PG: Byzantine Fault-Tolerant and Privacy-Preserving Sensor Fusion With Guaranteed Output Delivery

Chenglu Jin*
chenglu.jin@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, the Netherlands

Chao Yin*
c.yin@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
Centrum Wiskunde & Informatica
Amsterdam, the Netherlands

Marten van Dijk
marten.van.dijk@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, the Netherlands
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands

Sisi Duan
duansisi@mail.tsinghua.edu.cn
Tsinghua University
Beijing, China

Fabio Massacci
fabio.massacci@ieee.org
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
The University of Trento
Trento, Italy

Michael K. Reiter
michael.reiter@duke.edu
Duke University
Durham, NC, the United States

Haibin Zhang†
bchainzhang@aliyun.com
Yangtze Delta Region Institute of
Tsinghua University, Zhejiang and
Beijing Institute of Technology
Jiaxing and Beijing, China

ABSTRACT

We design and implement PG, a Byzantine fault-tolerant and privacy-preserving multi-sensor fusion system. PG is flexible and extensible, supporting a variety of fusion algorithms and application scenarios.

On the theoretical side, PG develops and unifies techniques from dependable distributed systems and modern cryptography. PG can provably protect the privacy of individual sensor inputs and fusion results. In contrast to prior works, PG can *provably* defend against pollution attacks and guarantee output delivery, even in the presence of malicious sensors that may lie about their inputs, contribute ill-formed inputs, and provide no inputs at all to sway the final result, and in the presence of malicious servers serving as aggregators.

On the practical side, we implement PG in the client-server-sensor setting. Moreover, we deploy PG in a cloud-based system with 261 sensors and a cyber-physical system with 19 resource-constrained sensors. In both settings, we show that PG is efficient and scalable in both failure-free and failure scenarios.

CCS CONCEPTS

• **Security and privacy** → **Privacy-preserving protocols**; *Distributed systems security*.

KEYWORDS

Garbled Circuit; Fault-Tolerant Algorithms; Sensor Fusion; Guaranteed Output Delivery

ACM Reference Format:
Chenglu Jin, Chao Yin, Marten van Dijk, Sisi Duan, Fabio Massacci, Michael K. Reiter, and Haibin Zhang. 2024. PG: Byzantine Fault-Tolerant and Privacy-Preserving Sensor Fusion With Guaranteed Output Delivery. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670343

1 INTRODUCTION

Numerous modern systems, spanning industry, agriculture, military, and beyond, are increasingly relying on distributed data sources (hereinafter without loss of generality, sensors) to support critical decisions and actions. As depicted in Fig. 1, the integration of these data is most often achieved with the help of a server (proxy, aggregator, or averager), which, upon receiving a client request, gets inputs from a set of sensors, integrates the inputs, and returns

*Both authors contributed equally to this research. Chenglu primarily contributed to P0 and P1, while Chao primarily contributed to P2 and P3.
†Corresponding author.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670343

the result to the client. Its application scenarios are almost everywhere, including sensor networks, smart metering, GPS devices and satellites, soldiers on battlefields, smartphones and the cloud, time-keeping mechanisms [58], and so on.

Privacy and *integrity* are widely regarded as primary concerns or even hurdles for many of these applications as studied in myriad papers across different areas [14, 22–24, 28, 31, 34, 35, 42, 43, 49, 60, 73, 74, 82]. First, we need to protect the privacy of individual data sources. Ideally, the client should only know the result of the function it has specified, but nothing more, and the server should learn nothing (about sensor inputs or the final result). Second, we require integrity in the sense that the server should faithfully return the correct, aggregated results to the client.

Applications requiring both privacy of individual sensor input and server integrity are numerous. Sensor networks deploying sensors for mission-critical applications (e.g., safety monitoring, target tracking) are natural examples, especially when sensors are deployed across multiple organizations [73]. Consider the smart metering application [70], where meters need to periodically report user data to a server. Individual meter inputs can easily reveal sensitive household information (e.g., habits). Meanwhile, the server may report false data to gain benefits. As another example, companies use the cloud to store and aggregate user data. While this appears to be a popular approach to modern business, privacy, and integrity concerns of user data are major hurdles to the broader adoption of user data collection and analysis. Major IT companies, such as Google [36], have deployed large-scale privacy-preserving systems for the collection of user data to compute aggregated statistics.

Correspondingly, an impressive amount of work on secure sensor fusion or data aggregation protocols have been proposed [1, 22, 23, 28, 31, 34, 35, 38, 42, 43, 49, 60, 73, 76, 82].

Pollution attacks. An equally important but notoriously difficult goal in multi-sensor fusion is to defend against *pollution attacks*, where some malicious sensors lie about their values to sway the final result. Specifically, motivated attackers can mount this kind of attack by either corrupting sensors and contributing malicious inputs or maliciously altering environmental variables.

At first glance, defending against pollution attacks seems to be at odds with attaining privacy. Indeed, to achieve the strongest privacy goal mentioned above, the server is not supposed to learn individual sensor inputs. Thus, the server cannot distinguish an incorrect malicious sensor input from a correct one.

In spite of the risk of pollution attacks, the vast majority of existing privacy-preserving systems treat defending against this attack as out-of-scope. Only a handful of prior work attempts to *mitigate* the problem [28, 50]. Their approach is to ask the sensors to provide a cryptographic proof to show that their inputs are in a prescribed range (or more generally satisfying some predicate)

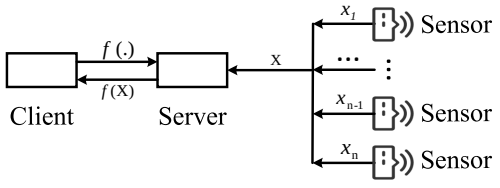

Figure 1: Distributed sensor fusion system architecture.

in the hope that a coalition of malicious sensors would not affect the final result by much. Take the average function as an example. Suppose we have ten sensors, each of which can have an input selected from the range [1, 100]. Also, assume that the "correct" value is around 20, and the correct sensors will output a value around this. If three malicious sensors contribute 100 (which is in the range), they would introduce a significant error in the final result. Moreover, for many applications, there are no prescribed limits on sensor inputs. To the best of our knowledge, all existing privacy-preserving aggregation and fusion schemes are vulnerable to pollution attacks to a significant degree.

Our approach. We design and implement PG, a privacy-preserving and Byzantine fault-tolerant sensor fusion system that 1) formally defends against pollution attacks, 2) performs within the computation and bandwidth limitations of sensors, 3) covers different application scenarios, and 4) is efficient and scalable in both failure-free and failure scenarios.

To prevent pollution attacks, instead of relying on validity proofs, our strategy is to "tolerate" so that no matter what inputs malicious sensors provide, the fused value represents the correct physical value with good accuracy, still in a privacy-preserving manner.

PG *combines* and *develops* techniques from distributed systems and secure multi-party computation. At the core of PG is (a privacy-preserving version of) Marzullo's Byzantine fault-tolerant (BFT) sensor fusion algorithm that takes as input sensor inputs and outputs a fused value. The fused value is guaranteed to contain the correct fused value, if at most g out of $2g + 1$ sensors are malicious. We also extend the framework of Yao's garbled circuits (GC) [79], which guarantees information privacy and computation integrity, from a two-party setting to a client-server-sensors setting.

As depicted in Fig. 1, we deconstruct GC by explicitly identifying three roles: clients (garblers), the server (the evaluator), and sensors (garbled input providers). More specifically, a client is responsible for generating a garbled circuit for Marzullo's algorithm; then sensors contribute garbled inputs; and finally, the server evaluates the garbled circuit using the garbled inputs and sends the client the garbled output. Our framework opportunistically leverages the bandwidth and computation asymmetry property in the sensor fusion setting, where the bandwidth and computation power at the client and the server sides are ample, but the bandwidth and computation power of sensors are usually limited.

In the above-mentioned system, both the server and a fraction of sensors may be malicious, but the server should not collude with malicious sensors [44, 45]. We stress that in our system, malicious sensors can collude, and in fact, we consider a strong adversary can coordinate malicious sensors to compromise the system. This assumption has been used in a large number of practical multi-party computation systems [20, 21, 28, 31, 44, 45, 64]. Our system, therefore, is suitable for applications where the server and sensors lack the motivation to collude, or the adversary lacks the means to corrupt both the server and some sensors simultaneously. For example, in the GPS system, the adversary may lack the means to compromise the satellite and some GPS devices together. In the context of smart metering, another example is that the utility company, concerned about its business reputation, lacks the motivation to collude with malicious meters.

Based on the simple framework, to provably defend against pollution attacks and achieve guaranteed output delivery, we have to tackle the following challenges:

- How do we achieve liveness and guaranteed output delivery when some sensors crash? In the context of garbled circuit evaluation, even if a single sensor fails to provide its garbled input, the server cannot evaluate the whole circuit and cannot deliver an output.
- How do we distinguish ill-formed garbled inputs from well-formed garbled inputs without compromising privacy? Malicious (Byzantine) sensors may contribute ill-formed garbled inputs so that the server can still evaluate the circuit, but the aggregated result will always be a meaningless $\perp$.
- How do we still guarantee privacy, correctness, and liveness when the server does not follow the protocol and acts maliciously? Can a malicious server trick other parties into revealing additional information?

In PG, we resolve all the above challenges and, in fact, describe our results following the above challenge pathway.

Our contributions. Our work on PG addresses the following key challenges:

- We design an efficient and scalable framework for privacy-preserving and Byzantine fault-tolerant sensor fusion. The framework leverages the bandwidth and computation asymmetry property in the sensor fusion setting and uses a novel combination of Byzantine fault-tolerant sensor fusion [56] and Yao's garbled circuits [79].
- We develop new techniques to achieve liveness and guaranteed output delivery in GC when a fraction of sensors are Byzantine malicious, meaning we cover all the cases where a malicious sensor can potentially influence the correctness or liveness of the system. Specifically, the malicious sensors can choose to provide no input, provide incorrect value with the correct encoding/garbling, or provide incorrectly garbled/ill-formed inputs.
- We also develop a new garbling scheme that allows us to extend our system to defend against malicious servers while keeping all the other security properties of the system.
- We make a general GC implementation specifically for our client-server-sensors setting by extending TinyGarble [75], which is one of the state-of-the-art two-party GC frameworks.
- We present a general methodology for designing optimized circuits to realize a variety of fault-tolerant algorithms.
- We conduct a large-scale scalability study on the AWS cloud with up to 261 sensors and another small-scale local evaluation on cyber-physical system with up to 19 sensors to demonstrate its performance in a wireless local network. The evaluation results show that the system is highly efficient and scalable in both environments and both failure-free and failure scenarios.

Paper organization. Sec. 2 describes the system setting and threat model. Sec. 3 introduces the preliminaries of our scheme, including garbled circuits and fault-tolerant algorithms. The protocol includes four versions which we describe in Sec. 4. Sec. 5 explains how we optimize the circuit designs of fault-tolerant algorithms. Sec. 6 contains our implementation and evaluation results. Related work is discussed in Sec. 7. Our paper concludes in Sec. 8.

2 SYSTEM AND THREAT MODEL

The setting. As depicted in Fig. 1, our system, PG, consists of a client, a server (aggregator), and a set of sensors. Client and server are denoted c and S respectively. We denote the number of sensors by n, and a bound on the number of faulty sensors by g. The set of sensors is denoted as $\Pi = (s_1, \cdots, s_n)$. Let l be the length of the sensor input. Let k be the security parameter of cryptographic primitives (AES key size in our implementation).

In PG, a client sends a request to the single server for computing a function $f(.)$, and the server collects readings from some or all of the sensors. The server then runs a sensor fusion algorithm and sends the aggregated result to the client. Throughout the entire protocol, the client and sensors communicate only with the server. In particular, there is no need for sensors to be aware of each other or to exchange information.

Threat model. A correct participant is one who faithfully executes our protocol until completion. It can nevertheless be *semi-honest*; namely, it conforms to the protocol but may additionally preserve the transcript of everything it observes in an effort to glean information to which it is not entitled. Malicious participants exhibit malicious behaviour limited only by the cryptographic assumptions adopted, also referred to as Byzantine participants. A malicious participant conforms to the protocol until some point at which it simply stops executing (permanently) is said to *crash*.

Semi-honest participants must be non-colluding (e.g., [44, 45, 68]), meaning they do not share information unless explicitly prescribed by the protocol. Byzantine sensors are allowed to collude with each other, but they are not permitted to collude with a malicious server, assuming one exists. This is a widely used assumption in a large number of practical multi-party computation systems [20, 21, 28, 31, 44, 45, 64]. Communication in our protocol happens in pairwise authenticated confidential channels to prevent information leakage from eavesdropping. Therefore, we consider general eavesdropping attackers or any malicious party in the protocol and eavesdrops on the communication between other parties out of the scope of the paper.

Two threat models, TM_A and TM_B, are considered in this work.

- **Threat model** TM_A: Up to g out of n sensors can be malicious. The relation between g and n depends on the concrete fault-tolerant algorithms used. Malicious sensors can collude with one another. Both the client and the server are assumed to be semi-honest, and they do not collude with the other parties.
- **Threat model** TM_B: In addition to the g malicious sensors, the server itself may also be malicious. Malicious sensors can still collude with each other, but they do not collude with the malicious server. The client is still semi-honest.

To achieve meaningful robustness against pollution attacks, the number of malicious sensors should be bounded (less than 1/2 or 1/3 of the total number, depending on concrete applications and algorithms). This assumption is a standard one in distributed systems and multi-party computation systems where sensors, ideally, are independently distributed in different hosts (running diverse software and hardware), and the adversary has only limited capacity to compromise a part of the overall system.

Goals. PG aims to achieve privacy, correctness, and liveness.

- **Privacy**: If a semi-honest client does not collude with the server and does not collude with sensors, then it learns only the aggregated result returned from the server but nothing more. If the server does not collude with the client and does not collude with sensors, then it learns nothing about sensor inputs or the final result.
- **Correctness**: If no more than g sensors are Byzantine, then the only response a correct client will accept is the correctly aggregated result that is guaranteed to contain the correct value (in the sense of Lemma 1 in the next section).
- **Liveness**: If the server is correct, and if no more than g sensors are Byzantine, then every correct client will get a non-$\perp$ response to its request. Note that liveness implies guaranteed output delivery, so we do not define it separately.

3 BUILDING BLOCKS

We describe the building blocks for PG—Garbling schemes [17] and Marzullo's algorithm [56].

3.1 Garbling Schemes

Garbling schemes allow two-party secure function evaluation by two mutually untrusted parties. A function must be represented as a Boolean circuit consisting of 2-input gates in order for the garbled circuit protocol to work[1]. In a garbled circuit protocol, a garbler and an evaluator contribute their private inputs for the function evaluation. The garbler garbles the function circuit while the evaluator generates the output of the function by evaluating the circuit based on the private inputs provided by both parties. Note that the inputs are also in a garbled form and do not reveal any information about the original private inputs of the two parties.

Bellare, Hoang, and Rogaway (BHR) [17] introduced the notion of a *garbling scheme* as a first-class cryptographic primitive. Here, we mainly adopt this abstraction but tailor it for our purpose; specifically, we require that *all* the garbling scheme algorithms, but Ev, be *dominated* by random coins.[2] The change is only notational. Note that in the presence of a dominant random coin, oblivious transfer protocol in traditional garbled circuit evaluation is not needed anymore. However, we will need to assume the evaluator/server to not collude with other parties to access the random coin.

A garbling scheme is a tuple of algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$. Gb takes as input 1^k, a random coin r and a Boolean circuit f, and outputs a garbled circuit F. En takes an input x and a random coin r and outputs a garbled input X. Ev takes a garbled circuit F and garbled input X and outputs a garbled output Y. De takes a garbled output Y and a coin r and outputs a plain-circuit output y (or $\perp$).

More specifically, when the garbler garbles a Boolean circuit, it assigns two labels for each wire in the circuit, and each label is actually a k-bit random string, where k is the security parameter of the scheme. The two labels are used to represent the truth values of 0 and 1 on the wire, respectively. For each gate in the circuit, the garbler generates one ciphertext per row in the truth table of a two-input gate. Each ciphertext is obtained by encrypting the output wire label using two input wire labels selected according to the gate's functionality. Similarly, when the evaluator evaluates the garbled circuit, it receives one label per input wire, decrypts the correct row in the garbled table, and gets the corresponding output wire label as the garbled output. A concrete example of a garbling scheme is a double-encryption scheme $(\mathbb{E}, \mathbb{D})$. $\mathbb{E}$ takes keys A, B, and a plaintext X, and returns a ciphertext $Y = \mathbb{E}(A, B, X)$, and $\mathbb{D}$ takes keys A, B, a ciphertext Y, and returns a plaintext $X = \mathbb{D}(A, B, Y)$.

We require a correctness condition on garbling schemes: if $\mathsf{F} \leftarrow \mathsf{Gb}(1^k, r, f)$, then $\mathsf{De}(r, \mathsf{Ev}(\mathsf{F}, \mathsf{En}(r, x))) = f(x)$.

In our work, we require the prv.sim (privacy), obv.sim (obliviousness), and aut (authenticity) security definitions in BHR, which we briefly describe here:

- prv.sim (privacy): There is a simulator $\mathcal{S}$ that takes as input $(1^k, f, f(x))$ and produces output which is indistinguishable from (F, X, r) generated normally.
- obv.sim (obliviousness): There is a simulator $\mathcal{S}$ that takes as input $(1^k, f)$ and produces output which is indistinguishable from (F, X) generated normally.
- aut (authenticity): Any adversary should not be able to generate a $Y' \neq \mathsf{Ev}(\mathsf{F}, X)$ such that $\mathsf{De}(r, Y') \neq \perp$.

3.2 Marzullo's algorithm

We describe Marzullo's algorithm [56], a fault-tolerant data averaging algorithm that is well-known in the reliable distributed systems community.

In the presence of n sensor values from n replicated sensors, a *fault-tolerant sensor averaging algorithm* [56] is used to compute a correct aggregated value, even if some of the individual sensors are incorrect (in which case the sensor is said to be malicious, Byzantine, or simply faulty). Marzullo [56] considered the case where each individual sensor value can be represented by an interval $I = [u, v]$ over the reals. Let $(u - v)$, the width of the interval, denote the *accuracy* (or *inaccuracy*) of a sensor. Let $(u + v)/2$ be the *midpoint* or *center* of the interval. Then, a sensor value is correct if the interval I contains the actual value of the measured feature, and the sensor is faulty otherwise. The goal of Marzullo's algorithm is to find the minimum (and correct) interval given n different intervals $I = \{I_1, \cdots, I_n\}$, with at most $g < n$ of those being faulty. The fused interval is at least as accurate as the range of the *least* accurate individual non-faulty sensors.

We start by introducing algorithms where the number of failed sensors g is known. The underlying idea is as follows: Since g or less sensors are incorrect, any $(n - g)$ mutually intersecting sensors (i.e., *clique*) may contain the correct value. The algorithm computes the "cover" (not the "union") of *all* $(n - g)$-cliques.[3] Let lo be the smallest value contained in at least $n - g$ of the intervals and hi be the largest value contained in at least $n - g$ of the intervals. Then, the correct aggregated result is the interval [lo, hi].

Marzullo [56] describes a general algorithm with $O(n \log n)$ complexity to compute this result. It uses a *sweeping* idea: First, sort all

[1] We are aware of the existence of the recently introduced arithmetic garbled circuit [7], but in this paper, we follow the convention and simply refer to Boolean garble circuit as garbled circuit.

[2] In BHR's original definition, only Gb is probabilistic, while the rest are deterministic. In their syntax, there are two more notations e (encoding information) and d (decoding information). For all the efficient garbling schemes known, both e and d can be generated by a single random coin together with some associate data.

[3] Picking the cover, instead of the union, can help preserve the shape of the sensor value.

the endpoints of all the intervals. Second, moving from the lowest value to the highest value, keep track of the number of intervals containing each value. The final result can then be determined from these counts. The algorithm cost is dominated by the sorting procedure.

The above algorithm can be applied to the case of arbitrary failures with unbounded inaccuracy or to the case of arbitrary failures with bounded inaccuracy, where the maximum length of the interval is known. Let M-g-U denote the algorithm that deals with unbounded inaccuracy failures. M-g-U needs $3g + 1$ sensors to tolerate g Byzantine failures with unbounded inaccuracies. We quote the theorem in [56] as a lemma below.

Lemma 1. *For a system with n sensors, if $g \leq \lfloor n/3 \rfloor$, then M-g-U outputs the smallest interval that is guaranteed to contain the correct value.*

4 PG PROTOCOLS

We describe our system PG and present four increasingly strong protocols from P0 to P3. P0, P1, and P2 achieve the privacy and correctness goals in TM_A, meaning that the client and the server are semi-honest and up to g sensors are malicious. However, only P2 can fully achieve the liveness goal when all g malicious sensors are Byzantine. P3, being the strongest version of PG, is secure against the threat model TM_B, where the malicious server is allowed to deviate from our protocol but not collude with malicious sensors.

In PG, a sensor fusion task is represented as a Boolean circuit. Please refer to Sec. 5 for the details of our circuit design. Here, we assume circuits that implement a fault-tolerant algorithm are ready.

Consider that there is a garbler (client), an evaluator (server), and several garbled data providers (sensors). The client specifies the functional circuit and generates the corresponding garbled circuit. The protocol starts with the client sending the garbled circuit to the server. After receiving the garbled circuit from the client, the server collects data (garbled inputs) from sensors. Garbled inputs prevent the server from knowing the truth value of the sensor measurements. The server then evaluates the functional circuit with these garbled inputs and sends the result of its evaluation (garbled output) to the client. The client decodes the garbled output and gets the plain sensor fusion result. In PG, only the client can decode the garbled output to a plaintext truth value.

4.1 P0: The Basic Protocol

In P0, the client is responsible for running Gb and generating a garbled circuit; then sensors run En and contribute the garbled inputs; and finally, the server evaluates (Ev) the circuit using the garbled inputs and sends the client the garbled output for the client to decode (De).

Each time the client wants to obtain a fused result of sensor inputs, the client and the sensors need to agree on a *fresh*, random coin r that is used to garble the circuit and garble the inputs, respectively, and they should prevent the value r from being known by the server. In the semi-honest model, we can easily achieve this by allowing the client to dictate the coin. In PG, we assume that the client shares an independent, symmetric, pairwise key with each sensor. A client chooses a random coin and wraps the coin using an authenticated encryption with the pairwise keys shared. The ciphertexts will be sent to the server, and they will be distributed to the respective sensors. Alternatively, we can assume public key infrastructure and our system can be easily adapted. Also, the client can send both the wrapped coins and the garbled circuit in the same round, saving one communication round.

Setup and inputs: Let fta be any fault-tolerant sensor fusion function. Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ be a garbling scheme. Let $\langle \text{REQ} \rangle$ be a client request that contains the function description. Let $x_\Pi = \{x_1, \cdots, x_n\}$ and $X_\Pi = \{X_1, \cdots, X_n\}$ be sensors' inputs and garbled inputs respectively.

00 Client c selects a random coin r, runs Gb (using r) to generate a garbled circuit F for fta, and sends F, $\langle \text{REQ} \rangle$, encrypted coins to server S.
01 S forwards $\langle \text{REQ} \rangle$ and the corresponding encrypted coins to sensors.
02 Sensors Π run En (using r and x_Π) and send S garbled inputs X_Π.
03 Server S runs Ev on X_Π and sends the garbled output Y to c.
04 Client c runs De (using r and Y) to get $\text{fta}(x_\Pi)$.

Figure 2: P0. When the server receives a garbled circuit, the server collects garbled inputs from the sensors and returns the garbled output to the client.

The above approach opportunistically leverages the bandwidth and computation asymmetry property in the sensor fusion setting, where the client and server have a much better network connection and a much stronger computation power than the sensors. It is common in modern systems to shift part of work to clients to improve the service throughput and reduce the latency. Moreover, in our approach, the circuit size (related to the accuracy of the returned results to clients) can be flexibly decided by clients. In addition, the client can potentially precompute the garbled circuit off-line to reduce online latency.

We describe the above system in Fig. 2, using a language of garbling schemes that is slightly modified from BHR. Without loss of generality, we make black-box use of a general sensor averaging function fta (instead of using Marzullo's algorithm described in Sec. 3). The difference between P0, Feige, Kilian, and Naor (FKN) [37], Kamara, Mohassel, and Raykova (KMR) [44], and Naor, Pinkas, and Sumner (NPS) [65] can be found in Sec 7.

While P0 achieves Privacy, it does not achieve liveness (guaranteed output delivery). This means that even if only one sensor fails to provide its garbled inputs, the server cannot start evaluating the garbled circuit and would have to wait for the missing input.

4.2 P1: Achieving Liveness in the Crash Failure Model

In Fig. 3, we describe P1 achieving liveness in the crash failure model, where sensors can fail by crashing. In P1, the absence of a reply from a sensor will be treated as an input of $[-\infty, +\infty]$ (or the prescribed upper and lower bounds), which means this reply will not be counted. The reason why we can do this is that our fault-tolerant algorithms can natively tolerate meaningless inputs as long as the number of these inputs (and together with malicious inputs) are g-bounded. More specifically, if the server does not receive the garbled input from some sensors in time, it will ask the client to send corresponding garbled inputs for the missing sensors for values $[-\infty, +\infty]$. The whole protocol is depicted in Fig. 3.

In synchronous environments, if the server is correct and if all malicious sensors crash, the server will request garbled inputs from the client after the timer expires. As the client is semi-honest,

Setup and inputs: Let fta be any fault-tolerant sensor fusion function. Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ be a garbling scheme. Let $\langle \text{REQ} \rangle$ be a client request that contains the function description. Let $x_\Pi = \{x_1, \cdots, x_n\}$ and $X_\Pi = \{X_1, \cdots, X_n\}$ be sensors' inputs and garbled inputs respectively.

10 Client c selects a random coin r, runs Gb (using r) to generate a garbled circuit F for fta, and sends F, $\langle \text{REQ} \rangle$, encrypted coins to server S.
11 Server S forwards $\langle \text{REQ} \rangle$ and the corresponding encrypted coins to sensors.
12 Sensors Π run En (using r and x_Π) and send S garbled inputs X_Π.
13 If S does not receive all the garbled inputs before the timer expires, it requests from the client the missing garbled inputs that encode $[-\infty, +\infty]$.
14 Server S runs Ev on X_Π and sends the garbled output Y to c.
15 Client c runs De (using r and Y) to get $\text{fta}(x_\Pi)$.

Figure 3: P1. The protocol is completed in one round in the failure-free scenario and in two rounds if some sensors fail to provide garbled inputs.

Setup and inputs: Let fta be any fault-tolerant sensor fusion function. Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ be a garbling scheme. Let $\langle \text{REQ} \rangle$ be a client request that contains the function description. Let $x_\Pi = \{x_1, \cdots, x_n\}$ and $X_\Pi = \{X_1, \cdots, X_n\}$ be sensors' inputs and garbled inputs respectively.

20 Client c selects a random coin r, runs Gb (using r) to generate a garbled circuit F for fta, and sends F, $\langle \text{REQ} \rangle$, encrypted coins to server S.
21 Server S forwards $\langle \text{REQ} \rangle$ and the corresponding encrypted coins to sensors.
22 Sensors Π run En (using r and x_Π) and send S garbled inputs X_Π.
23 If S does not receive all the garbled inputs before the timer expires, it marks the missing sensors as malicious in a list $list_m$.
24 Server S checks the validity of the received garbled inputs using the checking gates, and if it finds that any garbled input X_i of a sensor i is ill-formed, it marks the sensor i as malicious in the list $list_m$.
25 Server S sends $list_m$ to Client c to request well-formed garbled inputs that encode $[-\infty, +\infty]$ for the malicious sensors in $list_m$.
26 Server S runs Ev on X_Π and sends the garbled output Y to c.
27 Client c runs De (using r and Y) to get $\text{fta}(x_\Pi)$.

Figure 4: P2. The protocol checks the validity of received garbled inputs and completes in one round in the failure-free scenario and in two rounds if some garbled inputs are missing or ill-formed.

the server will receive these "dummy" garbled inputs and start evaluating the garbled circuit. The effect of these "dummy" garbled inputs will be tolerated by the underlying fault-tolerant algorithms.

What if sensors are Byzantine? We have addressed the issue that sensors do not contribute any garbled inputs; however, the sensors can be Byzantine malicious and may contribute ill-formed garbled inputs, i.e., the input is not one of the two valid labels on the input wire, but it looks indistinguishable from a random string without running a decoding procedure. Thus, modern GC frameworks will just treat the ill-formed garbled inputs as valid ones and finish the evaluation of the whole circuit, but in the end, the client can only end up getting $\perp$ due to a decoding failure. Although this issue does not violate any correctness or privacy properties, it does introduce an easy denial-of-service attack, and the behaviours of the malicious sensors are hidden behind the privacy guarantee of a garbled circuit, so neither the client nor the server can catch them. Note that all popular garbled circuit compilers or implementations [15, 17, 40, 54, 75, 80] do not provide a way of telling which garbled input is ill-formed.

4.3 P2: Efficient Garbling Schemes for Detecting Ill-Formed Garbled Inputs

To build a garbling scheme that detects ill-formed garbled inputs, we extend the conventional double-encryption scheme to realize false-key awareness. Once the semi-honest evaluator finds some of the sensor inputs are ill-formed, then it can also request the client for the corresponding garbled inputs for the malicious sensors for values $[-\infty, +\infty]$. It is just like how P1 treats missing sensor inputs. In fact, the evaluator in P2 compiles a list of the malicious sensors that send no or ill-formed inputs and shares this list with the client to request the missing inputs for starting the circuit evaluation. This guarantees the liveness of the system in threat model TM_A and, therefore, achieves guaranteed output delivery as well. The complete protocol for P2 is presented in Fig. 4.

False-key awareness means that given a double-encryption ciphertext, anyone can tell if a given key is false (i.e., ill-formed). We note that LP [52] defined a symmetric encryption scheme with an *elusive* and *efficiently verifiable* range. The goal of LP is to allow the evaluator to know which gate entry in a four-entry table corresponds to the given inputs. Not surprisingly, the LP construction also meets the requirement for false-key awareness. Specifically, the evaluator can simply check if the decrypted result is within the pre-defined *elusive* and *efficiently verifiable* range. However, to our knowledge, it has not been implemented for this purpose. All the popular GC implementations and compilers do not achieve false-key awareness. These implementations use the point-and-permute technique to identify the correct entry [11]. Below, we provide two schemes that efficiently achieve false-key awareness.

- **FKA1.** Let $\mathbb{E}_{A,B}(X) = E_A(E_B(X))$, where E is an IND-CPA secure symmetric encryption scheme with "elusive" and "efficiently verifiable range" as defined by Lindell and Pinkas (LP) [52]. E can be constructed from any IND-CPA secure encryption scheme SE so that $E_K(X) = SE_K(X||0^k)$. The LP construction is originally designed to find the correct entry in a garbled table to decrypt, sharing the same goal as the "point-and-permute" technique [11]. FKA1 combines the LP technique and the "point-and-permute" technique so that the evaluator only needs to try a single gate according to the point-and-permute bits. If the decrypted result does not have 0^k as the second half, the server can conclude that at least one of the two keys is ill-formed. One can instantiate SE using CBC-AES128 and incorporate the point-and-permute technique by leveraging one bit of the keys as a pointer to identify the correct entry to decrypt.
- **FKA2.** Let $\mathbb{E}_{A,B}(X) = E_A(E_B(X))$, where E is a pseudorandom injection (PRI, also known as deterministic authenticated encryption) [69]. With a PRI ciphertext, the evaluator knows if a given key is valid. PRI can be initiated using the SIV mode [69].

PG implements a *hybrid* construction of FKA1 and the garbling scheme used in TinyGarble [75] (the state-of-the-art garbling scheme implementation), where FKA1 is used to check whether the sensor inputs are well-formed and the garbling scheme in TinyGarble is used at the remaining circuit.

Caveat. One may think that FKA1 or FKA2 can be easily applied to the "first layer" gates in the circuit. However, some "first layer" gates may only get one input from the sensor, and the other input is the output of another gate. In this case, for checking N input bits, depending on the actual circuit, one may need to implement FKA1 or FKA2 on more than $\frac{N}{2}$ gates. To optimize this process, we propose decoupling the two halves in the FKA1 scheme and

Setup and inputs: Let fta be any fault-tolerant sensor fusion function. Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ be a garbling scheme. Let $\langle \text{REQ} \rangle$ be a client request that contains the function description. Let $x_\Pi = \{x_1, \cdots, x_n\}$ and $X_\Pi = \{X_1, \cdots, X_n\}$ be sensors' inputs and garbled inputs, respectively. Let X_c be the garbled inputs of filter gates provided by the client. Let X_{fc} be the garbled inputs (i.e., the output of the filter gates) for the functional circuit.

30 Client c selects a random coin r, runs Gb (using r) to generate a garbled circuit F for fta, and sends F, $\langle \text{REQ} \rangle$, encrypted coins to server S.
31 S forwards $\langle \text{REQ} \rangle$ and the corresponding encrypted coins to sensors.
32 Sensors run En (using r and x_i) and send S garbled inputs X_Π.
33 If S does not receive all the garbled inputs before the timer expires, it marks the missing sensors as malicious in a list $list_m$.
34 Server S checks the validity of the received garbled inputs using the checking gates, and if it finds that any garbled input X_i of a sensor i is ill-formed, it marks the sensor i as malicious in the list $list_m$.
35 Server S sends $list_m$ to Client c.
36 Client c run En (using r and m_{list}) and send S garbled inputs X_c.
37 Server S evaluates filter gates to obtain the garbled inputs X_{fc} for the functional circuit.
38 Server S runs Ev on X_{fc} and sends the garbled output Y to c.
39 Client c runs De (using r and Y) to get $\text{fta}(x_\Pi)$.

Figure 5: P3. The server's access to the garbled inputs of the functional circuit is controlled by filter gates. The protocol is always completed in two rounds.

introducing a special type of gate called checking gates. This means that instead of $E_{A,B}(X) = SE_{A,B}(X||0^k)$, the checking gates only encode $E_{A,B}(0^k) = SE_{A,B}(0^k)$. If the decoded output is 0^k, then we can conclude that the input labels/keys A and B are well-formed and then use these two labels to evaluate the functional circuit as usual. One checking gate can check two inputs from the same sensor, so we only need exactly $\frac{N}{2}$ additional gates for checking the validity of N garbled inputs if every sensor provides an even number of inputs. We use FKA1 instead of FKA2 because functional evaluation and false key detection can be easily decoupled.

4.4 P3: Efficient Garbling Schemes for Security against a Malicious Server

In P1 and P2, we allow the semi-honest evaluator to request valid garbled inputs from the client for the missing garbled inputs. Although the value $[-\infty, +\infty]$ encoded does not reveal any private information, this feature may be abused by a malicious server.

In the threat model TM_B, the server can also be malicious. A malicious server can potentially deceive the client by asking for another garbled input, even if it has received a well-formed one from the sensor. Once the server has both labels for a single wire, it can extract additional information by differential analysis on the underlying algorithm, or it can extract some secrets of the garbling scheme if certain optimization techniques are used. For example, from two valid labels of any wire, the server can derive the global label offset used in the free XOR optimization [47], which is one of the de-facto optimizations in all modern GC frameworks. This knowledge can potentially allow the server to recover the sensor's private input (violating the privacy guarantee) or falsify the garbled output without being detected by the client (violating the correctness guarantee). Of course, a malicious server can choose to stop responding to the client to violate the liveness guarantee, but in practice, such an attack can be easily mitigated by switching to a different server since the server does not possess any unrecoverable or secret information. Thus, we exclude the simple denial-of-service attack by a malicious server from the discussion in this paper.

To prevent a malicious server from obtaining two labels of any wire in the circuit, we introduce a layer of filter gates. A layer of filter gates is deployed in front of the functional circuit. The server can only obtain the input labels for the functional circuit through the evaluation of these filter gates, as they act as the only entry point for the server to these input labels.

Filter Gate. Each filter gate takes two input labels: one is from a sensor, and the other is from the client. To use the filter gate, the sensors still garble their own inputs as usual. Depending on the result of the checking gates and the time-out timer, the server knows which sensors' garbled inputs are missing or ill-formed. Then, the server sends the list of malicious sensors to the client. Based on the list provided by the server, the client replies with a label of the client's input on each of the filter gates. For the sensors that have provided well-formed garbled inputs, the client replies with a label corresponding to value 1, and for the other sensors, the client replies with a label corresponding to value 0. The server subsequently evaluates filter gates using garbled inputs it received from the client and/or the corresponding sensor. The outputs of the layer of filter gates are the input labels for the client's designated functional circuit. Then, the rest of the protocol is carried out as usual. The complete protocol of $P3$ is in Fig. 5.

We introduce a special garbling scheme for the filter gates. There are only three, instead of four, entries in the garbled table for a filter gate. Two entries are $E_{X_S^b, X_C^1}(X_O^b) = SE_{X_S^b, X_C^1}(X_O^b)$, where X_S, X_C, X_O are the labels from the sensor, from the client, and gate output, the superscript b can be 0 or 1, representing the truth value of the label. Another entry of the table is $E_{X_C^0}(X_O^\infty) = SE_{X_C^0}(X_O^\infty)$, where X_O^∞ is the label corresponding to the infinity value of the sensor. Note that the first two entries simply pass the value represented by X_S^b to X_O^b when the client gives label X_C^1, representing 1. The last entry only takes one input from the client, so it allows the server to recover the embedded X_O^∞ label when no well-formed garbled input from the sensor is available. Effectively, the filter gate implicitly transfers a label that corresponds to $-\infty$ or $+\infty$ to the server when a well-formed label is missing.

Filter gates ensure that a malicious server can only obtain at most one valid garbled label of each wire (input or output) of a filter gate, no matter how the server manipulates the list of malicious sensors. If the server claims a sensor input is missing, it will only get X_C^0 from the client and subsequently unlock X_O^∞ at the output. If the server claims a sensor input is well-formed, it can, at best, only get the corresponding X_O^b as output because if the server actually does not have a sensor input, it will not be able to evaluate the circuit further and be detected by the client as a malicious server.

Remark. When a malicious server colludes with malicious sensors (i.e., obtain two different sensor labels X_S on the same wire or get to know the random coin r), it can derive two different valid labels X_O for the corresponding gate output wires by evaluating a filter gate. It can then XOR these two X_O labels on the same wire to derive the circuit-global R (the global offset used in the FreeXOR optimization). This allows the server to derive all possible labels in the whole circuit and tamper with the final output without being detected by the client, as well as perform differential analysis on the (intermediate) computation result to reveal extra information about the honest sensors' inputs. More garbling-scheme-specific

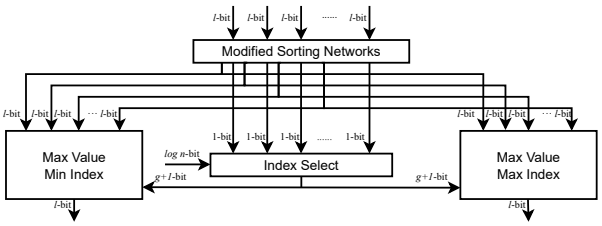

Figure 6: The complete circuit design for M-g-U. n is the number of sensors, l is the length of the endpoint of the input interval, and g is the number of faulty sensors. There are $2n$ l-bit inputs for the circuit.

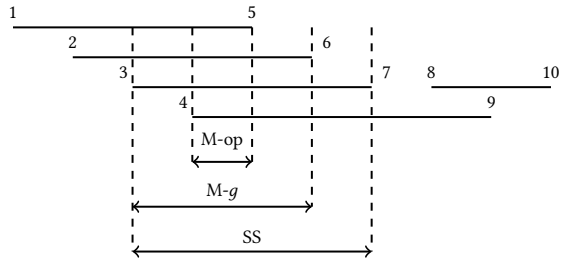

Figure 7: A five-sensor system with M-g, M-op and SS. The sensor input intervals are [1, 5], [2, 6], [3, 7], [4, 9], [8, 10]. The resulting intervals are [3, 6], [4, 5], and [3, 7] for M-g, M-op, and SS, respectively.

attacks are possible when the attacker knows both labels on the same wire [16]. To prevent this, we will have to run oblivious transfer(OT) [8, 53] between the client and the sensors to prevent the server and the sensors from knowing more labels than they need.

5 ALGORITHMIC CIRCUIT DESIGN

5.1 Variants of Fault-Tolerant Averaging Algorithms

We briefly survey the key fault-tolerant sensor fusion/averaging algorithms [25, 56–58, 72]. Before the protocol starts, all participants should agree on a representation for the intervals. For intervals, each possible value is given an integer label with l bits, and we assume the lower and upper bounds are 0 and σ, respectively. Therefore, $l = \log \sigma$.

M-g-U. Our complete circuit design for M-g-U is depicted in Fig. 6. To implement Marzullo's algorithm, we first need to sort all the input endpoints of sensor inputs, resulting in a sorted array of $2n$ values (considering each sensor is providing one interval in the form of two endpoints). This is achieved using *modified sorting networks*. For each point in this sorted array, we need to count how many input intervals can cover this point. This is handled by adding 1 to an intersecting interval counter if the point is a left end of an input interval and subtracting 1 if it is a right end. After that, we compare the intersecting interval counters for each point with $n - g$, in order to find the points that are covered by *exact* $n - g$ intervals. We do this using *index select*. As in Fig. 6, the left end of the resulting interval is the output of the *max value min index* module.

Likewise, the circuit for computing the right end of the resulting interval can be implemented in a symmetric way by again running the modified sorting networks and index select; however, we will show in Section 5.2 that we can *reuse* the modified sorting networks and the index select module for computing the right endpoint.

M-g. M-g-U applies to the case of arbitrary failures with unbounded inaccuracy. The same algorithm, in fact, applies to the case of arbitrary failures with bounded inaccuracy, where the maximum length of the interval is known and values that are too inaccurate can be detected [56]. In this case, the algorithm (M-g) needs $2g + 1$ sensors to tolerate g Byzantine failures with bounded inaccuracies.

M-g-m. It is not uncommon to require the averaging algorithm to only return the midpoint of the interval [58]. This may even be more desirable in a privacy-preserving setting, as providing the lo and hi values might reveal too much unnecessary information. We write M-g-m to denote this variant calculating the midpoint of M-g.

M-op. Marzullo [58] also gave a solution to the case where the system parameter g is unknown or unspecified. The goal is to find the cover for the *maximum* intersection groups for all the intervals. Thus, the algorithm is "optimistic", and we write M-op to denote it.

SS. Marzullo's algorithms may exhibit a somewhat irregular behavior: it is possible that when Marzullo's algorithms are applied to two slightly different input sets, the output may be quite different. This is formalized as violation of the *Lipschitz condition* regarding a certain metric [51]. Schmid and Schossmaier [72] offered a solution, SS, which can satisfy the Lipschitz condition. The algorithm is simple: Given n intervals $I_i = [u_i, v_i]$ $(1 \le i \le n)$, (at most) g of which may be faulty, SS simply outputs $[\max^{g+1}\{u_1, \cdots, u_n\}, \min^{g+1}\{v_1, \cdots, v_n\}]$, where $\max^{g+1}\{u_1, \cdots, u_n\}$ denotes the element $u_{j_{g+1}}$ in the ordering $u_{j_1}, \cdots, u_{j_n}$ of $\{u_1, \cdots, u_n\}$ from largest to smallest, and $\min^{g+1}\{v_1, \cdots, v_n\}$ denotes the element $v_{j_{g+1}}$ in the ordering $v_{j_1}, \cdots, v_{j_n}$ of $\{v_1, \cdots, v_n\}$ from smallest to largest. While SS shares the same worst-case performance as Marzullo's, SS may generate a larger output interval.

Example. To help understand the algorithms described, we visualize an example in Fig. 7 which shows how three algorithms (M-g, M-op, and SS) work. All of the three algorithms deal with bounded accuracy and use $2g + 1$ sensors to tolerate g faults.

As in Fig. 7, the input intervals for the sensors are [1, 5], [2, 6], [3, 7], [4, 9] and [8, 10]. For all the algorithms, all input endpoints need to be sorted. To find the left endpoint of the resulting interval for M-g, we can imagine that there is a vertical line sweeping from left to right. The vertical line can stop at the leftmost point that intersects $n - g = 3$ intervals. In the example, this point is 3. Similarly, to find the right endpoint, a vertical line can sweep from right to left and find the right end (6). Thus, the resulting interval is [3, 6].

Instead of outputting an interval, M-g-m will output the midpoint of the resulting interval generated by M-g.

In contrast to M-g, the M-op algorithm does not need to know the g value beforehand. A vertical line will sweep over all the endpoints and find the leftmost and rightmost points that intersect with the maximum input intervals. In the example, points 4 and 5 are covered

Table 1: PG characteristics. n is the total number of sensors, g is the upper bound on the number of malicious sensors, k is the security parameter (in this paper, 128-bit), and l is the length of the sensor input. The column labeled "complexity" specifies the time complexity of the algorithm in the plain setting. The columns labeled "server circuit," "sensor time," "sensor communication," and "#rounds" specify for server circuit complexity, sensor time complexity (measured using the number of pseudorandom function calls), sensor communication complexity, and the round complexity for PG, respectively.

algorithms	#sensors	description	complexity	server circuit	sensor time	sensor communication	#rounds
M-g-U	$3g+1$	unbounded accuracy	$O(n\log(n))$	$O(ln\log^2(n))$	$O(l)$	$O(lk)$	1 or 2
M-g	$2g+1$	bounded accuracy	$O(n\log(n))$	$O(ln\log^2(n))$	$O(l)$	$O(lk)$	1 or 2
M-g-m	$2g+1$	only reveal midpoint	$O(n\log(n))$	$O(ln\log^2(n))$	$O(l)$	$O(lk)$	1 or 2
M-op	$2g+1$	"optimistic"	$O(n\log(n))$	$O(ln\log^2(n))$	$O(l)$	$O(lk)$	1 or 2
SS	$2g+1$	Lipschitz condition	$O(n\log(n))$	$O(ln\log^2(n))$	$O(l)$	$O(lk)$	1 or 2

by four input intervals, while the rest endpoints are covered by at most three input intervals. Thus, M-op will output [4, 5] as the result.

For SS, one needs to find the $(n-g)$-th smallest left endpoint and the $(n-g)$-th largest right endpoint. In the example, point 3 and point 7 are picked as they are the third smallest left end and the third largest right end, respectively.

The example would be easily extended to explain M-g-U with unbounded accuracy. However, it requires at least $3g+1$ sensors to tolerate g failures.

5.2 Our Circuit Design in Detail

Instead of using $(+1, u)$ and $(-1, v)$ to represent an interval $[u, v]$ (as in the code version of Marzullo's algorithm), in our circuit design, each sensor is just required to provide an interval in the form of two values u, v to the server, without specifying which one is larger. This is because sensors may be malicious, and it will result in a wrong result if the left end provided by a malicious sensor is actually larger than the right end. Therefore, at the first level of our modified sorting network, we need to add an array of compare-and-swap modules to sort the two values from the same sensor. Note that the above problem is not what the conventional garbled circuit design would care about.

Before we proceed, let's recall sorting networks [9, 27, 59], which are circuits that sort a sequence into a monotonically increasing sequence. The core building block of a sorting network is a compare-exchange circuit, which takes as input a pair of values (x, y) and returns a sorted pair (x', y') so that $x' = min(x, y)$ and $y' = max(x, y)$. To realize sorting networks, prior constructions [39, 46, 47] used the idea of *compare then conditional-swap:* the circuit keeps two inputs unchanged if and only if the comparator returns 1, i.e., x is less than y. For our design, the first layer of our modified sorting network guarantees that the two values from the same sensor will form an interval with its right end always greater or equal to its left end. Then we taint these two values with +1 and −1 respectively to indicate the order of two endpoints. In our implementation, we use one bit to represent ±1.

From the second layer of our modified sorting networks, it is essentially sorting networks built from compare-exchange components with a modified comparator, as illustrated in Fig. 8. Instead of using the *less-than* comparator, we follow Marzullo's algorithm [56] to realize a comparator, $<_m$, as defined below: Given two inputs x and y, each of which is of the form $s||u$ where $s \in \{1, 0\}$

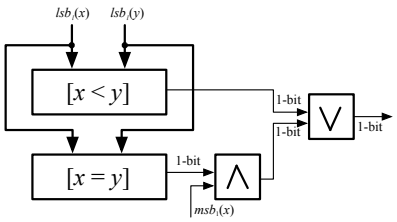

Figure 8: The comparator component.

and $|u| = l$, define $x <_m y = (lsb_l(x) < lsb_l(y)) \vee (lsb_l(x) = lsb_l(y) \wedge msb_1(x) > msb_1(y))$, where lsb_l and msb_l represent the least significant l bits and the most significant l bits, respectively. In other words, $x <_m y$ if and only if the value part of x is less than that of y, or the value parts happen to be equal and the sign part of x is greater than that of y. Note that the sign part of a left end is encoded by 1.

We follow [46] to realize the conventional less-than circuit and equal-to circuit (which takes advantage of the free-xor technique). Observing that when $lsb_l(x) = lsb_l(y)$ and $msb_1(x) = 1$, it does not matter whether we swap or not the two inputs. So we can further simplify the circuit, leading to a circuit exactly as in Fig. 8.

While an asymptotically optimal sorting network exists [3], PG uses Batcher sorting network [9, 59] which has much better performance for practical parameters, as studied in [54].

Now, we describe how to find the position of the minimum value of the resulting interval by our index select module composed of a prefix sum circuit and an array of equality checkers, as shown in Fig. 9. All the sorted one-bit inputs, representing +1 or −1, first go through a prefix sum circuit to compute their prefix sums. Prefix sum circuit allows one to compute on input $(z_1, z_2, \cdots, z_n)$ and produce as output $(m_1, m_2, \cdots, m_n)$, where $m_j = z_1 + z_2 + \cdots + z_j$ for $1 \le j \le n$. Indeed, this circuit fits perfectly for our purpose as the intersecting interval counter in M-g-U. A straightforward instantiation of n-prefix sum circuit requires n additions.[4]

The next layer is to convert every prefix sum which is equal to the value $n-g$ to 1 and convert the rest to 0 otherwise. This can be trivially done by a simple equal-to circuit [46]. Observing that not every position in the array of prefix sum can possibly equal $n-g$, we can apply another optimization that only implements comparators

[4]In a system that can evaluate garbled circuits in parallel [62], we recommend implementing a parallel prefix sum circuit as mentioned in (cf. [71, Chapter 2.6]), which has a depth of $O(\log n)$ and $O(n)$ additions.

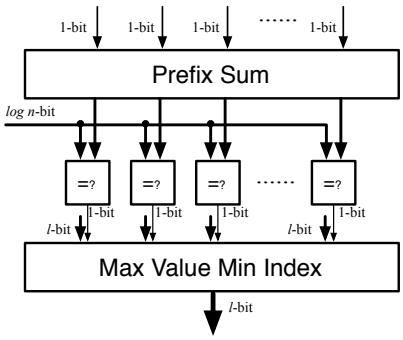

Figure 9: The index select module and the max value min index module. The index select module is composed by a prefix sum circuit and an array of equality checkers.

for the positions where $n - g$ can be the possible output. To be precise, for an array with $2n$ values provided by n sensors, only $g + 1$ positions can possibly have a prefix sum equal to $n - g$. This observation reduces the number of comparators and the width of the max value min index by roughly 83.3% of that with a straightforward implementation because it only compares at $g + 1$ positions instead of $2n = 6g + 2$ positions in a straightforward implementation. The size reduction for the other algorithms, however, is slightly smaller, with a 75% reduction, since $n = 2g + 1$ for the other algorithms.

Then these $g+1$ values go through a max value min index circuit which computes the value with the minimum index and the maximum value (which is 1). Effectively, it outputs the leftmost point, which covers $n - g$ sensor input intervals. We can easily modify the circuit from [46] to obtain this circuit.

To compute the maximum value, we find that one can reuse the result of modified sorting networks and index select modules. Specifically, we just need to left shift one position of the sorted input value array (shifting is free in circuits), and apply it to a max value max index circuit. Then, we will generate the right end of the result interval. Since the whole circuit complexity is dominated by the sorting network, this optimization avoids another sorting network and index select circuit for computing the right end of the resulting interval, thereby halving the overall computation overhead.

PG for M-g-U. With the above circuit, we can have the following system tolerating Byzantine sensors. Specifically, if a server receives no replies or ill-formed sensor inputs, the server will ask the client to send corresponding garbled inputs for the faulty or missing sensors explicitly via communication or implicitly via filter gates. We have the following theorem establishing the security of the above scheme in the synchronous environment.

Theorem 5.1. *The PG (P2 and P3) protocol achieves Privacy, Correctness, and Liveness for M-g-U in TM_A and TM_B, respectively.*

6 IMPLEMENTATION AND RESULTS

We evaluate PG[5] in two different environments: one is on the AWS cloud with up to 261 sensor nodes, which shows the scalability of PG, and the other is a local cyber-physical system setting with up to 19 sensor nodes, which shows the performance of PG in resource-constrained devices.

6.1 GC Compiler

While multiple generic GC compilers or tools that can translate a program to a circuit exist [15, 40, 41, 48, 54, 75, 80], there is significant room for improvement for some specific programs. Our circuit optimization requires non-trivial efforts and analysis for the correctness of the design.

Among all the existing GC compilers/frameworks, TinyGarble [75] is generally deemed to be (one of) the most efficient one, especially for large programs, as it incorporates state-of-the-art optimizations such as free-XOR [47], row reduction [65], fixed-key AES garbling [15], and half gates [81], and more importantly, uses logic synthesis to reduce the size of circuits.

We make a general garbled circuit compiler specifically for our clients-server-sensors setting by modifying and extending TinyGarble [75], a two-party garbled circuit compiler. Specifically, we first decompose TinyGarble into three parties: clients (garbled circuit generators), the server (garbled circuit evaluator), and sensors (garbled input providers). We then modify the system to be coin-based: clients and sensors now take as input shared random coins. Last, we modify the garbled circuit evaluation function so that garbled output is hidden from the server and only decodable by the client, which is essential for achieving privacy.

As TinyGarble does not support efficient sorting networks or other primitives we need, we directly build optimized circuits described in Sec. 5 in Verilog. The resulting circuits go through another logic synthesis process by Synopsys Design Compiler to obtain the netlists of the implemented algorithms. Lastly, we applied the V2SCD tool in TinyGarble to convert netlists into simple circuit description files, which can be taken as the inputs of the TinyGarble framework. We choose not to build PG based on the latest TinyGarble2 [41] just because the new features in TinyGarble2 are not needed in our implementation, i.e., we directly implement algorithmic circuits in Verilog, instead of C++.

6.2 Cloud Implementation and Deployment

To show the practicality of our system, we deploy and evaluate PG (P1, P2, P3) on AWS EC2 instances (t2.micro, 8 GiB, 1vCPU, 1 GiB memory). We create an Amazon Virtual Private Cloud (Amazon VPC) with two subnets (i.e., a public subnet and a private subnet). Our PG is deployed in the isolated private subnet where the EC2 instances can only communicate with AWS EC2 instances and cannot access the Internet or receive external access. In this way, we get rid of unnecessary interference from the Internet. Each party (i.e., a server, a client, or a sensor) is simulated by an EC2 instance on the private subnet. Notice that each sensor is simulated independently by one EC2 instance.

All sensor values have a fixed length of eight bits. To garble an interval, each sensor is required to process a 16-bit input, resulting in a 256-byte garbled input. The computation carried out on each sensor involves only a small number of pseudo-random

[5]Code and evaluation results available:
https://doi.org/10.6084/m9.figshare.25669026.v2

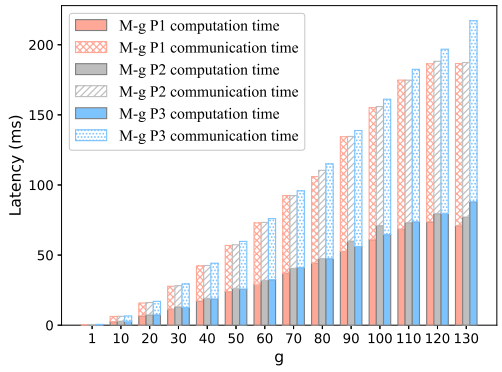

Figure 10: Latency (in ms) of three versions of PG for $g = 1$ to 130 when using M-g.

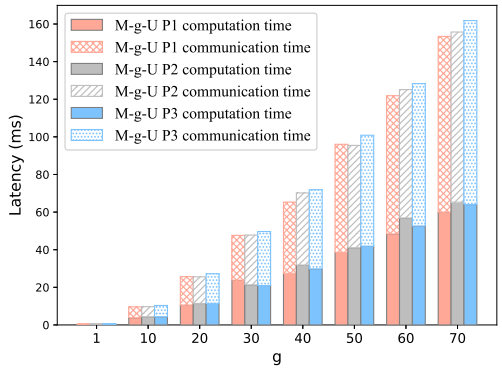

Figure 11: Latency (in ms) of three versions of PG for $g = 1$ to 70 when using M-g-U.

function evaluations and XOR operations. This makes the computation a highly efficient process that can be executed on resource-constrained sensors with minimal memory and processing power, as we will demonstrate in Sec.6.3.

We evaluate PG with five different algorithms: M-g (Fig. 10), M-g-m, M-g-U (Fig. 11), M-op (Fig. 12), SS under different network sizes (the number of sensors) with up to 261 sensors. More specifically, we have tested M-g and SS with up to 261 ($g = 130$) sensors, Mop with up to 241 ($g = 120$) sensors, M-g-m with up to 221 ($g = 110$) sensors, and M-g-U with up to 211 ($g = 70$) sensors. A more detailed comparison between these algorithms can be found in Table 1.

Latency. The latency for M-g, M-g-U, M-op with different g values are presented in Figs. 10 to 12. To better understand the performance of PG, we measure the time consumed by computation and communication separately. The computation time only contains the circuit evaluation time on the server's side and the circuit garbling time on the client's side. We omit the input garbling time on the sensors since it is negligible compared with the computation on the server or the client. We can see in Figs. 10 to 12 that both the computation time and the communication time increase gradually with the number of sensors as expected, and the computation time is always about 40% of the whole latency. For example, when running P3, the computation time and the communication time of M-g algorithm grow from 0.163ms to 87ms and from 0.404ms to 129ms, respectively, when the number of sensors n increases from $n = 3$

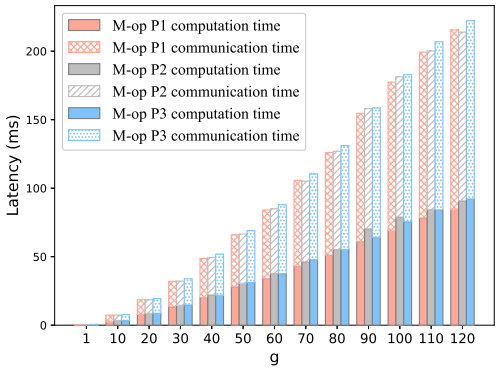

Figure 12: Latency (in ms) of three versions of PG for $g = 1$ to 120 when using M-op.

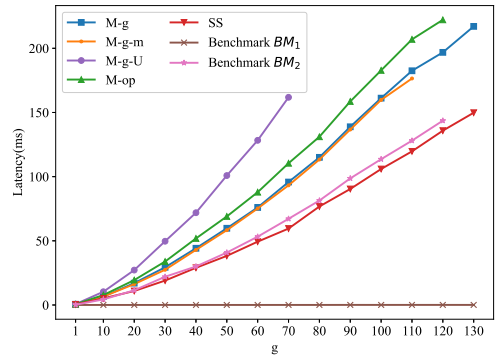

Figure 13: Latency (in ms) of P3. Two benchmarks are also shown in this figure.

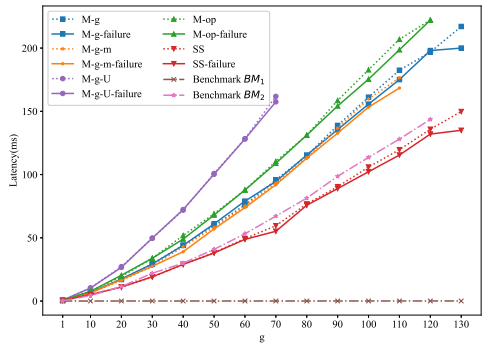

Figure 14: Latency (in ms) of P3, including both failure and failure-free scenarios.

to $n = 261$. This shows the majority of the latency is consumed by communication, so if we can reduce the communication latency, and we would be able to boost the performance of PG.

Bandwidth. We use nethogs to measure the bandwidth of each sensor. In PG, the only communication between the sensor and the server happens when the sensor submits its labels. The size of sensor input labels is independent of the version and, thus, should always be the same. When the length of the sensor input size $l = 16$ bits, the theoretical communication between a sensor and the server is 2048 bits. In our experiments, this communication is measured to be

4880 bits, where the extra bits come from packet headers, encoding, encryption IV, etc. The total communication from n sensors to the server is thus $4880n$ bits.

Scalability. We evaluate the scalability of PG by using up to 261 sensors. The largest latency we obtained in our evaluation is the latency of M-op in P3 when $g = 120$, which is only 222ms, as shown in Fig. 12. This demonstrates that all three versions of PG are efficient for all algorithms and network sizes we tested.

From Fig. 13, we see that for all the evaluated versions and algorithms, the computation time and communication time increase gradually as the number of sensors increases. Overall, the results match the asymptotic analysis of the circuit complexity: $O(ln\log^2(n))$, where l is the input length from each sensor, and n is the number of sensors. Both the computation time and communication time are dominated by the size of the circuit, as all the gates in a garbled circuit need to be garbled, evaluated, and transmitted one by one.

The scalability of PG means the ability to support a large number of sensors (inputs) or a large input size. This scalability metric was also used in [40, 63, 77]. However, there are several other interpretations of scalability in the field of secure multiparty computing (MPC). For example, [19] aims to compile and optimize large programs in a scalable way; [12] aims to reduce communication costs when the number of participants increases.

Performance in failure scenarios. Since our system can achieve liveness, we also evaluate PG when g sensors have crashed and are not responsive. Note that we only report the performance of PG when g sensors crash, but not when g sensors are sending ill-formed garbled inputs or a mix of these two scenarios. This is because the performance of PG (P2 or P3) is almost identical, without counting the time-out timer, in all failure scenarios when g sensors are Byzantine, and P2 and P3 treat missing sensor inputs and ill-formed sensor inputs in the exactly same way. The latency of PG in failure, without counting the time-out timer, and failure-free scenarios is compared in Fig. 14. Interestingly, we notice that almost in all of the cases, PG runs faster in the failure scenario than in the failure-free scenario. This is because the server does not need to communicate with the g crashed sensors anymore.

Benchmarks. In addition to our system, we implement two benchmarks BM_1 and BM_2 for comparison. BM_1 directly executes the M-g algorithm in plaintext, so it achieves the liveness and correctness (fault-tolerance) of our protocol but not privacy. BM_2 executes a garbled circuit that implements an algorithm for finding the median of the inputs provided by n sensors, so it achieves privacy but not liveness and correctness. The latency of these two benchmarks with different numbers of sensors are presented in Fig. 13 and Fig. 14. We can clearly see that BM_1 runs very fast as no cryptographic operation is involved. BM_2 exhibits comparable performance to the SS algorithm because the circuit sizes of these two algorithms are roughly the same.

6.3 Case study: CPS Implementation and Deployment

In addition to the large-scale cloud evaluation, where we can scale up to 261 sensors, we also deploy PG locally in a cyber-physical system setting and conduct a small-scale case study to show the practicality of deploying PG in a real sensor network. We build a sensor fusion system using up to 19 Raspberry Pi Zero W (1GHz, single core CPU and 512MB RAM) as sensors. Raspberry Pi Zero W is the cheapest Raspberry Pi device supporting WiFi connection and is smaller than a credit card. Notice that we selected Raspberry Pi Zero W just because of its built-in WiFi module, not its computational power. The server runs on an Intel Core i7-4790 processor, and the client runs on another computer with an Intel Core i7-4702HQ processor. Sensors and the server are connected using one wireless router. To support concurrent transmissions of sensor inputs, we implement multi-threading for data collection on the server side. The client and the server are physically connected via an Ethernet cable. The length of all sensor values is still the same as the cloud evaluation, i.e., $l = 16$.

The system deployed as a case study is just P1 because there is not much difference in the performance of P1, P2, and P3, according to the large-scale cloud-based evaluation above. We evaluate PG using up to 19 sensors, with the same five algorithms: M-g, M-g-m, M-g-U, M-op, SS. We measure the latency in both failure and failure-free scenarios. For failure-free scenarios, each sensor will provide a well-formed garbled input, even if the underlying value is faulty. In contrast, our failure scenario captures crash failures where some sensors do not provide the server with garbled inputs in time.

Latency. The latency evaluation for the five algorithms in the failure-free scenario is depicted in Fig. 15 for $g = 1$ to 9. We find that in the CPS scenario, the communication between the server and sensors is two orders of magnitude larger than the cryptographic operations and the rest of the communication. Combining the previous large-scale evaluation results, we can see that when g is small, the communication is a more dominant factor in the system, and when g is large, the time consumed by garbling, transmitting, and evaluating the circuit will be more noticeable.

When comparing the performance of different algorithms, we notice that for a given g, M-g, M-g-m, M-op, and SS have roughly the same latency. This is because the latency is dominated by the communication time between the server and sensors, and the total size of garbled inputs transmitted for these algorithms is exactly the same. If we compare the latency between M-g-U and the other algorithms for the case where the total number of sensors are equal, e.g., $g = 6$ for M-g-U and $g = 9$ for M-g-m, the latency difference is almost not noticeable. This confirms that in the CPS setting, the latency is dominated by the communication, not the cryptographic operations.

Performance in failure scenarios. In failure scenarios, we test the case where g out of n sensors fail at the same time. The latency of PG for $g = 1$ to 9 in failure scenarios are also shown in Fig. 15. Like the cloud evaluation, we do not count the time-out timer in the reported latency. In the CPS setting, we also notice that PG is faster in the failure scenario than in the failure-free scenario. Although what we deploy is P1, which needs one extra round of communication between the client and the server if any sensor crashes, the server still saves communication time with g sensors, and apparently, communicating with g sensors over WiFi would

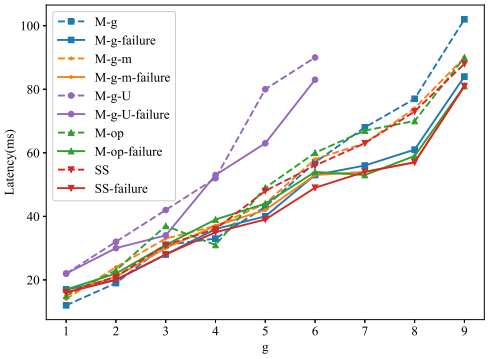

Figure 15: Latency of P1 in failure and failure-free scenarios in the CPS deployment.

take longer than the one additional round of communication with the client over an Ethernet cable.

Overall, PG has shown promising performance in practical scenarios and may be deployed in some real-time applications, e.g., water distribution system monitoring pressure and water leaks [4].

7 RELATED WORK

Comparison with privacy-preserving sensor fusion and aggregation. Most privacy-preserving data aggregation systems only protect individual sensor inputs but fail to handle malicious sensors that attempt to sway the aggregated value [22, 23, 35, 42, 43, 49]. Only a handful provide a partial solution by leveraging a cryptographic proof that sensor input has a specific property (say, is within a predetermined range) [28, 31, 73]. PG takes a fundamentally different approach by *tolerating* malicious sensor inputs without asking for input validity proofs. Other works (e.g., [23, 42]) provide privacy-preserving aggregation with tolerance to benign (crash) sensor failures, i.e., where some sensors fail to provide their inputs. Instead, our system deals with Byzantine failures, where corrupted sensors can provide the server with arbitrary values.

A number of privacy systems [34, 42, 60, 73] additionally provide differential privacy. These systems still do not formally defend against pollution attacks.

SIA [67] explored a setting where individual sensor inputs do not need to be privacy-protected but the central server needs to *verifiably* provide clients with correct values. This helps achieve integrity. In contrast, PG achieves both privacy and integrity.

Pollution attacks and Sybil attacks. Pollution attacks have also been studied in other areas such as network coding [2] and personalized services [78]. We work with a fundamentally different setting, focusing on data and sensor pollution attacks. Our pollution attack scenarios are also different from those of Sybil attacks [30], where an adversary may forge multiple or even an unlimited number of identities to damage distributed systems. While Sybil attacks and pollution attacks may share somewhat the same goal, Sybil defenses [5] offer no help for defending against data pollution.

Liveness in multi-party computation. Achieving liveness *efficiently* has been a difficult problem for garbled circuit-based multi-party computation. For instance, schemes in [44, 45] do not achieve liveness. Some prior work on GC [13, 26] achieves liveness by introducing multiple servers and using secret-sharing-based techniques to help liveness. This is not only less efficient in general but also requires significant communication and interaction, which makes it ill-suited in bandwidth and energy-restricted environments.

Comparison with differentially-private systems. Apple [6] and Google [36] use differential privacy [32, 33] mechanisms to compute aggregate statistics. In these systems, each sensor adds random noise, and the noisy data will be aggregated to get an estimation of the aggregated values. These systems allow us to achieve robustness but have to trade between accuracy and privacy. Increasing the noise level reduces information leakage for individuals, but also reduces the estimated accuracy.

Essentially, the goals between differentially-private systems and our system are orthogonal. In a differentially-private system, the server will know a noisy version of the value measured by each sensor, and thus learn much more than that in our system. However, the client learns slightly less than our system because of the noise added.

Fault-tolerance and garbled circuits. Nielsen and Orlandi [66] built LEGO for two-party computation in the malicious case. In LEGO, the garbler first sends many gates, and the receiver tests if they are constructed correctly by opening some of them. Then, the parties run interactively to solder the gates (as Lego blocks) into a circuit. They use a fault-tolerant circuit to ensure a valid output from a majority of good ones. In contrast, our system exploits the fault-tolerant features of the underlying algorithms to achieve a garbled circuit-based, privacy-preserving system that can tolerate pollution attacks, returning correct results even in the presence of Byzantine failures and malicious attacks.

Non-colluding multi-party computation. The assumption that a number of parties do not collude is not only used to build theoretical multi-party computation [10, 18, 44, 61], but used in practical multi-party computation systems [20, 21, 28, 31, 45, 64]. Among these systems, many use garbled circuits as a building block (e.g., [20, 21, 44, 45, 64]). Our notion of non-collusion follows this line of research but has an architecture that is different from all these existing systems. In our setting, we assume that clients only have the means to contact the server, and we assume the server and sensors do not collude. Note that non-collusion of parties does not imply that the parties are trusted. Rather, it simply means these parties do not work together (i.e., share internal states). A few systems [20, 21, 64] relying on the non-colluding assumption work in the three-party computation only, using modern mobile devices. They attempt to resolve different problems from ours.

Efficient GC implementations. Starting from Fairplay [55], a large number of GC tools (compilers or implementations) have been proposed [15, 29, 40, 48, 54, 75, 80]. Our system is based on (but makes significant modifications to) TinyGarble [75], an approach that in addition to using state-of-the-art optimizations such as free-XOR [47], row reduction [65], fixed-key AES garbling [15], and half gates [81], leverages logic synthesis to reduce the size of circuits.

Our PG achieves false-key awareness in the garbling scheme of P2. To the best of our knowledge, all existing GC frameworks do not have this feature implemented.

The difference among P0, FKN, KMR, and NPS. Our basic scheme, P0, shares some similarities with both Feige, Kilian, and

Naor (FKN) [37] and Kamara, Mohassel, and Raykova (KMR) [44]. The difference is that FKN and KMR only involve a server and parties (in our case, sensors) and the server needs to send back the garbled output to the parties, while in our model, the server needs to return the garbled output to the client and *optionally* to the sensors. In FKN and KMR, the server and one party do heavy work that is linear in the size of the circuit, while in our case, each sensor's role is symmetric, and each sensor only does work that is linear in the size of its input. The security of KMR requires only obliviousness and authenticity of the garbling scheme, while PG additionally requires privacy of the garbling scheme.

P0 is also similar to Naor, Pinkas, and Sumner (NPS) [65], one designed specifically for auctions. In NPS, there is an auction issuer who generates the circuit, a number of bidders who send their garbled values, and an auctioneer who computes the garbled values and returns the final result to all bidders. Instead of relying on an external, trusted circuit issuer, our circuit generator is just one participating client (who would also expect a reply from the server). Moreover, NPS uses proxy oblivious transfer to provide the parties with the garbled input, but we choose to use an agreed common coin, just as FKN and KMR, for the purpose of efficiency and scalability.

The servers in both FKN and NPS can learn the output, while KMR and P0 do not.

8 CONCLUSION

We design and implement PG—a Byzantine fault-tolerant and privacy-preserving multi-sensor fusion system by developing techniques from dependable distributed systems and modern cryptography. Besides protecting privacy, PG can defend against pollution attacks and provide guaranteed output delivery. Via a deployment on a cloud-based system and a cyber-physical system, we show that PG is efficient in both failure-free and failure scenarios.

ACKNOWLEDGMENT

This work was supported by National Key R&D Program of China under 2022YFB2701500. Chao Yin was partially supported by the China Scholarship Council and the Dutch Sectorplan. Marten was partially supported by NSF grant CNS-1413996 for MACS: A Modular Approach to Cloud Security. We would also like to thank Tara John, Syed Kamran Haider, and Hamza Omar for their assistance with the communication protocol in the very first prototype of the system. Additionally, we extend our gratitude to Raihan Sayeed Khan and Sirui Shen for their help with the Synopsys Design Compiler.

REFERENCES

[1] Soroush Abbasian Dehkordi, Kamran Farajzadeh, Javad Rezazadeh, Reza Farahbakhsh, Kumbesan Sandrasegaran, and Masih Abbasian Dehkordi. 2020. A survey on data aggregation techniques in IoT sensor networks. *Wireless Networks* 26 (2020), 1243–1263.

[2] Shweta Agrawal, Dan Boneh, Xavier Boyen, and David Mandell Freeman. 2010. Preventing pollution attacks in multi-source network coding. In *Public Key Cryptography–PKC 2010: 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings 13*. Springer, 161–176.

[3] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An O (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 1–9.

[4] Michael Allen, Ami Prels, Mudasser Lqbal, Seshan Srirangarajan, Hock Beng Llm, Lewis Glrod, and Andrew J Whittle. 2011. Real-time in-network distribution system monitoring to improve operational efficiency. *Journal-American Water Works Association* 103, 7 (2011), 63–75.

[5] Lorenzo Alvisi, Allen Clement, Alessandro Epasto, Silvio Lattanzi, and Alessandro Panconesi. 2013. Sok: The evolution of sybil defense via social networks. In *2013 ieee symposium on security and privacy*. IEEE, 382–396.

[6] Apple. [n. d.]. Differential privacy overview.

[7] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. 2014. How to garble arithmetic circuits. *SIAM J. Comput.* 43, 2 (2014), 905–929.

[8] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 535–548.

[9] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.

[10] D Beaver, S Micali, and P Rogaway. [n. d.]. The round complexity of secure protocols extended abstract. In *22nd ACM STOC*.

[11] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 503–513.

[12] Gabrielle Beck, Aarushi Goel, Aditya Hegde, Abhishek Jain, Zhengzhong Jin, and Gabriel Kaptchuk. 2023. Scalable multiparty garbling. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2158–2172.

[13] James Bell, Adria Gascon, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. 2022. Distributed, private, sparse histograms in the two-server model. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 307–321.

[14] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. 2023. {ACORN}: Input Validation for Secure Aggregation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4805–4822.

[15] 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*. IEEE, 478–492.

[16] 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*. IEEE, 478–492.

[17] 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.

[18] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 2019. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*. 351–371.

[19] Niklas Büscher and Stefan Katzenbeisser. 2017. *Compilation for secure multi-party computation*. Springer.

[20] 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*. 266–275.

[21] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. 2016. Secure outsourced garbled circuit evaluation for mobile devices. *Journal of Computer Security* 24, 2 (2016), 137–180.

[22] Claude Castelluccia, Aldar C.-F. Chan, Einar Mykletun, and Gene Tsudik. 2009. Efficient and provably secure aggregation of encrypted data in wireless sensor networks. *ACM Trans. Sens. Networks* 5, 3 (2009), 20:1–20:36.

[23] T H Hubert Chan, Elaine Shi, and Dawn Song. 2012. Privacy-preserving stream aggregation with fault tolerance. In *Financial Cryptography and Data Security: 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers 16*. Springer, 200–214.

[24] Wei-Ning Chen, Christopher A Choquette Choo, Peter Kairouz, and Ananda Theertha Suresh. 2022. The fundamental price of secure aggregation in differentially private federated learning. In *International Conference on Machine Learning*. PMLR, 3056–3089.

[25] Paul Chew and Keith Marzullo. 1990. *Masking failures of multidimensional sensors*. Technical Report.

[26] Michele Ciampi, Vipul Goyal, and Rafail Ostrovsky. 2021. Threshold garbled circuits and ad hoc secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 64–93.

[27] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[28] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *NSDI*. USENIX Association, 259–282.

[29] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.

[30] John R Douceur. 2002. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 251–260.

[31] Yitao Duan, John Canny, and Justin Zhan. 2010. {P4P}: Practical {Large-Scale}{Privacy-Preserving} Distributed Computation Robust against Malicious Users. In *19th USENIX Security Symposium (USENIX Security 10)*.

[32] Cynthia Dwork. 2006. Differential privacy. In *International colloquium on automata, languages, and programming*. Springer, 1–12.

[33] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our data, ourselves: Privacy via distributed noise generation. In *Advances in Cryptology-EUROCRYPT 2006: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28-June 1, 2006. Proceedings 25*. Springer, 486–503.

[34] Tariq Elahi, George Danezis, and Ian Goldberg. 2014. Privex: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1068–1079.

[35] Zekeriya Erkin and Gene Tsudik. 2012. Private computation of spatial and temporal power consumption with smart meters. In *Applied Cryptography and Network Security: 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings 10*. Springer, 561–577.

[36] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1054–1067.

[37] U Feige, J Kilian, and M Naor. [n. d.]. A minimal model for secure computation extended abstract. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*.

[38] Zhitao Guan, Yue Zhang, Liehuang Zhu, Longfei Wu, and Shui Yu. 2019. EFFECT: An efficient flexible privacy-preserving data aggregation scheme with authentication in smart grid. *Science China Information Sciences* 62 (2019), 1–14.

[39] Yan Huang, David Evans, and Jonathan Katz. 2012. Private set intersection: Are garbled circuits better than custom protocols?. In *NDSS*.

[40] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster secure {Two-Party} computation using garbled circuits. In *20th USENIX Security Symposium (USENIX Security 11)*.

[41] Siam Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. 2020. TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*. 65–67.

[42] Marek Jawurek and Florian Kerschbaum. 2012. Fault-tolerant privacy-preserving statistics. In *Privacy Enhancing Technologies: 12th International Symposium, PETS 2012, Vigo, Spain, July 11-13, 2012. Proceedings 12*. Springer, 221–238.

[43] Marc Joye and Benoît Libert. 2013. A scalable scheme for privacy-preserving aggregation of time-series data. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17*. Springer, 111–125.

[44] Seny Kamara, Payman Mohassel, and Mariana Raykova. 2011. Outsourcing multi-party computation. *Cryptology ePrint Archive* (2011).

[45] 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.

[46] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security: 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings 8*. Springer, 1–20.

[47] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*. Springer, 486–498.

[48] Ben Kreuter, Abhi Shelat, Benjamin Mood, and Kevin Butler. 2013. {PCF}: A Portable Circuit Format for Scalable {Two-Party} Secure Computation. In *22nd USENIX Security Symposium (USENIX Security 13)*. 321–336.

[49] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. 2011. Privacy-friendly aggregation for the smart-grid. In *Privacy Enhancing Technologies: 11th International Symposium, PETS 2011, Waterloo, ON, Canada, July 27-29, 2011. Proceedings 11*. Springer, 175–191.

[50] Leslie Lamport. 1984. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Trans. Program. Lang. Syst.* 6, 2 (1984), 254–280.

[51] Leslie Lamport. 1987. Synchronizing time servers. (1987).

[52] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptol.* 22, 2 (2009), 161–188.

[53] Yehuda Lindell and Benny Pinkas. 2012. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology* 25 (2012), 680–722.

[54] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 359–376.

[55] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security Symposium*. USENIX, 287–302.

[56] Keith Marzullo. 1990. Tolerating Failures of Continuous-Valued Sensors. *ACM Trans. Comput. Syst.* 8, 4 (1990), 284–304.

[57] Keith Marzullo and Susan Owicki. 1983. Maintaining the time in a distributed system. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*. 295–305.

[58] Keith Ansel Marzullo. 1984. *Maintaining the time in a distributed system: an example of a loosely-coupled distributed service (synchronization, fault-tolerance, debugging)*. Ph. D. Dissertation. Stanford University.

[59] Michael T McClellan and Jack Minker. 1974. The art of computer programming, vol. 3: sorting and searching.

[60] Luca Melis, George Danezis, and Emiliano De Cristofaro. 2015. Efficient private statistics with succinct sketches. *arXiv preprint arXiv:1508.06110* (2015).

[61] Silvio Micali, Oded Goldreich, and Avi Wigderson. 1987. How to play any mental game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*. ACM New York, NY, USA, 218–229.

[62] Jianqiao Mo, Jayanth Gopinath, and Brandon Reagen. 2023. HAAC: A Hardware-Software Co-Design to Accelerate Garbled Circuits. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[63] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[64] Benjamin Mood, Debayan Gupta, Kevin Butler, and Joan Feigenbaum. 2014. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 582–596.

[65] Moni Naor, Benny Pinkas, and Reuban Sumner. 1999. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*. 129–139.

[66] Jesper Buus Nielsen and Claudio Orlandi. 2009. LEGO for two-party secure computation. In *Theory of Cryptography Conference*. Springer, 368–386.

[67] Bartosz Przydatek, Dawn Song, and Adrian Perrig. 2003. SIA: Secure information aggregation in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*. 255–265.

[68] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia conference on computer and communications security*. 707–721.

[69] Phillip Rogaway and Tom Shrimpton. 2006. Deterministic Authenticated-Encryption: A Provable-Security Treatment of the Keywrap Problem. EUROCRYPT 2006.

[70] Cristina Rottondi, Giacomo Verticale, and Christoph Krauss. 2013. Distributed privacy-preserving aggregation of metering data in smart grids. *IEEE Journal on Selected Areas in Communications* 31, 7 (2013), 1342–1354.

[71] JE Savage. 1998. Models of computation. Exploring the power of computing Addison-Wesley. *Reading, MA* (1998).

[72] Ulrich Schmid and Klaus Schossmaier. 2001. How to reconcile fault-tolerant interval intersection with the Lipschitz condition. *Distributed Computing* 14 (2001), 101–111.

[73] Elaine Shi, HTH Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-preserving aggregation of time-series data. In *Annual Network & Distributed System Security Symposium (NDSS)*. Internet Society.

[74] Jinhyun So, Ramy E Ali, Başak Güler, Jiantao Jiao, and A Salman Avestimehr. 2023. Securing secure aggregation: Mitigating multi-round privacy leakage in federated learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 9864–9873.

[75] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. 2015. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 411–428.

[76] Samet Tonyali, Kemal Akkaya, Nico Saputro, A Selcuk Uluagac, and Mehrdad Nojoumian. 2018. Privacy-preserving protocols for secure and reliable data aggregation in IoT-enabled smart metering systems. *Future Generation Computer Systems* 78 (2018), 547–557.

[77] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 39–56.

[78] Xinyu Xing, Wei Meng, Dan Doozan, Alex C. Snoeren, Nick Feamster, and Wenke Lee. 2013. Take This Personally: Pollution Attacks on Personalized Services. In *USENIX Security Symposium*. USENIX Association, 671–686.

[79] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*. IEEE Computer Society, 162–167.

[80] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.* (2015), 1153.

[81] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT (2) (Lecture Notes in Computer Science, Vol. 9057)*. Springer, 220–250.

[82] Shuai Zhao, Fenghua Li, Hongwei Li, Rongxing Lu, Siqi Ren, Haiyong Bao, Jian-Hong Lin, and Song Han. 2020. Smart and practical privacy-preserving data aggregation for fog-based smart grids. *IEEE Transactions on Information Forensics and Security* 16 (2020), 521–536.