable API's, or simply through web-scraping).

However, there is a growing recognition that it is desirable to utilize application-agnostic middleware that allows the number of consumption points to scale. Thus, there has been a concerted effort to build and deploy IoT applications using real-time multi-point to multi-point middleware (that sits above the transport layer), and the most commonly used pattern for this kind of communication is the publish-subscribe paradigm [13]. Publish-Subscribe middleware allow multiple data consumers to connect to and receive streams of real time data from large numbers of sensors. Commonly used examples of publish-subscribe middleware protocols are Advanced Message Queuing Protocol (AMQP) [18] and Message Queue Telemetry Transport (MQTT) [19], and also on the rise are commercial publish-subscribe platform as a service (PaaS) providers such as PubNub [35] which provide their own proprietary protocol and API.

The basic idea behind publish-subscribe systems such as MQTT is that there is a broker (typically centralized and implemented on a cloud server, such as the mosquitto broker for MQTT [27], though there are also some distributed server implementations). Sensors publish messages to specified "topics" that are sent to the broker. Data consumers send to the broker a subscribe request to the specified topics by name. Following this, all messages published to any topic are forwarded by the broker to each subscriber of that topic. While we focus in this paper on sensing-oriented applications, note that pub-sub middleware could also be used to connect applications that generate control actions with actuators (in this case the actuator devices would be the subscribers and controlling applications would be the publishers). As one concrete example of the benefits of a pub-sub system, consider its possible use in a bike-sharing system; signals from bikes that are available could be published to a topic defined by spatial region (e.g. road stretch), and potential riders' devices can subscribe to topics corresponding to the locations near them, allowing them to find the location where they can pick up an available bike.

The broker in a traditional pub-sub system such as MQTT plays primarily an application layer forwarding role. It may also be enhanced to do some basic authentication of subscribers and publishers to ensure that only authorized users have access to send/receive relevant raw data streams from the broker.

More than a decade of research, particularly into wireless sensor networks [23, 40], has shown the value of incorporating in-network aggregation and processing into sensing applications. In-network processing allows for greater throughput and energy efficiency (through compression), more meaningful information to derived

from raw sensors (through collection of key statistics, multi-modal sensor fusion), anomaly detection, privacy (by allowing aggregation and anonymization of raw data). Prior work has largely addressed incorporating in-network aggregation in the context of data-centric pub-sub systems that operate at the network-layer, but there is a growing recognition that pub-sub middleware at the application layer can also benefit greatly from incorporating processing at the intermediate broker. In this new emerging publish-process-subscribe paradigm, the broker not only serves as a relay, but also enables computations over the raw streams so that the end subscriber receives real-time computed versions over one or more published streams [22]. This architecture is particularly appealing as it reduces bandwidth consumption and doesn't place computation on end points which may be resource constrained due to need for low-power and low-cost. Evidence of the growing appeal of this architecture is that the commercial PaaS pub-sub provider PubNub recently introduced processing capability into their system in the form of real-time compute functions they refer to as "BLOCKS" [36].

However, as brokers are typically hosted at third party servers, adding computational processing to pub-sub middleware introduces concerns about confidentiality. An application that wishes to make use of a third-party MQTT server for traditional pub-sub messaging could always encrypt the data end to end to provide confidentiality guarantees. But with computational functionality being moved to the server, these guarantees no longer hold. This is the focus of our work.

We propose a secure publish-process-subscribe protocol based on Yao's garbled circuits and private-set-intersection-cardinality (PSI-C). We present security definition for a secure publish-process-subscribe protocol in the REAL/IDEAL paradigm and provide a simulation based proof. We also develop extensions for reducing the communication and a private-set-intersection (PSI) based efficient wire label consistency protocol. We built a full-fledged system and developed a range of system techniques, such as, developing a new high-level functional language to describe computations and generate circuits, extending garbled circuits library libgarble, identity gates to make our protocol compatible with FreeXOR optimization, and establish authenticated and encrypted channel on top of MQTT.

Contributions.

- We propose the first provably-secure publish-process-subscribe protocol. We also present a security definition for a secure publish-process-subscribe protocol in REAL/IDEAL paradigm and a simulation based security proof.
- Using our protocol as a foundation, we develop a full-fledged publish-process-subscribe system with novel system components and report a comprehensive evaluation with real applications.

Organization. We describe related work in Section 2, propose a security definition for a publish-process-subscribe protocol in Section 3, detail our protocol along with extensions and security proof in Section 4, describe our system in Section 5, report evaluation using real applications in Section 6, and conclude in Section 7.

2 RELATED WORK

Nikolaenko, et al., [32] proposed a scalable privacy-preserving system for ridge-regression combining additive homorphic encryption and Yao's garbled circuits. In their setting, a single evaluator is interested in learning ridge regression over data of a large number of data owners without learning the indvidual data of data owners. Our setting and approach is different: (*a.*) we don't want to reveal output to the evaluator, (*b.*) we have multipe subscribers who want the output of computation, (*c.*) our data owners, publishers, are oblivous of subscribers and subscribers are oblivous of publishers, (*d.*) we support arbitrary polynomial-time computations as opposed to just ridge regression, and (*e.*) we don't necessarily need a third party `Garbler`, as showin in Figure 1b if a set subscribers controlled by a single entity can do a modest amount of computation. Nikolaenko, et al.,[32] proposed a similar system but for privacy-preserving matrix factoring.

Naveed, et al., [31] proposed a new primitive called controlled functional encryption inspired by functional encryption. Instead of using the same key for a function for computing the function over multiple ciphertexts, controlled functional encryption, require a fresh function key for every ciphertext. They construct an efficient controlled fucntional encryption scheme for arbitrary polynomial time computations based on Yao's garbled circuits and CCA2 secure public key encryption. Their setting invovles a data owner who wants a client to perform specific computations on it's data with the help of an online key authority. Their setting is similar to our setting of Figure 1b; however, in our setting publishers and subscribers are oblivous to each other.

Fully homomorphic encryption allows arbitrary compuatation on encrypted data. Gentry proposed the first fully homomorphic encryption scheme [8, 16] followed by several more efficient schemes [7]. Dijk, et al., [39] showed that privacy-preserving outsourced computation on data from multiple parties and supplying output to multiple parties, that is our setting, requires, in addition to fully homomorphic encryption, access-controlled ciphertexts and re-encryption. They reduce a scheme that computes on data from two parties and supply the output to two parties to black box obfuscation, which is impossible in general [5].

3 SECURITY DEFINITION

Before defining security, we present threat model that our definition needs to capture.

3.1 Threat Model

We assume that the adversary can *maliciously* compromise a subset of subscribers and a subset of publishers and *passively* compromise either `Broker` or `Garbler`, but not both. Malicious subscribers and publishers can collude either with `Garbler` or `Broker`, but not both.

3.2 Definition

We define security for a secure publish-process-subscribe protocol in REAL/IDEAL paradigm. Our definition captures both confidentilaity and correctness against adversary discussed above in the threat model.

Ideal World. Our ideal functionality $\mathcal{F}$ interacts with publishers, subscribers, `Broker`, and `Garbler` as follows:

- Each publisher sends a `policy` to $\mathcal{F}$ and each subscriber sends a subscription message to $\mathcal{F}$ containing subscribed computation C. Let S_C be the set of C's subscribers.
- The honest publishers upload data to $\mathcal{F}$. The malicious publishers controlled by `Adv` may abort (by not sending anything), sends their actual input, or send any arbitrary value that may depend upon their input, other malicious publishers' input, and auxiliary data. If a publisher value is not received before a time out period, $\mathcal{F}$ uses a nullifying value for its input, e.g., 0 for addition and 1 for multiplication.
- `Garbler` uploads to $\mathcal{F}$ a one-time use pseudorandom mask r.
- $\mathcal{F}$ determines a subset $P' \subset P$ of publishers whose data can be used to compute C. $\mathcal{F}$ sends to `Broker` P', policies of all publishers in P', and C.
- `Broker` sends $\mathcal{F}$ a subset $P_C \subset P'$ of publishers whose policies allow C.
- If data $\vec{x}_C$ of all publishers in P_C is enough to compute C, $\mathcal{F}$ sends $C(\vec{x}_C) \oplus r$ and r to all subscribers in S_C, otherwise $\mathcal{F}$ sends empty message $\perp$ to all subscribers in S_C.

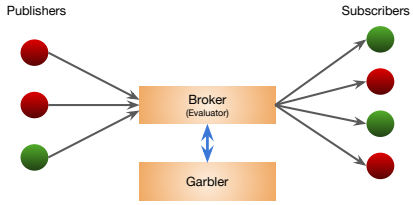
Real World. In real world, $\mathcal{F}$ is replaced by our protocol described in Figure 2.

Definition 1. *A publish-process-subscribe protocol is simulation secure if for every adversary* `Adv` *in the real world that maliciously corrupts a subset of publishers and a subset of subscribers, and only passively corrupts* `Broker` *and* `Garbler`*, with arbitrary collusion between malicious publishers, malicious subscribers, and hones-but-curious* `Broker` *and* no *collusion between* `Broker` *and* `Garbler`*, there exists a simulator* `Sim` *in the ideal world that also corrupts the same set of parties and produces an output identically distributed to* `Adv`*'s output in the real world.*

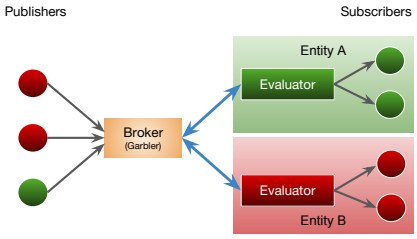
4 PROTOCOL

We propose two settings for secure publish-process-subscribe protocols : (*i.*) with an honest-but-curious external entity in addition to standard publish-subscribe entities, publishers, subscribers, and a broker, as shown in Figure 1a and (*ii.*) without such an external entity as shown in Figure 1b. First setting is suitable for less resourceful devices, such as, mobile phones, and the second setting is suitable for large organiazations with hundreds of subscribers. We describe our protocol and system in first setting, but they could be easily adapted to the second setting.

We describe a base secure publish-process-subscribe protocol first, for ease of understanding and reasoning about security, followed by extensions for reducing communication and efficiently handling malicious publishers disrupting the computation. We present a simulator for the base protocol to prove that a real world adversary cannot do more harm then a simulator can do in the ideal world (Definition 1). We also describe security of extensions. We



(a) Publish-Process-Subscribe setting with a separate entity, `Garbler`, to avoid expensive processing and communication at the subscriber's end. This is suitable for mobile devices. We describe our protocol and system in this setting.



(b) Publish-Process-Subscribe setting with only a `Broker`, just like traditional topic-based publish-subscribe systems. This is suitable for large organization with several subscribers who can set up local computation. While we describe our protocol and system in the setting of Figure 1a, they can be easily adapted to this setting.

Figure 1: Publish-Process-Subscribe System's Architecture

describe our system in Section 5, which is not a mere implementation of our protocol. We developed a range of system techniques to build our system.

We assume direct authenticated encrypted channels in this section. However, in our system, publishers and subscribers communicate directly only with `Broker` and communicate indirectly with `Garbler` through `Broker` as showing in Figure 1a. This allows us to implement our protocol on top of MQTT.

4.1 Base Protocol

The base protocol uses direct communication with `Garbler` for ease of explanation; publishers and subscribers in our system communicate with `Garbler` only through `Broker`. This ensures compatibility with a standard topic-based publish-subscribe system where publishers and subscribers communicate only through `Broker`.

Each new publisher initializes by sending its `policy` specifying allowed computations on its data. To subscribe computation C, a subscriber sends a subscription request to `Broker`. To publish a value, 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 `Garbler`, and only w_b to `Broker`. `Broker` waits for a specified time period to receive all input wire labels after which `Broker` requests `Garbler` to garble the circuit for $XOR \circ C$. A malicious publisher can send inconsistent labels for a wire to `Broker` and `Garbler`, e.g., w_0 and w_1 to `Garbler` and a random wire label w_r to `Broker`. Inconsistent labels prevent `Garbler` from correctly evaluating the garbled circuit. We use private-set-intersection-cardinality (PSI-C) to determine inconsistent labels. `Garbler` hard-wires nullifying values for publishers

Initialize

- Each new publisher sends Broker a policy specifying allowed computations on its data.

Subscribe

- To subscribe computation C, subscriber sends a subscription request containing C to Broker. If Broker allows subscriber to learn C's output, it adds the subscriber to a list of C's subscribers.

Publish

- To publish kth value, publisher generates two pseudorandom wire labels, w_0 and w_1, for each bit of the value.
- For each input bit b, publisher sends only w_b to Broker and sends both w_0 and w_1 to Garbler.

Process

- Broker waits for a specified time period t to receive input wire labels, a single label w_b for a bit b, from a subset of publishers P_C whose policies allow C. After time period t, Broker sends Garbler identifiers of publishers in P_C along with identifiers of publishers whose data wasn't received during time period t and requests Garbler to garble circuit for $XOR \circ C$. XOR is used to mask the output of the circuit.
- Broker sends Garbler the set of C's subscribers S_C.
- 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, received from publishers in set P_C hard-wiring nullifying value for publishers whose input labels weren't received by Broker as well as any publishers whose labels weren't received by Garbler. Garbler generates a random mask r and use it to mask the output o of C, such that evaluating GC would result in a masked output $o + r$.
- Garbler sends r to all subscribers in the set S_C and GC to Broker along with identifiers for publishers for which it hard-wired nullifying values.
- For every bit b of the C inputs, Garbler and Broker run a private-set-intersection-cardinality (PSI-C) protocol to determine wire labels consistency, i.e., if Broker input label for bit b, w_b, is one of the two Garbler labels for bit b, w_0 and w_1. If PSI-C outputs 1, then the labels are consistent and if it's 0 then the labels are inconsistent. Garbler will use nullifying values for inputs of all publishers with at least 1 inconsistent wire label.
- Broker evaluates the garbled circuit using wire labels sent by publishers in set P_C ignoring labels for which Garbler hard-wired nullifying values, obtains masked output $o \oplus r$, and sends $o \oplus r$ to all subscribers of computation C.
- Subscribers in the set S_C use mask r to unmask the output o.

Figure 2: Base Protocol

who sent at least 1 inconsistent label and publishers who didn't sent labels to either Broker or Garbler. Garbler garbles the circuit such that evaluating it outputs a masked output and send it to Broker. Therefore, when Broker evaluates the garbled circuit, it only learns a masked output, which it forwards to C's subscribers. Garbler sends the output mask to C's subscribers, who can then unmask the output.

4.2 Security Proof of Base Protocol

We describe a simulator Sim that simulates the view of the Adv in the real world execution of our base protocol of Figure 2. Our security definition 1 and simulator Sim ensures both confidentiality and correctness.

The interesting case is when a subset of publishers are malicious and are colluding with honest-but-curious Broker. Garbler is non-colluding honest-but-curious. A malicious subscriber only receives output; it can only reveal its own output, which is also possible in the ideal world execution.

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

Sim receives from $\mathcal{F}$ $\mathcal{F}(M \circ C, \vec{x}_C)$, where M is an XOR masking function. Sim sends $\mathcal{F}(M \circ C, \vec{x}_C)$ to Sim_{GC} and obtains a fake garbled GC_{fake}. Sim generates a random string o_r of the same length as output. Sim sends $(GC_{fake}, r_0^0, \ldots, r_{2l|P_C|-1}^0, o_r)$ to Adv. 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, Sim creates a fake garbled circuit GC_{fake} that 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, Adv 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 adversary to distinguish between real and ideal executions.

A malicious publisher can choose arbitrary wire labels in the real execution; however, as long as the labels used in garbling are consistent with the labels used for evluation, the honest subscriber output will be indistinguishable in real and ideal executions. Our protocol ensures consistent wire labels.

Initialize

- Each new publisher generates and sends to Garbler a truly random seed s. This seed will be used to create wire labels without interaction.

Subscribe

- In addition to registering subscription with Broker, subscribers for a computation C also register with Garbler. Garbler sends a truly random seed s' for computation C and send it to every subscriber who subscribes for C; generating a new seed for the first subscription for computation C.

Publish

- To publish kth value, publisher generates two pseudorandom wire labels, w_0 and w_1, using seed s in 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 seed s; $2kL \leq i < 2(k+1)L$, L being the bit-length of a value.
- For each input bit b, publisher sends only wire label w_b to Broker.

Process

- Garbler independently generates input wire labels using seed s from each publisher contributing input and an output mask r using seed s' for the output.

Forward Secure Seeds

Following procedure ensures that seeds s and s' used above are forward secure, i.e., compromise of seed does not affect the confidentiality of past data. We adapt Signal, a popular secure messaging protocol, key ratcheting protocol for forward security.

- 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: Reduced Communication Extension with Forward Security

4.3 Reduced Communication Extension

We develop an extension to our base protocol, described in Figure 3, that allows publishers and Garbler to generate wire labels for all input bits of a circuit independently. Our system uses this protocol.

Publishers and Garbler shares a truly random seed s and use a pseudorandom number generator to independently generate two wire labels for each input bit, eliminating wire labels communication between publishers and Garbler. Similarly, subscribers for computation C shares a truly random seed s' and use pseudorandom number generator to independently generate output masks, eliminating output mask communication between subscribers and Garbler.

Wire Label Synchronization. This method requires synchronization between publishers, subscribers and Garbler; we explain in Section 5 how publishers and subscribers remain in synchronization with Garbler using a round counter.

Forward Secure Seeds. While this protocol significantly reduces publishers' and subscribers' communication with Garbler, an adversary stealing a seed s from a publisher and colluding with Broker compromises the confidentiality of all of the publisher's inputs, including past, current, and future. An adversary stealing the seed s' for computation C from a subscriber and colluding with Broker compromises the confidentiality of outputs of all executions of computation C.

We adapt key ratcheting protocol of Signal, a popular secure messaging protocol, to make all seeds s and s' forward secure, i.e., an adversary stealing a seed wouldn't be able to compromise confidentiality of any past inputs and outputs. An adversary stealing publisher's seed s and computation C seed s' would still learn all current and future inputs of the publisher and outputs for computation C. However, an adversary learns a compromised publisher's current and future inputs and outputs of all current and future computations to which a compromised subscriber is subscribed to, even without stealing the seeds. In other words, an adversary stealing seeds s and s' can only do as much harm in the real world execution as a simulator stealing the same seeds can do in the ideal world. Therefore, forward secure seeds, provide the same protection as our base protocol.

4.4 Efficient Wire Labels Consistency Check

In base protocol, Figure 2, we execute a new instance of private-set-intersection-cardinality (PSI-C) protocol for every input bit of the circuit, which is expensive. We present an efficient wire labels consistency checking protocol in Figure 4, which we use in our system.

This protocol uses Pinkas, et al., [34] oblivious transfer (OT) based private-set-intersection (PSI) as opposed to private-set-intersection-cardinality (PSI-C) used in base protocol and only Broker learns the intersection. Running PSI alone cannot detect if a publisher sends valid labels for different input bits, e.g., both labels w_0 and w_1 for input bit b of C to Garbler and $w'_{b'}$ label for a different input bit b' of C to Broker. To address this issue, we use PSI over masked labels. Garbler and Broker shares a truly random seed s'' with the first message of the PSI protocol. Garbler and Broker generates a pseudorandom mask r_b for each input bit b of C from this seed; Broker computes masked labels $w_b \oplus r_b$ for input bit b and Garbler computes masked labels $w_0 \oplus r_b$ and $w_1 \oplus r_b$ for each bit b. Broker learns all the consistent wire labels and requests Garbler to hard-wire nullifying values for all publishers with at least 1 inconsistent wire label.

For every input bit b of circuit C, Garbler has both wire labels w_0 and w_1, we call the set of these labels $W_{\texttt{Garbler}}$ and Broker has a wire label w_b, we call the set of these labels $W_{\texttt{Broker}}$. Input consistency check ensures that w_b is either w_0 or w_1.

- Broker and Garbler share an ephemeral seed s'' for wire label consistency check before every computation C.
- Broker generates a pseudorandom mask r_b, using seed s'', and computes $w_b \oplus r_b$. We call the set of these masked input labels $R_{\texttt{Broker}}$.
- Garbler generates the same pseudorandom mask r_b, using seed s'', and computes $w_0 \oplus r_b$ and $w_1 \oplus r_b$. We call the set of these masked labels $R_{\texttt{Garbler}}$.
- Broker and Garbler runs private-set-intersection (PSI) protocol with $R_{\texttt{Broker}}$ and $R_{\texttt{Garbler}}$ as inputs and only Broker receives the intersection.
- If cardinality of $R_{\texttt{Broker}} \cap R_{\texttt{Garbler}}$ is $|R_{\texttt{Broker}}| = |R_{\texttt{Garbler}}/2|$, then all wire labels are consistent.
- If cardinality of $R_{\texttt{Broker}} \cap R_{\texttt{Garbler}}$ is less than $|R_{\texttt{Broker}}| = |R_{\texttt{Garbler}}/2|$, then $|R_{\texttt{Broker}}| - |R_{\texttt{Broker}} \cap R_{\texttt{Garbler}}|$ labels are inconsistent. In this case:
  - Broker unmasks all masked labels in $R_{\texttt{Broker}} \cap R_{\texttt{Garbler}}$ to learn $W_{\texttt{Broker}} \cap W_{\texttt{Garbler}}$.
  - Broker sends all inconsistent labels $W_{\texttt{Broker}} \setminus (W_{\texttt{Broker}} \cap W_{\texttt{Garbler}})$ to Garbler.
  - Garbler hard-wires in the garbled circuit GC nullifying values for inputs of all publishers with at least 1 inconsistent label.

Figure 4: Efficient Wire Label Consistency Checking Protocol

5 SYSTEM

Our protocol implementation works on top of the IoT *MQTT* protocol, which allows exchanging messages in a publisher-subscriber model arbitrated by a Broker. In order to set up the ratchet keys, both Subscribers and Publishers need a private communication channel with the Garbler. In order to embed this communication into the MQTT protocol each client (either Publisher or Subscriber) will have a device-specific topic that will allow a two-way authenticated communication between each client and the Broker. The Broker will forward the messages from this channel to the Garbler so that clients can establish a ratchet key with the Garbler. This design choice helps us maintain the *MQTT* semantics on the clients, since they only exchange messages with a single Broker at the *MQTT* level.

Although *MQTT* can be used on top of TLS to provide secrecy, authenticity and forward-secrecy; when clients obtain the ratchet keys from the Garbler the communication is forwarded by the Broker at the application layer, so we need to protect it to maintain secrecy, authenticity and forward-secrecy against the Broker. For this reason we choose to add authentication in the *MQTT* messages by means of public key signature and we use a key exchange protocol to share secrets between Clients and the Garbler. In the case of Publishers, this authenticated key exchange is used to derive the Publisher' ratchet key. In the case of Subscribers, for every function subscription, a key exchange is performed to derive a key to encrypt the function ratchet key sent from the Garbler to the Subscriber.

Ratchet keys. The use of shared ratchet keys between Publishers and the Garbler, and between the Subscribers and the Garbler allows us to maintain forward secrecy at the published value level and at the function output level at a very low cost. With this we achieve the property that if an attacker gets hold of the secret keys from any participant in the protocol, this attacker will not be able to decrypt past values even if they have recorded all previous communications. Ratchet keys work by advancing a secret key every at round (by means of using the preimage-resistance property of a cryptographic hash function). At any round, a seed can be derived from a ratchet key to be used to generate pseudorandom byte strings. The idea of using ratchet keys to achieve forward secrecy has been popularized by the Signal messaging protocol (already analyzed by the security research community [9] [15]), from which we take inspiration.

This design achieves the desired cryptographic properties of TLS for selected parts of the messages underlying the *MQTT* communication, allowing the Broker to be aware of the operation without being able to reveal or tamper with any ratchet key.

Seed syncrhonization. To maintain syncrhonization of the ratchet keys between the clients and the Garbler, when sending values, the Publisher adds the round of the ratchet key used 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, so that the Garbler can advance the Publisher ratchet key accordingly to derive the same seed and generate matching labels. Similarly, the Garbler tells the Broker the function ratchet key round used to generate the mask, so that the Broker can forward this information to the Subscribers which in turn advance their stored ratchet keys to derive a matching mask.

During the Publisher setup phase, not only its ratchet key will be established but the Broker will also register a special publishing topic that the Publisher will use to submit its encrypted values. Every time the Broker evaluates a garbled circuit using Publishers encrypted values as inputs, it will publish the masked result to the relevant function topic so that the interested Subscribers can receive it after having subscribed to that function topic.

The *MQTT* messages for our protocol are encoded in JSON, with all the binary data encoded in base64. A possible improvement in performance and bandwidth usage would consist on replacing this encoding for a binary one such as Protocol Buffers.

The block cipher behind the garbled circuit in our implementation is AES-128 (both blocks and keys are 128 bits). In particular, the block size has a direct relation to the size of the garbled circuit (as every non-XOR gate requires a fixed number of cipher blocks) as well as the required bandwidth between Publishers and the Broker, and between Subscribers and the Broker. In particular, every secret bit transferred from or to a client (which will either be an input bit or an output bit of the garbled circuit) will be expanded to 128 bits.

5.1 Extending *libgarble*.

libgarble [28] is a garbling library written in C based on *JustGarble* [6] that is in current development. It extends *JustGarble* by

adding two different size optimizations in the garbled circuit: Half-gates (which combines the free-XOR optimization with AND gates that only require two ciphertexts) and Privacy-free garbling (which achieves the free-XOR optimization with AND gates that only require one ciphertext at the cost of losing privacy and only guaranteeing authenticity). Apart from restructuring the code to make it more concise, it also offers a better API than *JustGarble* to build circuits.

Nevertheless, because *libgarble* is in current development, it lacks some functionality which we had to add in order to build circuits and garble them according to our needs. On the garbling side, we implemented the NOT gate (expressed as the XOR of the input with 1 to take advantage of the free-XOR optimization) and the OR gate. Our implementation doesn't use the half-gates optimization due to the fact for this mode of operation, *libgarble* in its current design only reserves two ciphertext per gate, not allowing the implementation of OR gates in a straight-forward manner.

We added some arithmetic blocks to be used when building circuits in order to allow a wide variety of functions to be described: **signed fixed point multiplication**, **signed fixed point division** and **signed min/max**.

The motivation to operate with fixed point numbers was to be able to apply arbitrary functions based on arithmetic operations without the constraint of having just integer values. Even though it would be easy to scale input values to be integers maintaining the desired precision, it's common in several algorithms used in our applications to have intermediate values with decimals that are significant.

A similar reasoning goes for supporting signed numbers. We use the common two's complement to represent such numbers, which work straightforward for addition and integer multiplication but require special changes for fixed point multiplication, division and min/max.

Whereas expanding **multiplication** and **min/max** to support signed fixed point numbers was not hard (for multiplication we need to fix the scaling of the result and for min/max we need to figure out the signedness of the inputs), implementing **division** becomes a challenge. Most of the fast division algorithms are not fit for combinational circuits, which is a constraint for the garbled circuits because such algorithms assume circuit loops which we will need to unroll, among other optimizations. For this reason we implemented the simplest non-optimized division algorithm: division by repeated subtraction with a fixed number of iterations (assuming the worst case).

5.2 High level language to build circuits.

While *libgarble* exposes the functions needed to build circuits corresponding to the functions we are interested in, the process can become cumbersome and error prone. To facilitate building circuits from functions we have implemented an interpreter for a functional programming language inspired in Scheme, a dialect of Lisp. This choice was motivated both by the simplicity of parsing a Lisp-like language as well as the similarity between the descriptive nature of functional languages with combinational circuits, which helps reasoning about how the circuit is built. Moreover, the availability of high-order functions in the language (functions that take functions as arguments and/or return functions as a result) allows us to express circuits that combine different building blocks very concisely.

The interpreter is written in Go in just 470 lines of code, and it is based on a minimal Scheme interpreter published in an open source license[1].

Other solutions to express garbled circuits in higher level languages than the underlying gates themselves have been proposed; for example, *Obliv-C* [41]. *Obliv-C* allows the developer to embed the secure computation part of a protocol in C code with a GCC wrapper that takes care of the compilation, the circuit building and the integration in a distributed program. We don't require this embedding in our protocol because the secure function is directly described independently. We also have a different setting, in which one of the parties doesn't learn the circuit result.

```
(begin
  (define fold
    (lambda (f l)
      (if (equal? (cdr l) ())
        (car l)
        (f (car l) (fold f (cdr l))))))

  (define min
    (lambda (l)
      (fold min2 l)))
)
```

Listing 1: Definition of the *fold* high order function with an example of its usage with *min2* to define a new function.

In listing 1 we show a snippet of circuit building code, where the only *libgarble* building block is `min2` (which takes two signed integers and returns the smaller one). `fold` is a high order function that recursively traverses a list `l` combining its elements with a function `f` to build up a return value.

```
(begin
  (define t1 (val "Campus/CS/112/temp"))
  (define t2 (val "Campus/CS/114/temp"))
  (define t3 (val "Campus/CS/115/temp"))
  (define t4 (val "Campus/CS/221/temp"))
  (define t5 (val "Campus/CS/223/temp"))

  (define temps (list t1 t2 t3 t4 t5))

  (start-building)
  (min temps)
)
```

Listing 2: Example of a function over Publishers' values.

In listing 2 we show a circuit function definition that uses the functions defined in listing 1. From this function definition the interpreter is capable of constructing the circuit that evaluates the function as well as storing the list of publisher values required to evaluate it.

Four our implementation we decided to use the same representation for all numerical values: two's complement fixed point 32 bit numbers, with 26 bits for the integer part and 8 bits for the decimal part.

5.3 `Broker` and `Garbler`.

The `Broker` and the `Garbler` have been written in the Go programming language. The choice of Go was motivated on one hand due to the provided built-in concurrency facilities such as *goroutines*

[1] https://pkelchte.wordpress.com/2013/12/31/scm-go/

(light-weight threads) and *channels* (a primitive to send messages between *goroutines*), which are very appropriate for the networking server nature of both entities. On the other hand, this language offers the good performance of a compiled language with the benefits of memory safety (which is very welcomed when dealing with dynamic data structures).

The communication between the `Broker` and `Garbler` (to request the garbling of a circuit for Publishers inputs at some particular rounds) happens over a TCP RPC synchronously.

The `Broker` requires a regular *MQTT* `Broker` server which we take from an open source implementation [2]. We have modified it so that any received message whose topic starts with a name space reserved for our protocol will be handled according to the protocol description instead of being forwarded to the subscribers of that topic.

The `Broker` and `Garbler` are required to maintain replicated data structures to store information of every Publisher, Subscriber and function. Our design makes this requirement easy because all setup messages between clients and `Garbler` are forwarded by the `Broker`, allowing the later one to register the non-secret details of the setup.

The `Garbler` is in charge of maintaining Publishers ratchet keys in sync with Publishers, and Function ratchet keys in sync with Subscribers interested in that function. The `Garbler` also garbles circuits on demand for the `Broker`.

The `Broker` is in charge of storing Publishers encrypted values with their corresponding publisher ratchet key round, and of requesting the garbling of circuits to the `Garbler` (for those particular Publishers values rounds) to evaluate the garbled circuit and send the result to the subscribed Subscribers.

5.4 Publisher and Subscriber.

The Publisher has been written in C in order to minimize its memory footprint as well as its resource consumption, considering that the code may be running in a resource-constraint IoT device. For the *MQTT* part of the Publisher we have used the Eclipse Paho client library [14].

On the other hand, the Subscriber has been written in Go, assuming that it will probably be running on a more powerful device. The *MQTT* library used by the Subscriber is the same we used in the `Broker` čitemqttgo, which also provides client functionality.

Publishers and Subscribers follow a state machine to initialize themselves (i.e. for Publishers to obtain a Publisher ratchet key, for Subscribers to obtain the Function ratchet key of all desired functions). The initialization to establish the ratchet keys happens through a message exchange with the `Broker` and `Garbler` via *MQTT* messages published and received at a unique per-device specific topic. The device specific topic is formed using the Client's (Subscriber or Publisher) public key.

5.5 Identity Gates for FreeXOR Compatibility.

The free XOR optimization in garbled circuits require that for every wire, the XOR of the label for bit `0` and bit `1` be a unique secret constant *delta* value. Since input values are generated by independent Publishers using their own seed, it's not easy to achieve this pattern without adding complexity to the system. For this reason, the `Garbler` will generate one *identity* gate per input wire that will allow transforming the inputs sent by Publishers to garbled circuit inputs that follow the *delta* pattern.

The implementation of these identity gates has been written in Go and is integrated in the `Broker` and `Garbler`. Because it's not written in C, like the rest of the circuit garbling and garbled circuit evaluation, we consider this step not to be optimized for speed. Nevertheless, its computation overhead should be small and linear in the number of the circuit function inputs.

5.6 Cryptographic choices

For all the cryptographic operations we have used the *libsodium* C library [11] which is a fork of NaCl [10]. A particular feature of this library is that it offers basic cryptographic primitives with secure design choices for the underlying construction, algorithms and parameters. This provides a high level of usability and removes the possibility of some error-prone choices.

In particular, the library makes the following choices for the primitives we use.

- **Key Derivation Function**: *BLAKE2B* is used for the underlying hash function.
- **Public key signatures**: *Ed25519* elliptic curves are used.
- **Key Exchange**: Elliptic curve Diffie-Hellman with the *Curve25519* curve (that is, *X25519*) is used to obtain a shared secret, from which cryptographically secure keys are derived by applying the *BLAKE2B-512* hash function.
- **Encryption**: The *ChaCha20* stream cipher is used for encryption with *Poly1305* MAC to provide authenticity.
- **CPRNG**: Uses the *ChaCha20* stream cipher.

Even though *libsodium* allows us to use *AES* for encryption, we choose to stick with *ChaCha20* because it is faster in software-only implementations (which is the only option available in many IoT devices) and because it is not sensitive to cache-collision timing attacks by design.

The construction *ChaCha20-Poly1305* has also been proposed as a standard choice of ciphersuit in the upcoming TLS version 1.3, which gives us further confidence in the decision to use it in our protocol.

6 EVALUATION

6.1 Setup

We have evaluated our implementation in an environment consisting on two computers located in the same Gigabit Ethernet network. We run the `Broker`, `Garbler` and one Subscriber on the first computer, and all the Publishers in the second computer. We perform all the analysis on the performance of several operations in the `Broker` and `Garbler`, which run together in a computer running Arch Linux (kernel 4.8.17-grsec) with an Intel Core i7-6600U CPU with 16GB of RAM. In order to isolate the performance influence of the `Broker` and the `Garbler`, we have serialized the garbling and evaluation of the garbled circuit operations (forbidding evaluation operations while the `Garbler` is garbling), mimicking the situation in which the `Broker` and `Garbler` are run on different servers. We also forbid concurrent garbling operations and concurrent evaluation operations to avoid high fluctuations in the measures.

Since the `Broker` and the `Garbler` are running on the same computer, we won't observe the delay introduced by transferring

the garbled circuit over a physical network. For this reason we obtain an estimation of the time required to send the garbled circuit by simulating a transmission over a Gigabit bandwidth network. Since we know the size of the garbled circuit, we can easily compute the time required to transfer it over a Gigabit connection and add that delay to the measured time.

Our current implementation doesn't support the Private Set Intersection between `Broker` and `Garbler` used to guarantee that the labels sent by the Publishers are valid for the garbled circuit. In order to evaluate this step of the protocol we have run simulations of an implementation of the Private Set Intersection based on Oblivious Transfer extension [34], using the number of labels corresponding to 32 bit values.

In all our measured times, sending includes the marshaling and unmarshaling of the garbled circuit and associated data structures necessary for transmitting it from the `Garbler` to the `Broker` via RPC.

6.2 Microbenchmarks

We have selected 5 numerical operations of varying complexity (*summation, multiplication, mean, variance, minimum/maximum*) to evaluate the cost of the different parts of our implementation. We securely evaluate these functions over the values (encoded as 32 bit fixed point numbers) received from a variable number of Publishers.

In figure 5 we show the timing results for the most relevant steps of the secure computation happening between the `Broker` and the `Garbler` of the 5 numerical operations for varying number of Publishers from 10 to 1000. In particular, since we are encoding the Publishers values with 32 bits, the number of input wires will be the number of Publishers times 32. We would expect the evaluated functions to scale linearly in the three analyzed steps of the protocol. Whereas the *multiplication* shows the more clear linear scaling, other operations show a transitive state tending to linear, with the most extreme case happening on the *mean*. We attribute this later result to the fixed cost of the final division in the mean, which becomes the dominant overhead of the circuit until the number of Pulbishers start reaching the 1000. On the other hand, the *summation* is the most lightweight function, and doesn't show yet a stabilizing linear trend for the number of Publishers used in the experiment. Accompanying the garble (figure 5a) and the evaluate (figure 5b) we show the time it took to garble and evaluate the input identity gates. We have also decoupled the time spent following the Private Set Intersection between the `Broker` and `Garbler` from the sending time, and we show it in figure 5c.

We can see that the time spent dealing with the identity gates (encrypting and decrypting the inputs) makes a significant influence in the faster functions, being comparable in magnitude to the evaluation and garbling time of the *summation* (the input decryption time is even higher than the garbled circuit evaluation time for *summation*). We attribute this behavior to the fact that this part of the protocol is implemented in Go instead of C like the garbling and evaluation.

We can also see how the time spent on the Private Set Intersection protocol is higher than the time required for sending in the case of the *Mean*, *Min* and *summation*, although it seems that it will scale better than the later two functions when the number of Publishers well passes the thousands.

	Sum	**Mult**	**Mean**	**Var**	**Min/Max**
Reduction	06.63 %	09.76 %	06.87 %	09.68 %	14.47 %

Table 1: Number of gates reduction for each microbenchmark function we would obtain by using the half-gates optimization.

We can see in more detail a comparison of the time spent on garbling and evaluating the identity gates in figure 6. We expected the garbling to be roughly twice the times of evaluating (garbling involves encrypting two labels whereas evaluating involves decrypting one)

In figure 7 we show the number of non-XOR gates (that is, the gates that incur a garbling and evaluating and that increase the size of the garbled circuit) for every function. We can clearly see the relation between the number of the non-XOR gates and the cost of computing every function by comparing shape of the plot with timing results shown in figure 5.

In figure 8 we see the transfer size between the `Broker` and the `Garbler` for every function (that is, the size of the garbled circuit and the associated data structures like the masking round used). As expected, the shape is very similar to the number of non-XOR gates for function shown in figure 7. In the figure we can observe where the bottleneck of using garbled circuits come from: the need to transfer such amount of data over the network. This fact makes the choice of network bandwidth between the `Broker` and the `Garbler` critical, leading to very different results in performance (which is dominated by the sending time) depending on the choice. This figure also helps us understand the limits of using garbled circuits: the *variance* and *multiplication* garbled circuits for 1000 Publishers sizes are close to 700 MiB; considering the linear scaling we can easily predict the point where the garbled circuits would be too big to fit into the computer memory.

Considering the fact that we are not using the half-gates size optimization in our garbled circuits, we show in table 1 the reduction we would observe for the experiments with 1000 Publishers if we had used such optimization (computed by counting the number of AND gates used in each circuit). We observe the biggest reduction in the *Min/Max* functions, leading us to conclude that implementing the half-gates optimization would be a clear improvement overall by reducing the sending time of the garbled circuits.

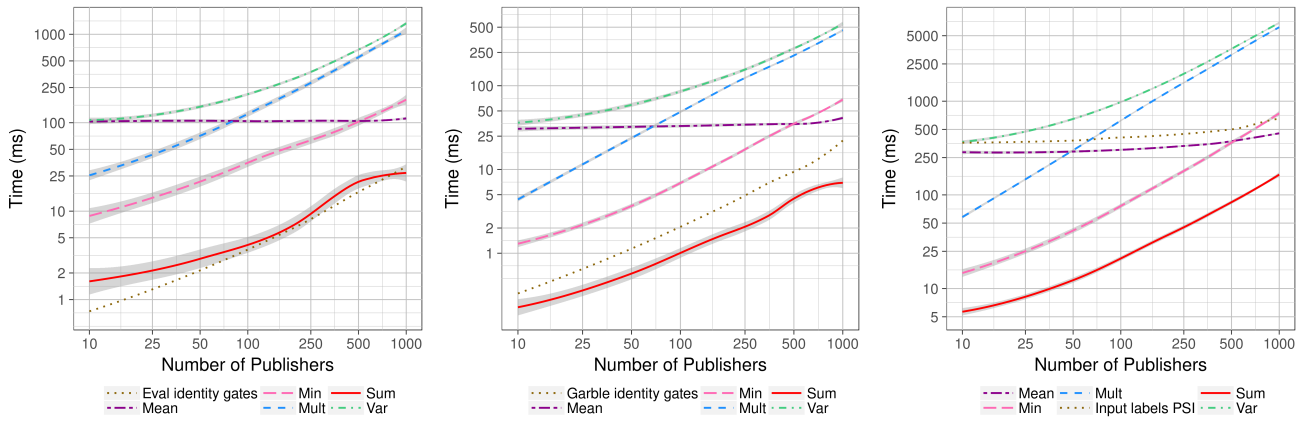
6.3 Applications

We have prepared 3 concrete applications based on real IoT datasets to show possible real usages of the presented protocol.

Wireless propagation constant

In this application we take the real time data provided by a testbed of IoT nodes deployed in our university forming a mesh network.

The mesh network is formed by a fixed number of nodes deployed in different static locations in a building that connect to each other via wireless. Every node tries to connect to the others unless they are too far, giving us an upper bound of $n \cdot (n-1)$ connections where n is the number of nodes. Each node periodically publishes the received signal strength (*RSS*) in dBm of the



(a) Garble (b) Evaluate (c) Send

Figure 5: Mean time required for garbling, evaluating and sending the garbled circuit to the `Broker` for each function in the microbenchmark. Results obtained from the mean of 5 repetitions for each configuration, with the confidence interval of 95% shown in gray.

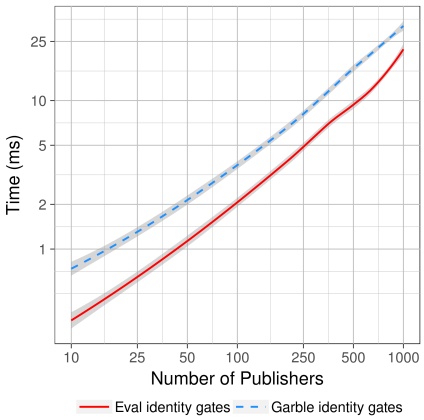


Figure 6: Time spent garbling and evaluating the identity input gates. Results obtained from the mean of all microbenchmark functions with 5 repetitions each, with the confidence interval of 95% shown in gray.

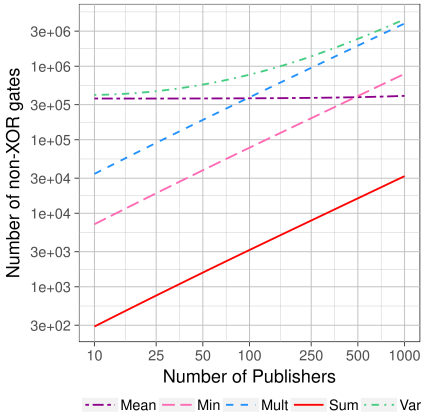


Figure 7: Non-XOR gates count per function used in the microbenchmarks.

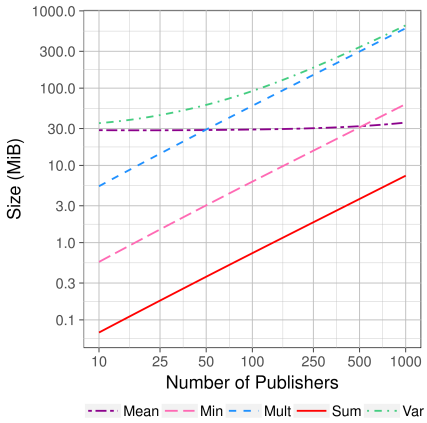


Figure 8: Size of the garbled circuit and the associated date required by the `Broker` to evaluate it.

nodes it is connected to; (this measure can fluctuate over time due to fading).

We are interested in estimating the wireless propagation constant (η) in the medium while preserving the privacy of the nodes (that is, their location, the distance between nodes and their reported signal strength measures). The value of η can be useful in characterizing the radio propagation environment and performance of the network and it can be derived from the following formula: $RSS = -10 \cdot \eta \cdot log_{10}(distance) - C$. We will estimate η by finding a linear model for the RSS-distances pairs obtained from the nodes.

In particular, we perform a linear regression so that we can model the RSS as a linear combination of a logarithmic function of the distance. To estimate the parameters of the one dimensional linear model ($y = ax + b$) we use the ordinary least squares technique, which gives us the following closed-form formula:

$$\begin{pmatrix} b \\ c \end{pmatrix} = \left(\sum_{i=1}^{n} \begin{pmatrix} 1 \\ x_i \end{pmatrix} \begin{pmatrix} 1 & x_i \end{pmatrix} \right)^{-1} \left(\sum_{i=1}^{n} y_i \begin{pmatrix} 1 \\ x_i \end{pmatrix} \right)$$

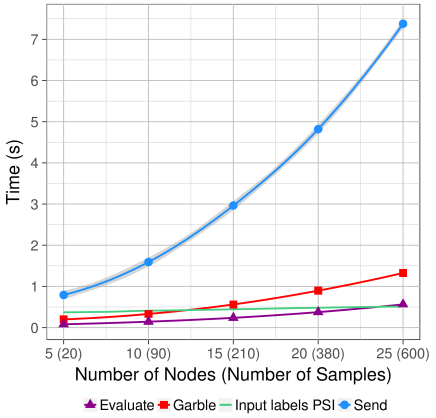


Figure 9: Mean time required for garbling, evaluating and sending the garbled circuit to the `Broker` for a varying number of nodes in the *wireless propagation constant* application. Results obtained from the mean of 5 repetitions for each configuration, with the confidence interval of 95% shown in gray.

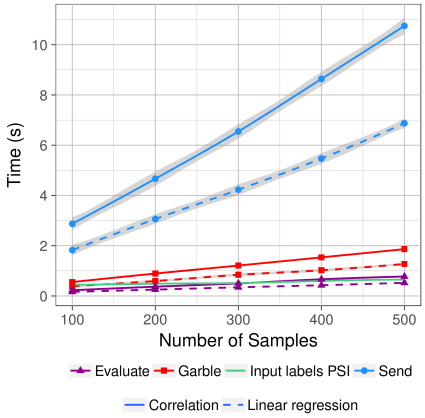


Figure 10: Mean time required for garbling, evaluating and sending the garbled circuit to the `Broker` for computing the squared correlation and linear regression of data streams. Garbling and evaluating includes the time to garble and evaluate the input identity. Results obtained from the mean of 5 repetitions for each configuration, with the confidence interval of 95% shown in gray.

Evaluating the formula requires an inversion of a $2x2$ matrix, which we perform by following the analytic solution.

For the evaluation of this application we construct a virtual scenario which simulates the IoT nodes by running one Publisher per node in a separate computer. The Publishers will publish the log_{10} of the distances to their peers as an initial step, allowing the `Broker` to store these values for some time avoiding the constant retransmission of these constants. After this, the Publishers will be sending values at the same rate of the nodes in order to obtain the same performance results we would get by using live data from the actual testbed. Following our protocol, we will be estimating the wireless propagation constant periodically, using the values received form the Publishers. We will vary the number of Publishers to analyze how this application scales (notice that the number of inputs for the linear regression formula grows quadratically when the number of nodes increases linearly)

We show the results of the evaluation in figure 9, where we can clearly see that the cost for this application quickly grows with the number of nodes. Our experiments end at 25 nodes because at at 30 Nodes we would be dealing with $30 \cdot 29 = 870$ samples to perform the correlation, which corresponds to a garbled circuit of size bigger than 1 GiB. At this magnitude we hit a limit on the encoding used when marshaling the garbled circuit in the Go RPC implementation, which doesn't support data structures bigger than 1 GiB. Notice however that we are using all the samples to calculate the linear regression even though a random sample of a smaller size would suffice; and that our scenario considers that all nodes are interconnected, which will usually not be the case when the number of nodes reaches a significant value.

Correlation of environmental indoor sensing data

For this application we will be using the environmental indoor sensing data provided by the Intel Berkeley Research lab [24]. This dataset provides 2.3 million readings collected from sensors with the following values: temperature in degree Celcius, humidity in percentage and light measured in Lux.

We are interested in finding relationships between pairs of data streams, each coming from a different sensor, while maintaining the individual readings private.

In this application's scenario, we assume that each sensor would be an individual Publisher that sends readings periodically. The `Broker` would accumulates streams of published values from each sensor, and when requested, the it would samples pairs of those streams at random over a specific period of time to perform a correlation analysis between the two. For our experiments we vary the number of samples and simulate the application by supplying the `Broker` with the given number of samples in a batch.

The correlation is a measure of statistical relationship among pairs of variables in which higher value represents higher dependence. Following the Pearson's product-moment coefficient, we can measure the dependence between samples of two variables (in our case, sensor readings) using the following closed-form formula:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2(y_i - \bar{y})^2}}$$

To improve efficiency, we evaluate the squared correlation (r_{xy}^2) to avoid computing the square root in the function circuit, which would be an expensive operation.

As a comparison, we will also evaluate the cost of computing the one-dimensional linear regression over two streams of values with a varying number of samples, which also gives us information about the relationship between the two data streams.

As we can see in figure 10 the cost of computing the correlation is roughly the double of the cost of computing the linear regression. Taking this result into consideration, we could devise a more efficient statistical relationship measure: for instance, we could first compute the linear regression and then find a normalized error between the linear model and the samples.

Daily statistics of airport parking lots

Statistic	Garble	Send	Evaluate	Size
Mean	199.8 ms	461.7 ms	98.7 ms	45.0 MB
Max/Min	163.3 ms	345.2 ms	84.9 ms	32.3 MB
Variance	500.7 ms	2376.3 ms	236.1 ms	222.4 MB
Rank free	121.3 ms	181.0 ms	67.5 ms	17.0 MB

Table 2: Mean time required for the different steps of the protocol to evaluate different statistical measures of the parking lot dataset. Garbling and evaluating includes the time to garble and evaluate the input identity. The mean time required to perform the Private Intersection Set of the input labls in all cases has been 1107.4 *ms*. **Results obtained from the mean of 5 repetitions for each configuration.**

The dataset for this application is the live status of the parking lots of a major airport [3]. 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.

In such application we are interested in obtaining daily statistics of the parking lots without revealing data at fine time granularity. We will only allow obtaining data accumulated throughout the day while preserving the privacy of the individual parking lots values.

For this scenario, we will have 9 Publishers, one for each lot, which will be sending the current number of free and occupied spots every 5 minutes. We simulate the Publishers by running them together in a single computer. The `Broker` will accumulate the data from each day and compute the daily statistics.

Using the occupied spots data, we will provide the *mean*, *min/-max*, and *variance* of the number of cars in all the lots combined during a day. Using the free spots data, we will compute which lot had more free spots and which one had less free spots on average during the day. We will refer to this later result as the *rank*.

We can see the results of the evaluation in table 2. We observe that for the given amount of daily data, the statistics can be computed in a short period of time; the `Broker` and `Garbler` could be computing daily statistics from data coming from many more sources. The most expensive operation is the *variance*, which coincides with the results obtained from the microbenchmarks; and the cheapest one is the *rank*.

6.4 Discussion

Bottlenecks. As expected, the time spent sending the garbled circuit from the `Garbler` to the `Broker` is the most expensive part of the protocol, and thus is the current bottleneck. This means that the quality and bandwidth of the network connection between the `Broker` and the `Garbler` is of critical importance.

The fraction (without considering the *PSI*) of time spent sending the garbled circuit varies depending on the circuit function and the number of Publishers, ranging from about 50 % as in the rank calculated in the *daily statistics of airport parking lot* to about 80% as in the *wireless propagation constant* application with 25 nodes.

When we consider the results of the *PSI* simulation, we observe that for the most lightweight applications it requires more time than all the garbled operations combined, whereas in the most heavyweight applications it can become as small as the 10%.

The total time required for computing the functions securely in our different applications experiments range from about two seconds as in the *daily statistics of airport parking lot* to a about 14 seconds as in the *correlation of environmental data* with 500 samples. Considering the fraction of time spent sending the garbled circuit in the more costly application configurations, we could get a good estimate of the total time by just knowing the number of non-XOR gates used (which would determine the size of the garbled circuit) and the network bandwidth available from `Broker` to `Garbler`.

The results obtained are perfectly suitable for computing several functions like the ones presented in the applications in a `Broker-Garbler` pair periodically. For example, 6 different applications similar to the ones shown could be running every minute. In our experiments, all the garbling and evaluating steps have been computed in serial in order to get result measures with the minimum fluctuation, but in a real system, several functions could be run at the same time allowing for concurrent garbling and evaluation making use of all the CPU cores available.

Optimizations. In general, the second most expensive part of the protocol is running the *PSI*. Decreasing the required time for this operation would have a significant decrease on the cost of our lightweight applications. We could take advantage of the fact that in our *PSI* setting, one of the parties (the `Garbler`) has the full set, and the other one (the `Broker`) a subset of it. Another possibility would be to try different ways of batching the labels in the *PSI*.

The garbling and evaluation of the identity gates could be optimized by incorporating the operations in the C code of libgarble. Even though the Go implementation of the AES encryption and decryption functions use the Intel AES-NI hardware instructions, conversions from byte vectors to AES blocks and vice versa slow down the operation. On the other hand, libgarble works natively with 128 bits data types (the size of an AES block) by using the Intel SSE2 extensions, thus not requiring any conversion during encryption/decryption.

7 CONCLUSION

We have proposed and implemented a secure computation protocol for the Publish-Subscribe model. By leveraging on the IoT *MQTT* protocol we can offer a protocol that is lightweight on the Publisher and Subscriber side.

From the evaluation of the system we conclude that it is computationally efficient to run both basic arithmetic functions like summation, minimum and variance; as well as more complex ones like correlation and linear regression securely on a significant number of input values. In particular, our system is able to compute the one-dimensional linear regression of 600 samples in 10.5 seconds, it can also compute 4 basic statistical parameters (mean, max/min, variance) on the daily accumulated data from streams publishing at 5 minutes interval in less than 7 seconds.

In general, the most expensive part of the protocol is sending the garbled circuit, being usually more than half of the overall cost in garbled circuits. The second most expensive part is running the Private Set Intersection required to detect malicious Publishers; which we believe could be improved in the future.

REFERENCES

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials* 17, 4 (2015), 2347–2376.

[2] Jeff R. Allen. 2016. MQTT Clients and Servers in Go. (2016). https://github.com/jeffallen/mqtt

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

[4] AQMD 2017 (last retrieved). Live Air Quality Data, South Coast Air Quality Management District, California. (2017 (last retrieved)). http://www.aqmd.gov/home/tools/air-quality

[5] 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.

[6] 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. https://doi.org/10.1109/SP.2013.39

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

[8] 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.

[9] 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. http://eprint.iacr.org/2016/1013

[10] Peter Schwabe Daniel J. Bernstein, Tanja Lange. 2016. NaCl: Networking and Cryptography library. (2016). https://nacl.cr.yp.to/

[11] Frank Denis. 2017. libsodium, A modern and easy-to-use crypto library. (2017). https://download.libsodium.org/doc/

[12] Prabal Dutta, Paul M Aoki, Neil Kumar, Alan Mainwaring, Chris Myers, Wesley Willett, and Allison Woodruff. 2009. Common sense: participatory urban sensing using a network of handheld air quality monitors. In *Proceedings of the 7th ACM conference on embedded networked sensor systems*. ACM, 349–350.

[13] 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.

[14] Eclipse Foundation. 2013. Eclipse Paho MQTT C Client. (2013). https://www.eclipse.org/paho/clients/c/

[15] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Joerg Schwenk, and Thorsten Holz. 2014. How Secure is TextSecure? Cryptology ePrint Archive, Report 2014/904. (2014). http://eprint.iacr.org/2014/904.

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

[17] Yann Glouche and Paul Couderc. 2013. A smart waste management with self-describing objects. In *The Second International Conference on Smart Systems, Devices and Technologies (SMART'13)*.

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

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

[20] Zhanlin Ji, Ivan Ganchev, Máirtín O'Droma, Li Zhao, and Xueji Zhang. 2014. A cloud-based car parking middleware for IoT-based smart cities: Design and implementation. *Sensors* 14, 12 (2014), 22372–22393.

[21] Tarek Khalifa, Kshirasagar Naik, and Amiya Nayak. 2011. A survey of communication protocols for automatic meter reading applications. *IEEE Communications Surveys & Tutorials* 13, 2 (2011), 168–182.

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

[23] L Krishnamachari, Deborah Estrin, and Stephen Wicker. 2002. The impact of data aggregation in wireless sensor networks. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*. IEEE, 575–578.

[24] Intel Berkeley Research lab. 2013. Berkeley indoor sensing data. (2013). http://db.csail.mit.edu/labdata/labdata.html

[25] Tuan Le Dinh, Wen Hu, Pavan Sikka, Peter Corke, Leslie Overs, and Stephen Brosnan. 2007. Design and deployment of a remote robust sensor network: Experiences from an outdoor water quality monitoring network. In *Local Computer Networks, 2007. LCN 2007. 32nd IEEE Conference on*. IEEE, 799–806.

[26] Guillaume Leduc. 2008. Road traffic data: Collection methods and applications. *Working Papers on Energy, Transport and Climate Change* 1, 55 (2008).

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

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

[29] Peter Midgley. 2009. The role of smart bike-sharing systems in urban mobility. *Journeys* 2, 1 (2009), 23–31.

[30] Prashanth Mohan, Venkata N Padmanabhan, and Ramachandran Ramjee. 2008. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 323–336.

[31] Muhammad Naveed, Shashank Agrawal, Manoj Prabhakaran, XiaoFeng Wang, Erman Ayday, Jean-Pierre Hubaux, and Carl Gunter. 2014. Controlled functional encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1280–1291.

[32] 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.

[33] PeMS 2017 (last retrieved). Performance Measurement System (PeMS) Real Time Traffic Data, Caltrans. (2017 (last retrieved)). http://pems.dot.ca.gov/

[34] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2014. Faster Private Set Intersection Based on OT Extension. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 797–812. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas

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

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

[37] Sabrina Sicari, Alessandra Rizzardi, Luigi Alfredo Grieco, and Alberto Coen-Porisini. 2015. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks* 76 (2015), 146–164.

[38] J. A. Stankovic. 2014. Research Directions for the Internet of Things. *IEEE Internet of Things Journal* 1, 1 (Feb 2014), 3–9. https://doi.org/10.1109/JIOT.2014.2312291

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

[40] Yong Yao and Johannes Gehrke. 2002. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod record* 31, 3 (2002), 9–18.

[41] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153. (2015). http://eprint.iacr.org/2015/1153.